DIVALGO: A SYSTEMATIC EVALUATION OF ALGORITHMIC DIVERSITY FOR CODE GENERATION MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Code generation models are actively being used in professional and educational settings. However, the competence of large language models (LLMs) to generate algorithmically diverse code remains largely unknown. We propose the `DivAlgo` benchmark, a large-scale dataset constituting algorithmically diverse solutions to programming problems. We also introduce a lightweight similarity metric to conduct an evaluation at scale. Our experiments spanning 24 modern code generation models found that uncontrolled model-generated code is significantly less diverse than the human solutions (captures $\sim$31% of human diversity), and even the best diverse code generation strategy could only achieve $\sim$70% of the human diversity. Moreover, while it was possible to steer the models to generate solutions based on human strategies, they were unable to reproduce the diversity of the human solutions. These findings indicate the limitations of code generation models to capture the algorithmic diversity covered by humans, and overexposure to model-generated code can narrow the algorithmic vocabulary of programmers.

1 INTRODUCTION

Using coding agents and Large Language Models (LLMs) to perform programming tasks has become ubiquitous not only in software development jobs but also in computer science education and research. The swift adoption of these systems has outpaced scrutiny of the generated outputs, with users overrelying on the model generated code (Shah et al., 2025; Khati et al., 2025). Much effort on improving LLMs for code generation has focused on executable correctness, using post-training techniques such reinforcement learning with verifiable rewards (RLVR) that optimize for solution correctness (Lambert et al., 2024; Shao et al., 2024; Wang et al., 2026; Li et al., 2026; Zeng et al., 2025). However, reliance only on functional correctness as the sole objective in these learning strategies often concentrates generations around repeated implementations, as examined by Le Bronnec et al. (2026) using a plagiarism detection system for code diversity (Prechelt et al., 2002). If a model is unable to generate the breadth of approaches that humans are capable of developing, then its widespread use narrows the 'algorithmic vocabulary' that is encountered and learned from. Moreover, solution homogeneity has direct engineering costs as diverse implementations are the foundation of N-version fault tolerance (Almeida et al., 2026).

Thus, investigating the algorithmic diversity of model generated solutions has become an important area of research with contribution in code similarity *metrics*, such as, lexical or embedding overlaps (Ren et al., 2020; Feng et al., 2020; Liu et al., 2024b), execution behavior similarity (Zhang & Lu, 2026), code diversity; *benchmarks* (Zhang & Lu, 2026; Narula et al., 2026) and *approaches* to increase the diversity of model generated algorithmic solutions (Lee et al., 2025; Le Bronnec et al., 2026). Building on the prior work, we ask three research questions: **RQ1:** How diverse are model-generated solutions compared to human-written solutions? **RQ2:** How much algorithmic overlap exists between solutions from different models for the same task? **RQ3:** Can models generate solutions that follow the same strategies adopted by humans? These questions enclose three different notions of algorithmic diversity: within a model, across models and relative to specific human strategies.

We make three main contributions. **First**, our new benchmark, `DivAlgo`, allows us to investigate algorithmic overlap at three levels of granularity: data structures, techniques, and programming paradigms. It contains two algorithmically distinct human solutions, along with a pool of human-

