NO STRESS NO GAIN: STRESS TESTING BASED SELF-CONSISTENCY FOR OLYMPIAD PROGRAMMING

Kunal Singh,* **Sayandeep Bhowmick,*** **Pradeep Moturi, Siva Kishore Gollapalli**
Fractal AI Research
Mumbai, India
{kunal.singh}@fractal.ai

ABSTRACT

We introduce a stress testing approach to improve performance of large language reasoning models on challenging competitive programming problems. By combining stress testing—inspired from a technique commonly used by expert programmers—with self-consistency and self-debugging methods, we demonstrate significant improvements in solution accuracy. Our method generates multiple brute-force solutions to validate and filter candidate solutions, leading to better performance than traditional majority voting approaches. Experimental results show that our approach successfully narrows the gap between pass@k and majority voting scores on the USACO benchmark for both o1-mini and o3-mini models, solving up to 246 out of 307 problems which is 17 more than the vanilla self-consistency.

1 INTRODUCTION

In recent years, Large Language Models (LLMs) have demonstrated remarkable capabilities in a wide array of reasoning and problem-solving tasks, including code generation. Scaling training-time compute has led to creation of Frontier system 1 (Guan et al., 2025) models such as GPT-4o(OpenAI, June, 2024), Claude 3.5 Sonnet(Anthropic, 2023) that have completely saturated coding benchmarks such as HumanEval(Chen et al., 2021) and MBPP(Austin et al., 2021). In addition to scaling pretraining, studies have also tried augmenting inference time compute/methods to improve code generation robustness. While methods such as COT(Wei et al., 2022), self-consistency(Wang et al., 2022), reflexion (Shinn et al., 2023) have shown to improve general reasoning ability, relevant studies in code generation domain have additionally focused on iterative self-improvement(refinement) by leveraging execution feedback. Self-debugging(Chen et al., 2023), self-edit(Zhang et al., 2023), self-refine(Madaan et al., 2023), in a similar fashion, use sample input-output pairs as unit test cases to generate execution output of the code solution and debug/refine the program similar to how a developer/programmer would debug. These methods have shown promising improvements on simpler coding benchmarks such as HumanEval and MBPP.

Due to saturation of performance on simpler code generation benchmarks, competitive programming has emerged as the new test-bed to benchmark the relevant models and methods. LiveCodeBench(Naman Jain, 2024) presents competitive programming problems of varying difficulties sourced from platforms such as LeetCode, AtCoder, and CodeForces. Further escalating the difficulty of the benchmarks, another study introduces USACO benchmark (Shi et al., 2024) with 307 highly challenging problems from past USA Computing Olympiad (USACO) competitions. Computing Olympiads contain some of the most challenging problems for humans, requiring complex algorithmic reasoning, puzzle solving, in addition to generating efficient code. Recent works such as (Wang et al., 2024; Naman Jain, 2024; Shi et al., 2024) have shown that the frontier system 1 LLMs have struggled to solve these challenging problems. They even point out that despite augmenting advanced inference time methods, there is not much accuracy gain.

However, the recent emergence of system 2 (Guan et al., 2025) reasoning models such as o1 (OpenAI, 2024a), DeepSeek R1 (DeepSeek-AI et al., 2025) and o3(OpenAI, 2024b) have shown major

*Equal contribution (alphabetical order)

