arXiv:2601.13682v1 [cs.SE] 20 Jan 2026

CodeContests-O：通过反馈驱动的迭代测试用例生成赋能大语言模型

蔡剑锋† 朱锦华*† Ruopei Sun† Kangwen Zhao†

Dongyun Xue† Mingxiao Feng† Wengang Zhou† Houqiang Li†

† 中国科学技术大学

{xiaobaicai,teslazhu}@mail.ustc.edu.cn

摘要

推理模型的兴起催生了对大规模可验证数据的需求，而编程任务正是此类数据的理想来源。然而，尽管竞赛编程平台提供了大量题目和解答，用于验证的高质量测试用例仍然稀缺。现有方法尝试使用大语言模型（LLM）合成测试用例，但仅依赖模型内在的生成能力，缺乏外部反馈，因此生成的用例往往不够多样。为解决这一局限，我们提出一种**反馈驱动的迭代框架**，用于全面构造测试用例。具体而言，我们的方法利用大语言模型生成初始测试用例，将其应用于已知的正确解答和错误解答，并利用失败结果作为反馈，引导大语言模型不断完善测试用例，提升其保真度和可区分性。随后，我们将该方法应用于 CodeContests 数据集，构建了经过优化的高质量衍生数据集**CodeContests-O**。在全部解答池（共1.1×10^7 个）上进行评估后，我们的数据集平均真正例率（TPR）达到 89.37%，平均真负例率（TNR）达到 90.89%，分别比 CodeContests 和 CodeContests+ 高出 4.32% 和 9.37%。此外，使用 CodeContests-O 对 Qwen2.5-7B 模型进行微调后，其在 LiveCodeBench 上的表现（Pass@1）提升了 9.52%。实验结果证明了该框架的有效性以及 CodeContests-O 的质量。为支持结果复现并促进未来研究，我们发布了代码[1] 和数据集[2]。

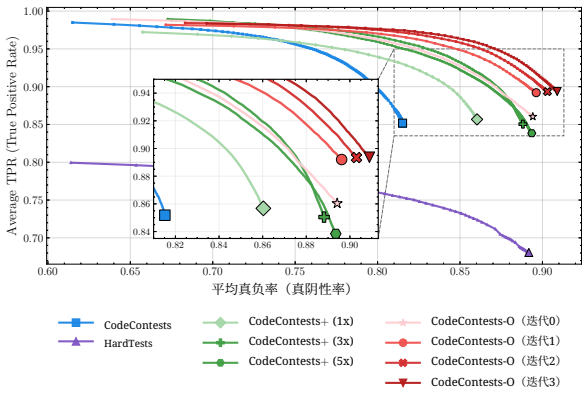

图1：不同数据集的真正例率（TPR，接受的正确解答所占比例，用于衡量测试用例保真度）和真负例率（TNR，拒绝的错误解答所占比例，用于衡量区分能力）的帕累托前沿。我们的 CodeContests-O 始终优于以往的数据集，展现出更优异的测试用例质量。迭代优化过程（迭代0–3）逐步提升了这两项指标，验证了我们以反馈驱动方法的有效性。

1 引言

近年来，以推理为中心的大语言模型（LLMs）（，2024年；Guo 等，2025年；Anthropic，2025；DeepMind，2025a，b；Yang 等人，2025）标志着人工智能的一次重要演进，展现出解决复杂逻辑和数学问题的空前潜力（El-Kishky 等人，2025年；Wang 等人，2025a）。这一进步催生了对大规模可验证数据的迫切需求，而编程任务因其严谨的逻辑和客观的可验证性，成为理想的数据来源（Lightman 等人，2023年；Ni 等人，2023年）。然而，一个显著的瓶颈依然存在：尽管竞技编程平台（Mirzayanov 等人，2020年）提供了大量题目描述以及相应的正确解答和错误解答，但验证所必需的高质量测试用例往往十分稀缺或难以获取。这一缺陷使得人们难以可靠地区分真正的推理与“幻觉式”解答；这类解答看似正确，却无法通过细微的边界情况，最终限制了

*通讯作者。[1]

[1] https://github.com/cai-jianfeng/CodeContests-O

[2] https://huggingface.co/datasets/caijanfeng/CodeContests- O

模型训练和评估的有效性 .

为解决测试用例匮乏的问题，先前研究探索了多种自动生成测试用例的方法。早期工作依赖人工整理的小规模基准，例如 HumanEval（Chen，2021年）和 MBPP（Austin 等人，2021年），或采用基于变异的增强，AlphaCode（Li 等人，2022）和 EvalPlus（Liu 等人，2023年）便是此类方法的代表，它们通过基于规则的变体扩充现有测试用例池。近来，研究者转向直接使用大语言模型生成，借助模型的内在知识，在 CodeT（Chen 等人，2022）和 TACO（Li 等人，2023年）等框架中直接根据问题合成测试用例。为进一步提高可靠性，CodeContests+（Wang 等人，2025c）和 AutoCode（Zhou 等人，2025年）等近期方法采用生成器-验证器范式，合成专用的生成器程序和验证器程序，以自动创建大规模测试用例。

尽管取得了这些进展，现有方法仍存在显著局限。它们遵循一种开放环范式，完全依赖大语言模型的内在能力，缺乏纳入**客观外部反馈**并进行**迭代优化**的机制。由于缺乏客观的反馈信号，这些方法无法捕捉在针对广泛解答池执行测试时所暴露的各种边界情况和动态失败。因此，生成的测试用例往往存在冗余或区分能力不足的问题，限制了它们在训练和评估稳健推理模型方面的效用。

为克服这些根本性局限，我们提出了一种新型框架，将测试用例合成从开放环生成转变为反馈驱动的迭代闭环过程。我们的方法以一个由三个系统化阶段组成的迭代优化循环为核心。首先，在初始测试用例生成阶段，我们采用生成器—验证器范式，由大语言模型分析问题约束，以开发生成器程序和一系列执行命令，确保结构完整性和多样性。其次，在执行与反馈收集阶段，将合成的测试用例针对多样化的解池执行，从中捕获包括假阳性、假阴性和执行错误日志在内的细粒度信号，以量化判别能力。最后，在反馈引导的优化阶段，框架对这些失败进行根因分析，通过“搜索与替换”机制，有策略地演进生成器及其命令。通过迭代闭合生成与执行之间的环，我们的方法确保生成的测试用例不仅有效，而且能够以高度严谨的方式暴露细微的算法缺陷。

借助这一框架，我们进一步生成了 CodeContests-O，这是一个源自 CodeContests 的高质量、可验证代码数据集（Wang 等人，2025年c）。通过系统地应用我们由反馈驱动的迭代优化框架，我们合成了大规模测试用例，达到了较高的真阳性率（TPR）和真负例率（TNR）。CodeContests-O 为推理大语言模型的训练和评估提供了高保真度的验证信号，有效地区分真正的逻辑推理与肤浅的模式匹配；后者往往无法应对复杂的边界情况。

实证评估表明，CodeContests-O 质量卓越，并在提升下游强化学习训练方面发挥着关键作用。通过针对完整解答池（1.1×10^7 份解答）进行广泛评估，我们迭代优化后的测试用例实现了 89.37% 的平均真正例率（TPR）和 90.89%的平均真负例率（TNR），显著优于原始的 CodeContests 和增强版 CodeContests+。此外，将 CodeContests-O 用作强化学习训练的奖励信号时，其在 LiveCodeBench（Jain 等人，2024年）上的表现大幅提升了 9.52%，尤其是在解决复杂且逻辑密集型问题方面。这些结果证实，相较于测试用例的数量，测试用例的质量才是提升大语言模型推理能力的主要驱动因素。