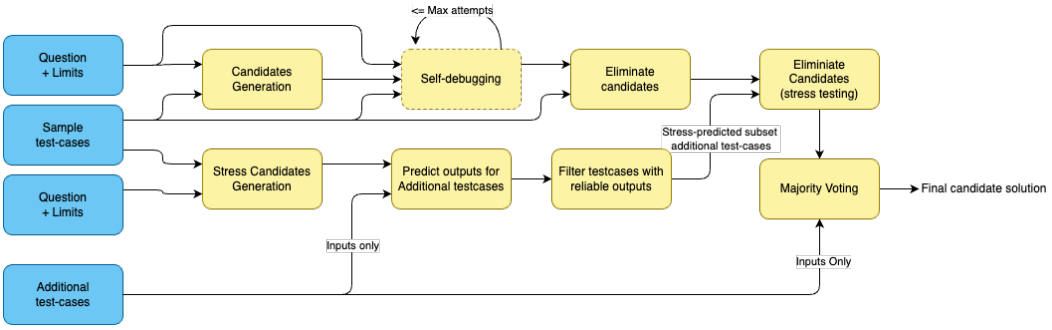

Figure 1: Overview of the stress testing based self-consistency, with a combination with self-debugging. For a given problem, we generated multiple candidate solutions. An optional step: When using self-debugging, we iterative to refine the generated solutions(that fail the sample test-cases) for maximum set attempts. Post this, we have first round of elimination where those candidate solutions that fail to pass all the sample test-cases are eliminated. In parallel to this, we also generate stress candidates which are brute-force based solution to the given problem. We use the stress candidates to predict the outputs of the additional test-cases provided by the benchmark. We select those test-cases that have reliable outputs (mode 2 or more). This process of output generation and selection helps in the creation of stress-predicted subset test-cases using only the inputs of the additional test-cases. Now, we do our second round of elimination (stress-testing) and eliminate the candidate solutions that fail to pass all the stress predicted test-cases. Final candidate solution is selected from the remaining candidate solutions through the process of majority voting. Note: We enforce the time limits defined in the benchmark for each test case.

step level improvement in the ability of LLMs in solving harder competitive programming problems. In particular, o1-IOI (OpenAI) performs impressively on live IOI competition and relies on inference time augmentation framework to make o1 generate synthetic and test-cases. Though, the details of the framework is not revealed but it clearly shows the need for further-test time augmentation and that test-cases could be useful for system 2 reasoning models as well.

In this study, we try to make the following the contributions: a) We show and observe a big gap in the pass@k and maj@k scores on USACO benchmark with o1-mini and o3-mini. We make an attempt to close this gap through the efforts mentioned below. b) Stress testing(Ibrahim) is a very popular technique used by top human coders to debug their codes. Inspired by it, we implement stress testing to improve self-consistency by leveraging the additional test cases provided by the benchmark. c) We further combine self-debugging (Chen et al., 2023) (unit test feedback) to iteratively improve the solutions that fail on sample input-output test cases.

2 METHOD

We briefly introduce common practices in code generation task and then introduce our stress testing method.

2.1 BACKGROUND

Candidate generation: LLM-based code generation can be a complex task and the LLMs can give the correct solutions only with a certain probability. Given a problem statement x, and sample test cases $\{(i_k, o_k)\}_{k=1}^{K}$, we prompt the model L, to output independent candidate solutions $\{y_j\}_{j=1}^{N}$ following self-consistency (Wang et al., 2022). We maintain entire conversation chains $\{c_j\}_{j=1}^{N}$. [1]

$$y = L(x \oplus \{(i_k, o_k)\}_{k=1}^{K}) \tag{1}$$

Self-debugging: Given the probabilistic nature of LLMs, the initial code outputs may not always be fully accurate. Chen et al. (2023) executes the generated codes $\{y_i\}_{i=1}^{N}$ against sample test case

[1] $\oplus$ denotes concatenation

$\{(i_k, o_k)\}_{k=1}^{K}$ using a program executor P in a sandbox environment. Errors on any test cases are appended to the conversation c and then prompt the LLM to debug itself and update it's solution. This is repeated till all sample test cases are passed or maximum attempt limit is breached. Consequently, we filter out candidates that fail to solve the provided test cases.

$$c_j = c_j \oplus \{(i_k, P(y_j, x_k), o_k), \forall P(y_j, i_k) \neq o_k\}_{k=1}^{K} \tag{2}$$

$$y_j = L(c_j) \tag{3}$$

Majority voting: We need to select a single solution out of all the candidates. We do this by clustering the candidates on their outputs of the additional test cases $\{(\hat{i}_k, \hat{o}_k)\}_{k=1}^{\hat{K}}$. Candidates having same outputs to the additional testcases are considered to be part of the same cluster. We pick a random candidate from the largest cluster as the final solution.

2.2 Stress testing

From Table 1, We see a significant gap between pass scores (at-least one solution out of all candidate generations is a correct solution) and majority voting. This informs us that the model is able to generate correct solution but we are unable to select it as final solution with just majority voting. Approaches (Chen et al., 2022) have tried using LLMs and other models like CYaRon to generate test cases to add more reasoning and improve self debugging stage. We build upon this and use the additional test cases from the benchmark itself utilizing only the inputs to avoid the noise(test-cases with wrong/invalid inputs) from LLM based test case input generation.

Stress testing is a popular debugging and validation technique employed by top programmers to ensure the correctness of their solutions. This method involves implementing brute-force solutions, which serve as a reliable benchmark for verifying the accuracy of more optimized approaches. Drawing inspiration from this classical technique, we extend its application to LLMs to enhance their problem-solving capabilities. Our experiments demonstrate that LLMs are proficient at generating brute-force code, a skill that we systematically leverage to assess and filter candidate solutions.