我们的贡献总结如下 ：

- 我们提出了一种由反馈驱动的迭代生成框架，能够系统地合成、验证和优化测试用例，有效提升其保真度和区分能力，从而确保高质量验证。

- 我们推出 CodeContests-O，这是一个包含经过迭代优化的测试用例的大规模数据集，可提供严格的评估标准，从而更准确地区分真正的逻辑推理与表面上的模式匹配。

- 我们提供了全面的实证分析，证明从我们的数据集中提取的高精度奖励信号显著提升了下游强化学习训练，并在 LiveCodeBench 的所有难度级别上带来了稳定的性能提升。

2 相关工作

本节回顾了推理大语言模型测试用例构造的发展历程，涵盖从人工整理和自动化合成到基于大语言模型的程序生成方法。

人工整理与基于变异的增强。 早期具有奠基意义的基准测试，如 HumanEval（Chen，2021年）和 MBPP（Austin 等人，2021年），依赖高质量、由人工编写的测试用例。为缓解数据污染并跟上模型的快速迭代，LiveCodeBench（Jain 等人，2024年）通过收集近期竞赛中的题目，提供了一个全面且定期更新的平台。尽管这些数据集可靠，但其规模有限，难以捕捉竞赛编程的复杂性。为突破人工工作的限制，扩大测试用例生成规模，一些自动化方法结合了变异测试等传统技术。具体而言，AlphaCode（Li 等人，2022）和 EvalPlus（Liu 等人，2023年）都采用基于变异的方法生成大量测试用例，从而揭示那些原本可能通过较简单人工编写测试的解决方案中存在的漏洞。尽管这些方法增加了测试用例的数量，但从根本上受限于初始测试池，无法合成复杂的边界情况。

直接由大语言模型生成。 为克服静态测试用例的局限性，其他研究利用大语言模型的生成能力直接合成测试用例。CodeT（Chen 等人，2022）和 TACO（Li 等人，2023年）利用大语言模型的内在知识直接生成测试用例。同样，ChatTESTER（Yuan 等人，2023年）和 TestAug（Yang 等人，2022）展示了大语言模型在扩增测试用例方面的潜力。Co- daMosa（Lemieux 等人，2023年）采用混合方法，将大语言模型与基于搜索的软件测试（SBST）相结合，以提高测试覆盖率。然而，这些直接生成方法往往存在多样性不足的问题，且缺乏确保生成用例有效性的机制。

基于大语言模型的程序生成。 近期研究探索了生成器-验证器范式，利用大语言模型合成专用程序，以自动创建测试用例。CodeCon-tests+ (Wang 等人，2025年c) 和 AutoCode (Zhou 等人，2025年) 采用生成器-验证器系统，通过为每个问题合成专用的生成器和验证器程序，自动完成测试用例的创建，以生成并验证测试用例。类似地，rStar-Coder (Liu 等人，2025年) 利用相互验证来构建大规模的经验证数据集，从而扩展了这一范式。为了针对复杂的算法边界情况，HardTests (He 等人，2025年) 专注于合成专门针对时间限制和逻辑密集型约束的“刁钻”用例。该领域还开发了 TestEval (Wang 等人，2025年b) 等专用基准，并开展了关于基于大语言模型的测试生成器可靠性的研究 (Cao 等人，2025年)，以评估这些系统的判别能力。然而，这些程序化方法通常依赖内部一致性或一次性检查。相比之下，我们的工作提出了一种反馈驱动的迭代流程。通过将已知正确解答和错误解答的实际执行结果用作动态反馈信号，我们引导大语言模型迭代改进测试用例，以确保其质量。

3 方法

3.1 问题表述

典型的竞赛编程题由自然语言描述、参考解、一组公开测试用例，以及一系列标注了正确性的用户提交解答构成。形式上，我们将此类问题表示为五元组 $\mathcal{P} = (P,\ S^*,\ S^+,\ S^-,\ \mathcal{T}^p)$，其中 P是题目描述，S^* 是官方参考解，$S^+ = \{s_1^+, s_2^+, \ldots, s_m^+\}$ 是已知正确解答的集合，$S^- = \{s_1^-, s_2^-, \ldots, s_n^-\}$是已知错误解答的集合，$\mathcal{T}^p = \{t_1^p, t_2^p, \ldots, t_k^p\}$ 是题目附带的一小组公开测试用例。每个测试用例 t_i由输入 x_i 和预期输出y_i组成。我们的主要目标是合成一组高质量、全面的测试用例 $\mathcal{T}^e = \{t_1^e, t_2^e, \ldots, t_l^e\}$，使其具备出色的区分能力，从而更可靠地验证生成的代码。为确保验证的严谨性，每个测试用例 $t_j^e = (x_j^e,\ y_j^e) \in \mathcal{T}^e$都必须满足两个基本标准。首先，必须保持保真度：每个正确解答 $s_i^+ \in S^+$ 在输入 x_j^e下执行时，都必须产生预期输出 y_j^e。其次，生成的测试用例必须确保可区分性：对于每个错误解答$s_i^- \in S^-$，至少应存在一个测试用例$t_j^e = (x_j^e, y_j^e) \in \mathcal{T}^e$，使其在输入 x_j^e下执行 s_i^-时，产生与预期输出 y_j^e不一致的结果。满足这些条件后，合成的测试用例便可作为严格的验证工具，用于模型训练和评估。

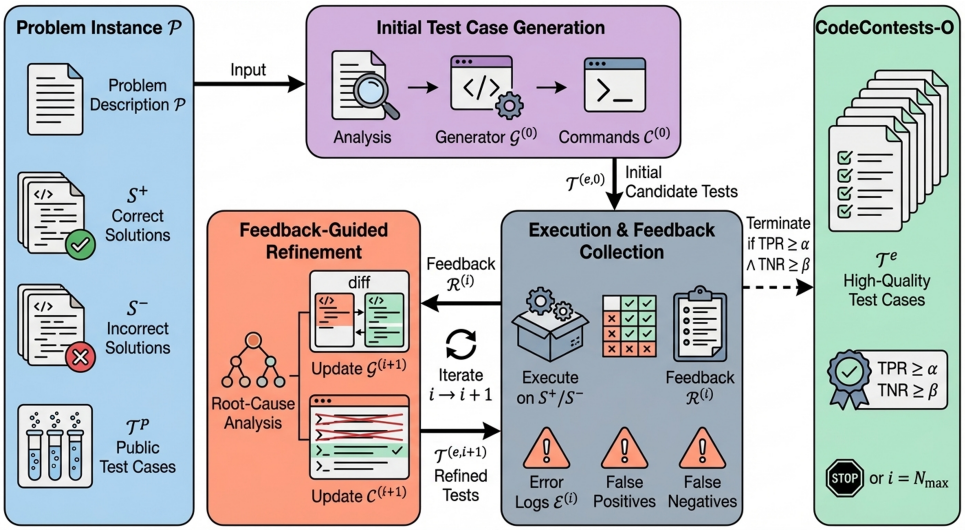

图 2：反馈驱动的迭代测试用例生成概览。该框架首先进行初始测试用例生成，分析问题描述 P 以生成生成器 $\mathcal{G}^{(0)}$ 和命令 $\mathcal{C}^{(0)}$，随后执行它们，合成初始候选测试用例 $\mathcal{T}_{(e,0)}$。接下来，执行与反馈收集 和反馈引导的优化之间会持续循环。在每次迭代 i中，使用 $S^{+/-}$ solutions 对测试用例 $\mathcal{T}_{(e,i)}$ 进行评估，从而提炼出结构化反馈报告 $\mathcal{R}^{(i)} = \{\mathcal{F}^{(i)}, \mathcal{E}^{(i)}\}$，其中$\mathcal{F}^{(i)}$ 用于识别假阳性/假阴性， $\mathcal{E}^{(i)}$ 则记录执行错误日志。随后，大语言模型会对 $\mathcal{R}^{(i)}$ 进行根因分析，以优化生成逻辑。当质量阈值（TPR、TNR）达标或达到最大迭代次数 N_{max} 时，流程即告终止，最终生成 CodeContests-O 数据集。

3.2 反馈驱动的迭代测试用例生成

所提出的高质量测试用例合成框架包含三个主要阶段：初始测试用例生成、执行与反馈收集，以及反馈引导的优化。图 2概述了该框架。

初始测试用例生成。 为建立高质量的初始测试用例池，我们采用生成器-验证器范式，该范式包括以下三个连续步骤。

首先，我们利用大语言模型对问题 P进行全面分析，同时识别结构化的输入输出约束，并预判常见的算法陷阱，例如整数溢出、边界条件或效率瓶颈。此阶段会生成一份系统化摘要，通过深入理解任务要求和潜在的边缘情况场景，为生成过程提供指导。

其次，大语言模型根据分析摘要开发专用的生成器程序 $\mathcal{G}^{(0)}$，以确保生成的测试输入具备结构完整性和有效性。与静态文本生成不同，这种程序化方法利用外部参数精确控制数据属性，使我们能够系统地调整范围和约束条件，从而为各种场景创建多样且严谨的测试输入。

最后，大语言模型制定一系列执行命令 $\mathcal{C}^{(0)}$，每条命令都为生成器定义一组独特的参数。通过在这些不同参数下运行生成器，并利用参考解 S^* 生成相应的真实输出，我们构建初始测试用例集 $\mathcal{T}_{(e,0)} = \{t_1^{(0)}, t_2^{(0)}, \ldots, t_{l_0}^{(0)}\}$。

执行与反馈收集。 合成初始测试用例后，我们会评估其质量，为后续改进提供精确的信号。具体而言，我们使用解池 S^+ 和 S^-执行候选测试用例 $\mathcal{T}_{(e,i)}$，并记录每个正确解答 $s_k^+ \in S^+$ 和错误解答 $s_m^- \in S^-$的执行结果。通过将这些输出与参考解 S^*的真实输出进行验证，我们量化每个测试用例的判别能力。如果某个测试用例能够成功验证更大比例的正确解答，同时检测出更多错误解答中的潜在缺陷，则认为它更有效。

随后，这些结果被提炼为一份全面的反馈集 $\mathcal{F}^{(i)}$，其中明确指出假阴性（被错误拒绝的正确解答）和假阳性（绕过测试用例的错误解答）。这些失败揭示了

当前测试用例在区分细微算法差异方面的局限性。此外，当生成器 $\mathcal{G}^{(i)}$ 或参考解答 S^* 无法生成有效输出时，我们会捕获相应的执行错误日志和堆栈跟踪，记为 $\mathcal{E}^{(i)}$。这些辅助信息使大语言模型能够诊断生成过程中潜在的结构性缺陷或运行时崩溃。最后，我们将这些观察结果汇总为一份结构化反馈报告 $\mathcal{R}^{(i)} = \{\mathcal{F}^{(i)}, \mathcal{E}^{(i)}\}$，为大语言模型提供必要的上下文，以便改进并合成质量更高的测试用例。

反馈引导的优化。在每次迭代 i的最后阶段，我们利用结构化反馈报告 $\mathcal{R}^{(i)}$，引导大语言模型有针对性地优化测试用例生成逻辑。这一过程并非简单地重新生成，而是对生成器程序$\mathcal{G}^{(i)}$ 及其执行命令 $\mathcal{C}^{(i)}$进行策略性演进。

首先，大语言模型会对 $\mathcal{R}^{(i)}$ 中记录的失败进行根因分析。通过检查假阳性、假阴性和执行错误$\mathcal{E}^{(i)}$，模型能够识别当前生成逻辑中的具体弱点。基于这些发现，大语言模型会更新生成器程序，使其 $\mathcal{G}^{(i+1)}$优化内部逻辑，以捕获此前被忽略的边界条件和复杂的边界情况。同时，大语言模型会重新设计执行命令 $\mathcal{C}^{(i)}$，将其调整为 $\mathcal{C}^{(i+1)}$，以探索参数空间中尚未探索的区域，确保新的测试用例能够覆盖复杂的边界情况和多样的分布。

为有效实现这些优化，我们采用双轨更新机制，同时作用于生成器 $\mathcal{G}^{(i)}$ 和执行命令 $\mathcal{C}^{(i)}$。对于生成器程序，我们不重写整个程序，而是采用“搜索并替换”策略，精准定位并修正根因分析中发现的特定逻辑片段。在此结构更新的基础上，大语言模型还会通过动态修改命令行参数集，同时更新执行命令 $\mathcal{C}^{(i)}$。具体而言，我们会有选择地替换效果不佳的命令，以调整生成的测试输入特征，同时添加新的执行命令，以探查此前未涉及的场景。这种协同方法可确保最终的执行命令 $\mathcal{C}^{(i+1)}$ 在结构上有效，并且日益严谨。

通过在更新后的生成器 $\mathcal{G}^{(i+1)}$下执行经过优化的命令集 $\mathcal{C}^{(i+1)}$，我们生成一批新的测试输入。随后，这些输入由参考解 S^* 处理，以生成相应的真实输出，从而形成增强且更加严格的测试用例集$\mathcal{T}^{(e,\ i+1)} = \{t_1^{(i+1)}, t_2^{(i+1)}, \ldots, t_{l_{i+1}}^{(i+1)}\}$。然后将这一新测试用例集反馈到执行与反馈收集阶段，推动下一次迭代。

上下文压缩与检查器生成。为避免冗长的迭代对话可能导致上下文溢出，我们采用了一种上下文压缩技术。该方法将多个对话轮次中的信息浓缩为单轮摘要，显著缩短输入序列长度。此外，考虑到编程问题通常有多个有效输出，我们采用协同生成策略，在生成器的同时合成一个检查器。该检查器评估解答的输出与参考输出之间的逻辑一致性，而非依赖简单的字符串匹配，以确保在存在多个有效解答的情况下仍能保证正确性。有关这些技术的更多详情，请参阅附录B。

当测试用例集 $\mathcal{T}^e$ 达到目标性能（TPR $\geq \alpha$ 和 TNR $\geq \beta$），或满足最大迭代次数限制 N_{max}时，此迭代优化循环即告终止。这种多目标退出策略确保最终测试用例具备稳健的判别能力，同时保持较高的计算效率。完整的提示模板见附录B.3。

3.3 CodeContests-O 的构建

为评估所提出框架的有效性，我们应用第3.2 节中所述的迭代优化过程，构建 CodeContests-O——一个源自 CodeContests 的高质量代码数据集（Wang 等人，2025年c）。构建过程分为以下几个阶段。