We apply *candidate generation* to generate S brute force solutions termed **stress candidates**. We eliminate candidates which fail any sample test case. With remaining candidates, we aim to predict the ground truth outputs of additional test cases. We obtain outputs for all the test cases with each candidate using program executor P. For each test case, we consider the mode of outputs with repetitions greater than 1 as the predicted ground truth else we discard the test case. We obtain a stress-predicted subset from the entire additional test-cases.

Stress-testing flow, end to end: We perform multiple *candidate generation* for best program output. We eliminate candidates which fail any sample test case. Then, we generate stress candidates and generates outputs for the additional test-cases and obtain a stress-predicted subset of test cases. Now, we perform stress-testing, that is, we eliminate those candidates that don't pass any of the test cases from the stress-predicted subset. Then, we perform *majority voting* using all additional test cases on the remaining candidates to select the best solution. We show the entire process in Figure 1

3 Experiments

3.1 Dataset

We evaluate on USACO (Shi et al., 2024) benchmark with 307 problems from the USA Computing Olympiad. The dataset consists of questions of four difficulty levels - bronze, silver, gold and platinum. Each problem consists of a problem description with the input and the output formats; 0-2 sample tests along with their explanations in some cases; 10-17 additional tests for verifying solution correctness; time and memory limits verifying solution complexity; and an official human-written problem analysis explaining the solution in detail with corresponding Python/C++ code.

3.2 Results

All the experiments are run using the OpenAI APIs. We set reasoning effort parameter to medium for o3-mini. We enforce the time limits defined in the benchmark for each test case.

Table 1: Number of problems solved out of 307 from USACO dataset under different settings. SD max-attempts: maximum refinement attempts allowed per candidate in self debugging stage

Candidates	SD max-attempts	Stress candidates	o1-mini	o3-mini
pass@4	-	-	202	231
pass@9	-	-	220	242
pass@16	-	-	224	253
pass@25	-	-	234	259
Majority voting				
4	-	-	191	217
9	-	-	195	221
16	-	-	201	226
25	-	-	204	229
9	-	3	206	230
9	-	5	210	234
9	-	7	212	235
9	-	9	213	236
16	-	3	210	237
16	-	5	213	239
16	-	7	214	243
16	-	9	**217**	<u>244</u>
9	1	-	204	237
9	3	-	207	240
9	5	-	209	241
9	1	5	210	243
9	3	5	215	<u>244</u>
9	5	5	<u>216</u>	**246**

Table 1 presents the number of correctly solved questions using different approaches. The results show that there is a big gap between the pass and majority scores. The baseline majority score for maj@9 is 195 for o1-mini and 221 for o3-mini. We run self-debugging and observe that there is a significant improvement and we acheive scores of 209(o1-mini) and 241(o3-mini). These scores are even higher than maj@25 scores for both the models.

Applying stress testing on 9 chains without self-debugging gives us a score of 213 compared to 195 with majority voting with o1-mini and 236 compared to 221 with majority voting with o3-mini. This significant improvement is also observed with 16 chains. These scores are also more than the maj@25 scores for both the models. Using both the methods together, we get highest scores of 216 and 246 for o1-mini and o3-mini respectively which is approximately a 8% point improvement on the baseline and a 5% point improvement on the maj@25 scores.

4 Ablations

4.1 Stress test accuracy

We verify the accuracy of the predicted ground truths of the additional test cases with the ground truths from the dataset. We are able to obtain an accuracy of 91% with o1-mini and an accuracy of 97% with o3-mini. Furthermore, we explore the efficacy of the stress solutions in filtering the candidate codes. We observe that in 80% cases for o1-mini and 84% cases for o3-mini, the stress solutions help us in filtering the solutions.