数据整理。为迭代优化过程奠定可靠基础，我们对问题集进行严格的预处理 $\mathcal{D}$（由单个问题组成 $\mathcal{P} \in \mathcal{D}$）以及每个问题对应的候选解池 S^+ 和 S^-。具体而言，我们首先按照附录A.1 中详述的一组启发式规则筛选问题集$\mathcal{D}$，确保仅保留算法复杂度适当且需求定义明确的问题。同时，我们对每个问题的解池 S^+ 和 S^- $\mathcal{P}$ 进行迭代优化，剔除无法在标准环境中编译或执行的解答。具体而言，这一筛选过程

数据集	题目	平均测试用例数	平均 S^+	平均 S^-
CodeContests	13610	95.81	332.57	649.29
CodeContests+ (1倍)	11690	25.36	373.14	734.30
CodeContests+ (3倍)	11690	61.62	373.14	734.30
CodeContests+ (5倍)	11690	97.19	373.14	734.30
CodeContests-O	11682	40.19	309.23	594.12

表1：不同方法生成的数据集之间统计属性的比较。

通过使用公开测试用例 $\mathcal{T}^p$ 实现：凡是无法在所有公开测试输入上成功运行的解答，均予以剔除。对于正确解集合 S^+，我们进一步采用更严格的纳入标准，要求每个解答都通过 $\mathcal{T}^p$ 中的所有测试用例，以保证其正确性。

数据集合成。对于每个精选问题 $\mathcal{P}$，我们使用 GPT-5 模型（3.2OpenAI，2025年）执行反馈驱动的迭代测试用例生成框架，具体细节见第 3.2 节，以合成最终的 CodeContests-O数据集。为确保执行过程的可复现性和安全性，我们采用 SandboxFusion[3] 作为运行和验证所有代码提交的标准化环境。迭代优化过程持续进行，直至达到目标阈值；最终合成时，我们设置 $\alpha = 0.95$ 和 $\beta = 0.90$。这些标准确保生成的测试用例既能验证正确解答，也能有效拦截错误解答。此外，我们将最大迭代次数限制设为 $N_{max} = 3$，以平衡测试用例质量与计算成本。

数据集属性。最终得到的数据集 CodeContests-O 包含 11682 个独特的问题。对于每个问题，数据集都提供了一套全面的测试用例，平均每个问题有40。19 个测试用例，并附有一个经过验证的解答池，其中包含 309。23 个正确解答和 594。12 个错误解答。为全面概述数据集的特征，我们在表1中对统计属性进行了比较分析，包括：**(1) 问题数**，表示问题的数量；**(2)平均测试用例数**，反映每个问题的平均测试用例数；以及 **(3)** 正确和有缺陷解答池的平均规模，**平均** S^+ 和 **平均** S^-。显而易见，CodeContests 和 CodeContests+依靠增加测试用例数量来提高识别有缺陷解答的能力，而我们的方法则侧重于提升精确度以及每个测试用例的严格性。此外，通过排除无法编译或运行失败的解答，我们确保了用于评估的解答池具有更高的纯度。CodeContests-O 更详细的统计属性见附录 A.2。

[3]https://github.com/bytedance/SandboxFusion

4 实验

4.1 实验设置

对比方法。为评估所提出框架的有效性及最终数据集的质量，我们通过将 CodeContests-O 与三种具有代表性的现有方法进行比较，开展了全面实验。这些方法包括 **CodeContests** (Li 等人，2022)，这是一个从竞赛编程平台整理而来的大规模数据集；**CodeContests+**(Wang 等人，2025c)，这是一个利用大语言模型生成的程序合成测试用例的多智能体生成器-验证器系统；以及 **HardTests** (He 等人，2025)，该方法专注于整理对抗性实例，以挑战模型鲁棒性。这种系统化比较使我们能够通过标准数据源和专门的增强技术，验证迭代优化过程。

数据质量指标。我们通过两个核心指标对合成测试用例的质量进行定量评估：真正例率（**真正例率**）和真阴性率（**真负例率**）。真正例率反映生成测试用例的保真度，通过测量已知正确解答（S^+）中能够通过测试用例而不被错误拒绝的比例，确保这些测试用例在逻辑上有效。相反，真负例率通过测量已知错误解答（S^-）中被正确拒绝的比例，量化测试用例的可区分性。为确保这些指标的鲁棒性，我们使用全部已验证解答进行评估，总计约 1.1×10^7 个样本。最大化这两个指标可确保训练信号可靠，并有效剔除幻觉解答。

训练与评估详情。为了评估

数据集	真正例率 (%)	真负例率 (%)
CodeContests	85.18	81.52
CodeContests+ (1x)	85.68	86.03
CodeContests+ (3x)	85.05	88.82
CodeContests+ (5倍)	83.84	89.35
HardTests	68.06	89.17
CodeContests-O（迭代0）	86.04	89.42
CodeContests-O（迭代1）	89.20	89.62
CodeContests-O（迭代2）	89.35	90.30
CodeContests-O（迭代3）	**89.37**	**90.89**

表2：不同数据集上的真正例率和真负例率结果。这里，迭代0表示由初始测试用例生成所产生的初始测试用例集，而迭代1-3则表示在迭代优化过程相应第 i 次迭代中合成的测试用例。我们的迭代优化带来了稳定的指标增长，在保真度和判别能力两方面均优于现有数据集。

我们数据集的实际效用，我们以 Qwen2.5-7B（Yang 等，2024）为初始策略，并使用 CodeForces-CoTs 数据集（Penedo 等，2025）进行监督微调（SFT）。随后，我们在各自的数据集上采用 GRPO 训练范式（Shao 等，2024）。我们还将 Qwen2.5-7B-Instruct（Yang 等，2024）作为参考基线，以代表当前最先进模型的性能。所有最终模型均在 LiveCodeBench（Jain 等，2024）上进行评估，并根据其三个难度级别——简单、中等和困难——使用 Pass@1 指标进行衡量。该基准提供了动态且按时间排序的竞赛编程问题流。为确保公平比较，我们在所有对比数据集上始终采用相同的训练配置。该设置可直接比较数据集的效用，展示 CodeContests-O 卓越的判别能力。具体而言，训练过程使用 $8\times$ NVIDIA A100 GPU，并基于 veRL 框架（Sheng 等，2025）开展。对于 SFT，我们将最大序列长度设为 $8k$，批量大小设为 16，学习率设为 1×10^{-3}。对于 GRPO，最大响应长度扩展至 $16k$，批量大小为 16，展开规模为$n = 4$，学习率为 1×10^{-6}。

4.2 测试用例质量验证

表2 展示了不同数据集上的真正例率和真负例率结果，其中 CodeContests-O 展现出更高的保真度和显著增强的判别能力。具体而言，原始 CodeContests 的真负例率仅为 81.52%，而我们的数据集显著提高了评测的严格程度，真负例率大幅提升至 90.89%。此外，仅增加测试用例数量（如 CodeContests+ (3x) 和 (5x)）虽可将真负例率提高至 89.35%，却无法维持较高的真正例率，这表明由数量驱动的扩充往往会引入噪声或冗余。相比之下，CodeContests-O 实现了更优的平衡，在真正例率（89.37%）和真负例率（90.89%）两项指标上均优于 Code-Contests+ (5x)。值得注意的是，HardTests 专为对抗鲁棒性而设计，但其真负例率达到 89.17% 的同时，真正例率较低，仅为 68.06%；这表明其过度专注于边界情况，可能会无意中过滤掉有效解答。这种双重指标上的优势证实，我们的反馈驱动的迭代框架所生成的测试用例不仅更多样，也更擅长精准识别细微的逻辑错误。

为了进一步刻画质量分布，我们评估了这些数据集的帕累托前沿。如图 1 所示，我们根据各个测试用例的表现对其进行排序，并按降序汇总，以展示真正例率与真负例率之间的权衡。CodeContests-O 的帕累托前沿始终包络着对比数据集的帕累托前沿，表明我们的方法能够提供更优的测试用例集，在任何给定的判别能力水平下都能实现更高的保真度。此外，帕累托前沿逐步向右上角扩展（图 1），同时真正例率和真负例率结果（表2）从迭代0到迭代3持续改善，这验证了我们由反馈驱动的迭代框架的有效性。具体而言，真正例率从 86.04% 提高到 89.37%，而真负例率则从 89.42% 提高到 90.89%。这两方面的证据证实，我们的迭代优化过程能够有效筛除低质量测试用例，同时合成更具挑战性的测试用例，从而在广泛覆盖边界情况的同时，实现卓越的判别严谨性。

4.3 基于测试用例的强化学习结果

表3 总结了使用不同数据集训练的 Qwen2.5-7B 模型的性能。一个关键观察是初始模型之间的性能差距：Qwen2.5-7B-Instruct 的准确率达到29.33%，显著优于我们的初始 SFT 模型的 25.05%。这表明，尽管在

数据集	Pass@1（%）	简单（%）	中等（%）	困难（%）
Qwen2.5-7B-Instruct	29.33	57.52	23.98	**2.70**
Qwen2.5-7B（监督微调）	25.05	58.91	13.33	0.61
CodeContests	27.10	64.66	13.55	0.77
CodeContests+（1 倍）	26.52	65.08	12.19	0.10
CodeContests+ (3x)	29.17	67.52	16.42	0.77
CodeContests+ (5x)	29.61	64.37	19.93	1.17
HardTests	26.54	62.61	14.09	0.46
CodeContests-O	**34.57**	**70.42**	**26.56**	**2.45**

表3：使用不同数据集训练的模型在 LiveCodeBench 上的性能比较。Qwen2.5- 7B (SFT) 是经过监督微调后的模型，并作为强化学习的起点。我们的 CodeContests-O 取得了更优的 Pass@1 结果，这表明强化学习期间更高的奖励保真度可直接转化为竞赛编程任务中更强的泛化能力。

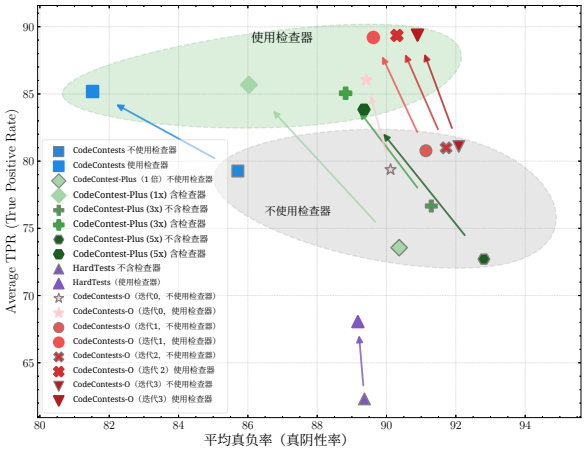

图3：基于逻辑的检查器对 TPR 和 TNR 的影响。“无检查器”表示字符串匹配评估，而“使用检查器”则采用基于逻辑的检查器。通过正确验证具有多个有效输出的解，该检查器在所有数据集上均能稳定提升 TPR。使用检查器的 CodeContests-O 实现了最佳平衡，体现了我们的同步检查器生成策略的有效性。

特定领域代码数据上进行 SFT 奠定了基础，但其问题解决能力不如通用指令模型全面。然而，使用我们的精选数据集进行强化学习后，在 CodeContests-O 上训练的模型准确率达到 34.57%，较原始 CodeContests 提升了 +7.47 %，取得显著进步；更重要的是，其表现超过了 Qwen2.5-7B-Instruct。

这一显著提升凸显了我们的高质量测试用例在增强模型推理能力方面的实际优势。相比之下，在 CodeContests+ (5x) 和 HardTests 上训练的模型准确率分别为 29.61% 和 26.54%，均不及 CodeContests-O。此外，CodeContests-O 始终优于所有对比方法，在简单、中等和困难任务上的表现均胜过其他方法。这些结果表明，对于有效的强化学习训练，测试用例质量比数量更重要，也验证了我们的框架能够生成高保真度的测试用例，并取得最先进的性能。

4.4 检查器的影响

如图3所示，启用检查器会持续提高真正例率，同时略微降低真负例率。这符合预期：在多解问题中，不使用检查器进行字符串匹配会错误地拒绝语义上正确的解答，导致假阴性。通过验证逻辑一致性，而非要求输出完全相同，检查器消除了这种歧义，并提高了验证保真度。由此产生的真正例率-真负例率权衡，使评估器在下游训练中更加均衡、可靠。

5 结论

在这项工作中，我们提出了一种反馈驱动的迭代框架，用于合成高质量测试用例。通过建立一个闭环过程，利用已知正确解答和错误解答的执行结果作为反馈信号，我们的方法能够系统地优化测试用例生成器，实现高保真度（真正例率）和强判别能力（真负例率）。我们将该框架应用于竞赛编程，构建了 CodeContests-O，其表现显著优于现有数据集；将其用作强化学习中的奖励信号时，也能带来显著的性能提升。这些结果证实，由执行反馈引导的迭代优化对于区分真正的逻辑推理与表面的模式匹配至关重要。

除了竞赛编程之外，我们的框架还可推广到任何将问题与可验证的正确解答和错误解答配对的领域，例如数学推理、形式化定理证明和约束满足任务，为构建涵盖多种领域的高保真可验证数据集提供了一种通用方案。

局限性

尽管我们的框架显著提升了测试用例质量，但其主要局限在于迭代过程所带来的计算开销。要达到最优的帕累托前沿，需要多轮模型推理和代码执行来改进生成器、检查器和命令集，因此延迟高于单次处理的方法。此外，我们的方法旨在通过生成高质量测试用例来增强现有编程问题；它不会从头生成新的问题陈述。

参考文献

Claude Team Anthropic。2025年。系统卡：Claude Sonnet 4.5。

Jacob Austin、Augustus Odena、Maxwell Nye、Maarten Bosma、Henryk Michalewski、David Dohan、Ellen Jiang、Carrie Cai、Michael Terry、Quoc Le及其他1位作者。2021年。利用大语言模型进行程序合成。*arXiv*预印本 *arXiv:2108.07732*。

Yuhan Cao、Zian Chen、Kun Quan、Ziliang Zhang、Yu Wang、Xiaoning Dong、Yeqi Feng、Guanzhong He、Jingcheng Huang、Jianhao Li及其他1位作者。2025年。大语言模型能否生成可靠的测试用例生成器？关于竞赛级编程问题的研究。*arXiv*预印本 *arXiv:2506.06821*。

Bei Chen、FengjiZhang、Anh Nguyen、DaoguangZan、Zeqi Lin、Jian-Guang Lou 和 Weizhu Chen。2022。CodeT：使用生成的测试进行代码生成。*arXiv* 预印本 *arXiv:2207.10397*。