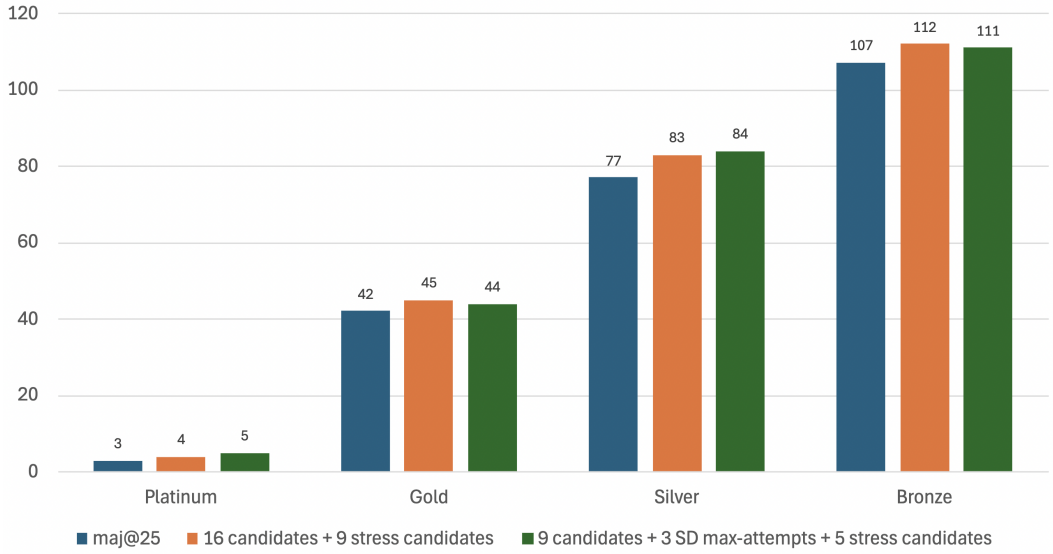

Figure 2: Number of questions solved in different difficulty levels in USACO benchmark. SD max-attempts: maximum refinement attempts allowed per candidate in self debugging stage

4.2 Difficulty Analysis

The distribution of difficulty level in the dataset is platinum(21), gold(63), silver(100) and bronze(123). Platinum and gold are the hardest with similar difficulty level as IOI problems. We observe improvements across all difficulty levels. Moreover, stress testing provides great improvement even in the harder problems as is evident from the Figure 2.

5 Conclusion

Our experiments on the USACO benchmark demonstrate significant improvements over baseline majority voting methods, with o3-mini achieving particularly strong results. The high accuracy of predicted ground truths (97% for o3-mini and 91% for o1-mini) validates the effectiveness of our stress testing approach. Furthermore, the stress solutions proved valuable in filtering candidate solutions, successfully doing so in 84% of cases for o3-mini and 80% for o1-mini.

References

Anthropic. Introducing claude 3.5, 2023. URL https://www-cdn.anthropic.com/fed9cc193a14b84131812372d8d5857f8f304c52/Model_Card_Claude_3_Addendum.pdf.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests, 2022. URL https://arxiv.org/abs/2207.10397.

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. Evaluating large language models trained on code. 2021.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug, 2023. URL https://arxiv.org/abs/2304.05128.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. URL https://arxiv.org/abs/2501.12948.

Xinyu Guan, Li Lyna Zhang, Yifei Liu, Ning Shang, Youran Sun, Yi Zhu, Fan Yang, and Mao Yang. rstar-math: Small llms can master math reasoning with self-evolved deep thinking, 2025. URL https://arxiv.org/abs/2501.04519.

Ali Ibrahim. Stress testing. URL https://ali-ibrahim137.github.io/competitive/programming/2020/08/23/Stress-Testing.html.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Sean Welleck, Bodhisattwa Prasad Majumder, Shashank Gupta, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback, 2023.

Alex Gu Wen-Ding Li Fanjia Yan Tianjun Zhang Sida Wang Armando Solar-Lezama Koushik Sen Ion Stoica Naman Jain, King Han. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint*, 2024.

OpenAI. Openai o1-ioi. URL https://neurips.cc/virtual/2024/108302.

OpenAI. o1, 2024a. URL https://openai.com/o1/.

OpenAI. o3-mini, 2024b. URL https://openai.com/index/openai-o3-mini/.

OpenAI. "hello gpt-4o.", June, 2024. URL https://openai.com/index/hello-gpt-4o/.

Quan Shi, Michael Tang, Karthik Narasimhan, and Shunyu Yao. Can language models solve olympiad programming?, 2024. URL https://arxiv.org/abs/2404.10952.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023. URL https://arxiv.org/abs/2303.11366.

Evan Wang, Federico Cassano, Catherine Wu, Yunfeng Bai, Will Song, Vaskar Nath, Ziwen Han, Sean Hendryx, Summer Yue, and Hugh Zhang. Planning in natural language improves llm search for code generation, 2024. URL https://arxiv.org/abs/2409.03733.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Huai hsin Chi, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *ArXiv*, abs/2203.11171, 2022. URL https://api.semanticscholar.org/CorpusID:247595263.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Huai hsin Chi, F. Xia, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. *ArXiv*, abs/2201.11903, 2022. URL https://api.semanticscholar.org/CorpusID:246411621.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. Self-edit: Fault-aware code editor for code generation. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (eds.), *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 769–787, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.45. URL https://aclanthology.org/2023.acl-long.45/.