Mark Chen。2021年。评估基于代码训练的大语言模型。*arXiv* 预印本 *arXiv:2107.03374*。

DeepMind。2025a。Gemini 2.5 Pro 模型卡。

DeepMind。2025b。Gemini 3 Pro 模型卡。

Ahmed El-Kishky、Alexander Wei、Andre Saraiva、Bo- rys Minaiev、Daniel Selsam、David Dohan、Francis Song、Hunter Lightman、Ignasi Clavera、Jakub Pa- chocki 等另外 1 位作者。2025年。使用大型推理模型进行竞技编程。*arXiv* 预印本 *arXiv:2502.06807*。

Daya Guo、Dejian Yang、Haowei Zhang、Junxiao Song、Ruoyu Zhang、Runxin Xu、Qihao Zhu、Shi- rong Ma、PeiyiWang、Xiao Bi 等另外 1 位作者。2025年。DeepSeek-R1：通过强化学习激励大语言模型的推理能力。*arXiv* 预印本 *arXiv:2501.12948*。

Zhongmou He、Yee Man Choi、Kexun Zhang、Jiabao Ji、Junting Zhou、Dejia Xu、Ivan Bercovich、Aidan Zhang 和 Lei Li。2025年。HardTests：为大语言模型编程合成高质量测试用例。*arXiv* 预印本 *arXiv:2505.24098*。

Aaron Jaech、Adam Kalai、Adam Lerer、Adam Richard- son、Ahmed El-Kishky、Aiden Low、Alec Helyar、Aleksander Madry、Alex Beutel、Alex Carney及另外1人。2024年。OpenAI o1 系统卡。*arXiv* 预印本 *arXiv:2412.16720*。

Naman Jain、King Han、Alex Gu、Wen-Ding Li、Fanjia Yan、Tianjun Zhang、Sida Wang、Armando Solar- Lezama、Koushik Sen 和 Ion Stoica。2024年。Live- codebench：对大语言模型代码能力进行全面且无污染的评估。*arXiv* 预印本 *arXiv:2403.07974*。Caroline Lemieux、Jeevana Priya Inala、Shuvendu K Lahiri 和 Siddhartha Sen。2023年。Codamosa：借助预训练大语言模型突破测试生成中的覆盖率瓶颈。载于*2023*年 *IEEE/ACM* 第*45*届软件工程国际会议（*ICSE*），第919–931页。IEEE。Rongao Li、Jie Fu、Bo-Wen Zhang、Tao Huang、Zhihong Sun、Chen Lyu、Guang Liu、Zhi Jin 和 Ge Li。2023年。TACO：算法代码生成主题数据集。*arXiv* 预印本 *arXiv:2312.14852*。Yujia Li、David Choi、Junyoung Chung、Nate Kush- man、Julian Schrittwieser、Rémi Leblond、Tom Ec- cles、James Keeling、Felix Gimeno、Agustin Dal Lago、Thomas Hubert、Peter Choy、Cyprien de Mas- son d'Autume、Igor Babuschkin、Xinyun Chen、Po- Sen Huang、Johannes Welbl、Sven Gowal、Alexey Cherepanov 等另外7位作者。2022年。使用 AlphaCode 实现竞赛级代码生成。《科学》，378(6624):1092–1097。Hunter Lightman、Vineet Kosaraju、Yuri Burda、Harri- son Edwards、Bowen Baker、Teddy Lee、Jan Leike、John Schulman、Ilya Sutskever 和 Karl Cobbe。2023年。让我们逐步验证。载于第十二届国际学习表征会议。Jiawei Liu、Chunqiu Steven Xia、Yuyao Wang 和 Ling- ming Zhang。2023年。你的代码真的是 ChatGPT 生成的吗？对大语言模型代码生成能力的严格评估。神经信息处理系统进展，36:21558–21572。Yifei Liu、Li Lyna Zhang、Yi Zhu、Bingcheng Dong、Xudong Zhou、Ning Shang、Fan Yang 和 Mao Yang。2025年。rStar-Coder：借助大规模经验证的数据集，提升代码推理竞争力。*arXiv* 预印本 *arXiv:2505.21297*。

Mike Mirzayanov、Oksana Pavlova、Pavel MAVRIN、Roman Melnikov、Andrew Plotnikov、Vladimir Par- fenov 和 Andrew Stankevich。2020年。数字化背景下，Codeforces 作为编程学习教育平台。信息学奥林匹克竞赛，14(133- 142):14。

Ansong Ni、Srini Iyer、Dragomir Radev、Veselin Stoy- anov、Wen-tau Yih、Sida Wang 和 Xi Victoria Lin。2023年。Lever：通过执行学习验证语言到代码生成。载于国际机器学习大会，第 26106–26128 页。PMLR。

OpenAI。2025年 GPT-5 系统卡。

Guilherme Penedo、Anton Lozhkov、Hynek Kydlí˘cek、Loubna Ben Allal、Edward Beeching、Agustín Pi- queres Lajarín、Quentin Gallouédec、Nathan Habib、Lewis Tunstall 和 Leandro von Werra。2025年。Codeforces 思维链。https://huggingface.co/datasets/open-r1/codeforces-cots。

Zhihong Shao、Peiyi Wang、Qihao Zhu、Runxin Xu、Junxiao Song、Xiao Bi、Haowei Zhang、Mingchuan Zhang、YK Li、Yang Wu 等 1 人。2024年。DeepSeekMath：推动开放式语言模型数学推理的极限。*arXiv* 预印本 *arXiv:2402.03300*。

Guangming Sheng、Chi Zhang、Zilingfeng Ye、Xibin Wu、Wang Zhang、Ru Zhang、Yanghua Peng、Haibin Lin 和 Chuan Wu。2025年。Hybridflow：一种灵活高效的基于人类反馈的强化学习框架。载于第二十届欧洲计算机系统会议论文集，第1279–1297页。

Peng-Yuan Wang、Tian-Shuo Liu、Chenyang Wang、Yi-Di Wang、Shu Yan、Cheng-Xing Jia、Xu-Hui Liu、Xin-Wei Chen、Jia-Cheng Xu、Ziniu Li 等共12人。2025a。大语言模型数学推理综述。*arXiv* 预印本 *arXiv:2506.08446*。

Wenhan Wang、Chenyuan Yang、Zhijie Wang、Yuheng Huang、Zhaoyang Chu、Da Song、Lingming Zhang、An Ran Chen 和 Lei Ma。2025b。Testeval：大语言模型测试用例生成基准测试。载于计算语言学协会 *NAACL 2025* 会议论文集：精选论文，第3547–3562页。

Zihan Wang、Siyao Liu、Yang Sun、Ming Ding 和 Hongyan Li。2025年c.CodeContests+：面向竞赛编程的高质量测试用例生成。载于计算语言学协会会议论文集：*EMNLP 2025*，第5576–5600页，中国苏州。计算语言学协会。

An Yang、Anfeng Li、Baosong Yang、Beichen Zhang、Binyuan Hui、Bo Zheng、Bowen Yu、Cha ng Gao、Chengen Huang、Chenxu Lv 等。2025年。Qwen3 技术报告。*arXiv* 预印本 *arXiv:2505.09388*。

An Yang、Baosong Yang、Beichen Zhang、Binyuan Hui、Bo Zheng、Bowen Yu、Chengyuan Li、Dayiheng Liu 等 1 人。2024年。Qwen2.5 技术报告。*arXiv* 预印本 *arXiv:2412.15115*。

Guanqun Yang、Mirazul Haque、Qiaochu Song、Wei Yang 和 Xueqing Liu。2022。Testaug：用于增强基于能力的自然语言处理测试的框架。*arXiv* 预印本 *arXiv:2210.08097*。

Zhiqiang Yuan、Yiling Lou、Mingwei Liu、Shiji Ding、Kaixin Wang、Yixuan Chen 和 Xin Peng。2023年。无需再手动编写测试？评估并改进 ChatGPT 的单元测试生成能力。*arXiv* 预印本 *arXiv:2305.04207*。

Shang Zhou、Zihan Zheng、Kaiyuan Liu、Zeyu Shen、Zerui Cheng、Zexing Chen、Hansen He、Jianzhu Yao、Huanzhi Mao、Qiuyang Mang 等人。2025年。AutoCode：将大语言模型用作竞赛编程出题者。*arXiv*预印本 *arXiv:2510.12803*。

A CodeContests-O 数据集详情

A.1 问题选择与筛选

为确保问题集的高质量，我们采用一系列启发式规则筛选初始问题池 $\mathcal{D}$。具体而言，如果问题 $\mathcal{P}$ 满足以下任一条件，则将其从数据集中排除：

- **描述不完整**：缺少正式文本描述或包含严重碎片化信息的问题，无法提取其底层逻辑。
- **缺少参考解**：没有任何经过验证的参考解 S^* 的问题，而参考解对于合成测试用例的初始验证至关重要。
- **多模态输入**：需要处理非文本信息（例如图像或图表）的问题。
- **非标准执行格式**：不要求完整、独立程序的问题，例如 (i)仅函数任务，其实现被限制在预定义框架中的特定函数内；或 (ii) 交互式问题，在执行过程中需要与外部评测器进行实时通信。为确保统一且自动化的环境，这些情况将被排除。

A.2 数据集分布与统计

CodeContests-O 基于原始 Code-Contests 数据集构建，主要改进集中在测试用例和解答池的质量上：

- **核心优化：** 我们的方法着重于生成高质量的测试用例 T^e 并筛选解答池（S^+ 和 S^-）。这一过程通过消除噪声和无法编译的条目，确保更高的奖励保真度和区分能力。
- **属性继承：** 每道题目的其余信息基本都继承自 CodeContests。这包括所有其他问题级元数据，例如时间/内存限制、难度评级和标签。

B 反馈驱动的迭代框架的更多细节

B.1 上下文压缩机制

为保持计算效率，并避免在迭代优化过程中超出模型的上下文窗口，我们采用了一种上下文压缩策略。在标准的多轮优化中，对话历史会逐步累积，内容如下：题目描述 $\rightarrow$ 生成器和命令的初始候选方案 $\rightarrow$ 关于执行结果的反馈 $\rightarrow$优化后的生成器和命令。

为此，我们将多轮交互压缩为一种整合表示。根据执行反馈优化生成器和命令集后，我们会丢弃中间的、次优的迭代结果以及细粒度的反馈日志。相反，我们会重新组织上下文，将原始题目描述直接与最新优化的生成器和命令配对。

通过将优化后的输出视为对初始问题的回答，我们实际上“压缩”了对话历史。这样可以确保模型仅保留最有价值的推理轨迹和最终的高保真输出，从而大幅降低词元开销，同时保留后续迭代所需的核心逻辑。

B.2 同步检查器生成与优化

为确保对具有多个有效输出的问题进行高保真评估，我们采用协同生成策略来生成检查器。模型不会采用静态字符串匹配，而是在同一过程中合成检查器和生成器。这样可以确保检查器与生成的测试用例的具体约束天然一致。在整个优化循环中，检查器会与生成器和命令集一同优化，使其验证逻辑能够适应日益复杂的边界情况。

该基于逻辑的检查器将测试输入作为上下文，用于评估解答输出与参考输出之间的一致性。通过分析逻辑正确性而非原始文本，即使存在多种正确解题路径，它也能准确识别有效解答。对测试用例和验证逻辑进行同步优化，显著增强了数据集的判别能力，确保能够对不同问题类型进行严格且公平的评估。

B.3 提示模板

此处提供反馈驱动的迭代框架中使用的完整提示模板。提示B.1 展示了初始测试用例生成阶段所用的模板，而提示B.2 则详细介绍了反馈引导的迭代优化模板。我们的生成器使用 testlib 实现[4], 。testlib 是 Codeforces 广泛采用的标准库，用于创建竞赛题目。由于检查器遵循与生成器相同的生成和迭代优化过程，为简洁起见，我们省略了检查器的提示。

由于 CodeContests+ 已经为每道题目合成了初步生成器，为降低计算开销，我们利用这些现有生成器作为起点，而不是从头开始生成。我们的提示会指示大语言模型通过查找替换机制分析并改进这些生成器。

C 大语言模型的使用

大语言模型仅用于语言润色，包括语法修正、措辞改进以及辅助可视化。大语言模型未参与研究构思、实验设计、数据分析或实质性内容创作。所有智力贡献均完全归作者所有。

[4] https://github.com/MikeMirzayanov/testlib

提示 B.1：初始测试用例生成提示

你是一位专家，擅长为编程题的边界情况生成程序生成命令行参数。

给定以下题目描述和一个 C++ 生成程序，你的任务是：1. 仔细阅读并理解题目描述。2. 仔细阅读并理解所提供的生成程序，该程序旨在为本题生成边界情况输入。3. 识别并总结输入数据的约束条件。4. 分析题目和生成程序，预测参赛者可能忽略的常见错误或边界情况。

5. 如果所提供的生成器不完整或不足以生成高质量的对抗性情况（例如，缺少模式/标志/分支，或存在有缺陷的逻辑），请使用搜索替换块提出最小且具体的生成器代码改进方案。每个代码块都必须严格遵循以下格式：

<<<<<<< SEARCH<要搜索的原始代码片段>
=======<替换片段（改进后的代码）>>>>>>>
REPLACE

注意：
- 仅提供足以确保唯一匹配的最少上下文；避免使用大段代码。

- 优先使用多个小而精确的替换，而不是一次性进行大规模替换。
- Do 不要在代码块周围添加说明；只返回代码块本身的字符串。
- 请密切注意代码的缩进、空格和换行；不要在搜索/替换片段中遗漏或更改这些内容。

- 对于每个 SEARCH 块，必须严格复制所提供的生成器中的确切内容。不要添加或修改任何字符，例如在行首添加“-”或“+”。SEARCH 块必须是生成器中的一个完全匹配的子字符串。

- 对于每个 REPLACE 代码块，严格遵循代码格式，并确保用 REPLACE 内容替换 SEARCH 内容后，生成器可以直接编译和运行。

- 在你添加的 REPLACE 代码块中，如果需要引入新的函数或变量，请确保这些函数或变量已在生成器中定义或导入。不要引入不存在的函数或变量，并仔细检查被调用函数的参数是否正确。例如，常见的 `rnd.shuffle()` 函数可能会导致错误：'class random_t'没有名为'shuffle'的成员； `rnd.next` 需要两个类型相同的参数； `ensure` 函数只接受一个参数，等等。下面是代码所用 testlib.h 中的头部注释：

```
/*
 * 强烈建议在代码中的任何其他 include 之前包含 "testlib.h"。在这种情况下，testlib 会覆盖编译器
 特定的 "random()"。 * * 如果你的代码无法编译，且编译器输出了有关 "random_shuffle"、"rand"
 或 "srand" 调用不明确的信息，这意味着你不应该使用它们。请改用 "shuffle" 和 "rnd.next"，因为
 这些调用在任何 C++ 编译器上都能产生更稳定的结果。阅读示例生成器的源代码可了解详情。 * * 请
 阅读类 "random_t" 的文档，并在生成器中使用 "rnd" 实例。以下示例调用可能对你有用： *
 rnd.next(); rnd.next(100); rnd.next(1, 2); * rnd.next(3.14); rnd.next("[a-z]{{1,100}}"). * * 此外，
 请阅读有关 wnext() 的说明，以生成偏向一侧的随机分布。 * * 请访问
 https://github.com/MikeMirzayanov/testlib/ 获取最新版本或错误跟踪器。 */
```

- 在你添加的 REPLACE 代码块中，如果需要引用代码其他部分的变量，请仔细检查其作用域，以确保所引用的变量在生成器中可见。

6. 根据你的分析和改进后的生成器，设计并输出一组多样化的命令行命令（"command_list"），这些命令在执行时将使用生成程序生成边界情况输入，尽可能涵盖各种特殊和对抗性情况。请注意，命令行的格式和参数必须符合生成程序的要求。例如，确保 --seed 可以是无效参数；当 --n 通常需要数值时，不要传入字符串。

题目描述：{problem_statement} 生成程序（C++）：{generator} **严格遵循以下输出要求：** - 你的回复必须采用符合以下结构的 JSON 格式：{{ "input_constraints_summary": "描述题目描述中输入约束的字符串", "search_replace_generator_blocks": ["<<<<<<< SEARCH\n<original>\n=======\n<replacement>\n>>>>>>> REPLACE", ...],"command_list": ["./gen --arg1 value1 ...", "./gen --arg2value2 ...", ...]}}

- “input_constraints_summary”字段应清晰、简洁地概述所有输入约束，包括题目描述中明确提及的约束（例如输入规模限制、取值范围、格式要求等），以及可从题目描述中推断出的任何隐含约束（例如问题背景所暗示的属性、不变量或隐藏要求）。

- `search_replace_generator_blocks`为可选项——仅在生成器需要改进时提供。每个条目都必须严格遵循上文所示的搜索替换块格式。如果无需更改，请返回空列表（[]）。如果提出了更改，请确保 `command_list` is generated against the updated 生成器 (i.e., 在一个 pplying 编辑之后).
- “command_list”字段必须包含一个 Shell 命令列表，每条命令都以“./gen”开头，后跟生成程序所需的适当参数。每条命令都应设计为生成一个边界情况输入。这些命令生成的所有边界情况输入都应尽可能多样且具有对抗性，涵盖广泛的边界情况和对抗性场景。

- 不要直接生成边界情况输入；只能生成用于运行生成程序的命令行。

- 命令应可直接在 Linux shell 中执行，并应按照生成程序的要求使用正确的参数格式。

提示 B.2：反馈引导的优化提示

现在，你需要根据评估反馈，优化先前为边界情况生成程序生成的命令列表。

你之前已为给定的编程题生成了一组命令。流程如下：

1. 已将生成的 `search_replace_generator_blocks` 应用于生成器。SEARCH 片段未能完全匹配的代码块已被跳过。 2. 执行每条命令以生成一个或多个边界情况输入。 3. 对于每个生成的边界情况输入，运行标准解以获得相应的输出，从而形成完整的边界情况（输入 + 输出）。 4. 随后使用这些边界情况评估正确解答和错误解答。

当前改进后的生成程序（C++）（注意：先前返回的 `search_replace_generator_blocks` 所做的编辑已应用于下方的生成器。未体现在此处的代码块，是因为其 SEARCH 片段未能完全匹配而被跳过。如果先前的 `search_replace_generator_blocks` 为空，或没有任何代码块被应用，那么此处显示的生成器与最初提供的生成器相同，并会显示为空字符串）：

{improved_生成器}

当前命令列表：{current_command_list}

对于每条命令，以下是相应生成的边界情况输入（对于某些会生成超长输入的命令，该命令的输入将替换为 `[输入]`）：{command_to_input_map}

如果任何命令执行失败，或在生成输入时产生错误，以下是错误消息（如有）：

{command_运行_错误}（理想情况下，此处应为空）

评估结果如下（理想情况下，以下三项都应为空）：
- 正确解答的输出：这些情况是指生成的边界情况错误地导致正确解答失败（即在这些情况下，正确解答被判定为错误）。{ correct_results}

- 错误解答的输出：这些情况是指生成的边界情况未能揭示错误解答中的错误（即在这些情况下，错误解答被判定为正确）。{incorrect_results}

- 标准解的输出（仅包含使用标准解运行时失败的情况的结果）：这些情况是指标准解本身在生成的边界情况下失败或产生错误。{outputs}

请注意：
- 对于某些会生成超长输入的命令，`stdin` 字段在 `正确_结果`/`错误_结果`/`输出` 中可能会被相应的命令字符串替代，并在字段末尾附加 ` [命令]` 标签，以表明进行了这种替换。看到这样的 `stdin`值时，在进行任何推理、比较或决策时，都应使用提供的命令与生成输入之间的映射，将命令隐式转换为实际的 `stdin` 内容。- 对于某些输出（ `stdout`/` 预期 _ 输出 `）非常长的情况， `stdout`/` 预期 _ 输出 ` 字段可能会被 `[ 输出 ]`/`[预期输出]`替代。看到 `[输出]`/`[预期输出]`时，如果解答的 `通过` 字段为 False，则在推理时应仅依赖给定的解答内容。以下是题目描述中提到的输入约束的清晰简要总结（例如，输入规模限制、取值范围、格式要求等）：{input_约束_总结} 你的任务是：1. 根据上述标准解结果，找出会生成无效或无用边界情况（即使用标准解运行时会失败）的命令，并标记为替换。2. 根据正确解答结果，找出会生成导致正确解答被错误判定为错误的边界情况的命令，并标记为替换。3. 分析上述结果，判断哪些命令无法有效区分正确解答和错误解答。4. 如果提供的生成器不完整/不足以生成高质量的对抗性情况（例如，缺少模式/标志/分支或存在错误逻辑），请使用搜索替换块提出精简、具体的生成器代码改进。5. 生成新的附加命令，以便更有效地暴露错误解答中的错误，并提高正确解答与错误解答之间的区分度。**严格遵守以下输出要求：** - 你的回复必须是符合以下结构的 JSON：{{ "搜索_替换_生成器_代码块": ["<<<<<<< SEARCH\n<原始内容>\n=======\n<替换内容>\n>>>>>>> REPLACE", ...],"替换_命令_列表": ["旧_命令_1", "旧_命令_2",...],"新增_命令_列表": ["新_命令_1", "新_命令_2", ...]}} - `搜索_替换_生成器_代码块`为可选项；仅当生成器需要改进时才包含。每个项目都必须严格遵循上述搜索替换块格式。如果不需要更改，请返回空列表（[]）。如果提出更改，请确保 `替换_命令_列表` 和 `新增_命令_列表` 均基于更新后的生成器生成（即应用编辑之后）。- `替换_命令_列表` 包含原列表中应删除/替换的命令，这些命令会生成无效或无用的边界情况，或导致正确解答被错误判定为错误。- `新增_命令_列表` 包含应添加的新命令，以便更好地区分正确解答和错误解答，包括被替换命令的改进版本以及全新的对抗性命令。- 每条命令都应是以“./gen”开头、后跟生成程序所需参数的 shell 命令。- 不要直接生成边界情况输入；只需生成用于运行生成程序的命令行。

- 这些命令应可直接在 Linux shell 中执行，并应按照生成程序的要求使用正确的参数格式。

请根据上述反馈，重点最大化所生成边界情况的对抗价值。