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

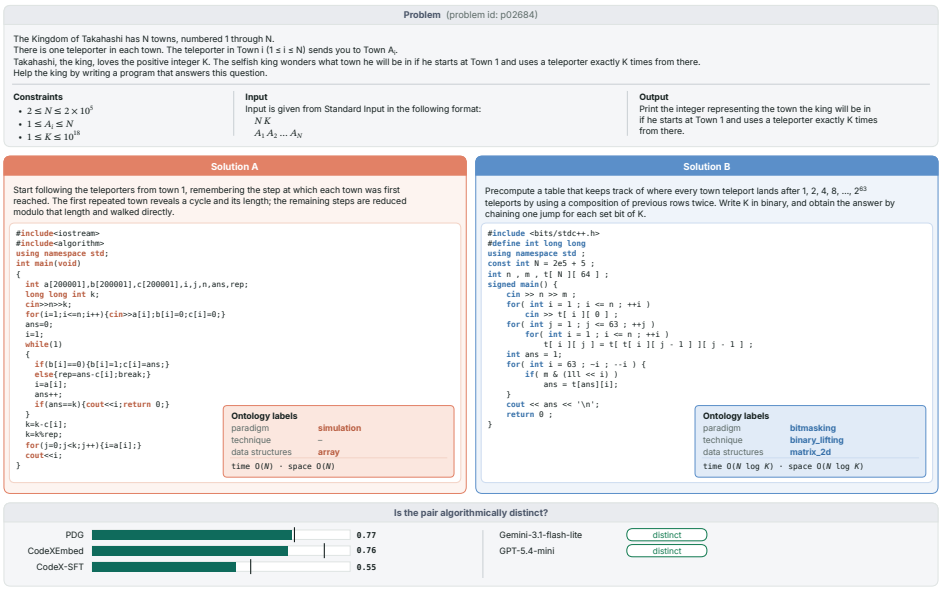


Figure 1: Illustrative example from our algorithmic diversity benchmark's test split demonstrating two algorithmically distinct solutions for the same problem. Solution A solves the problem using the *simulation* paradigm with an *array* data structure, whereas Solution B adopts a *bitmasking* paradigm with *binary lifting* technique using a *2D matrix* data structure. Both solutions have different time and space complexity. The bar graphs show the code similarity of the two solutions and the labels show the LLM-judgments. The ticks in the bar graph indicate the similarity thresholds, below which the solution pair is considered algorithmically diverse in respective metric space.

written solutions for population-level analysis (example in Figure 1). We systematically constructed the benchmark from `CodeContest+` (Wang et al., 2025) using cascaded filtration over distinct code similarity metrics and LLM-judge ensembles to obtain a high-quality and large-scale dataset (Section 3). **Second**, to answer the three research questions at scale, we introduce a distilled and validated lightweight similarity metric and use the Vendi score (Friedman & Dieng, 2022) with this metric to estimate the number of distinct algorithms in a population of solutions (Section 4.2). **Third**, we thoroughly evaluate the capabilities of a large set of LLMs to generate diverse solutions, answering our three research questions. For RQ1, we show that uncontrolled sampling (for 24 LLMs) yields considerably less diverse solutions than human solutions, with the best model achieving 31% diversity compared to human solutions. Even when using diverse code generation strategies such as temperature variations, verbalized sampling and iterative constrained generation , we can only obtain up to 70.5% diversity as compared to human solutions on matched number of samples (Section 5.1). For RQ2, pooling unconditioned solutions from all 24 models is less effective than a single model with appropriate prompting strategy; as we found the models aligned systematically with the more popular human solution approach (Section 5.2). For RQ3, even when the models were steered to generate solutions corresponding to both representative human solutions, they were unable to reproduce the diversity of the overall human solutions (Section 5.3).

2 Related Work

Code generation has been one of the most popular testbeds to measure reasoning capabilities of large language models (Chen et al., 2021). The verifiable nature of program evaluation makes them ideal for post-training paradigms such as group-relative policy optimization (Shao et al., 2024) and RLVR (Lambert et al., 2024), with prior work developing models capable of program synthesis (Wang et al., 2026; Li et al., 2026) and automatic test case generation (Zeng et al., 2025). There exist several benchmarks to evaluate the functional correctness of model-generated code, such as `HumanEval` (Chen et al., 2021), `CodeContests` (Li et al., 2022), and `CodeContests+` (Wang

et al., 2025). We refer the readers to the extensive surveys for a comprehensive review (Jiang et al., 2026; Dehaerne et al., 2022; Chen et al., 2024).

While evaluating functional correctness of model generated code remains the most popular focus of prior work, there are some efforts in investigating the diversity of model generated solutions. Lu et al. (2025) proposed "*denial prompting*", a systematic constrained generation approach that iteratively limits the techniques models can use to solve the programming problem, eliciting diversity beyond the vanilla model generated outputs. They evaluate diversity using presence of atomic labels (such as *while loop*, *break statement*) obtained using LLM-based classifiers to establish convergent and divergent creativity in the generated solutions against previously generated solutions and human solutions respectively. While they measure LLM's capabilities in generating diverse solutions, they limit the study to very simple problems (CodeForces 800 level problems) with partially syntax-dependent constraints. Lee et al. (2025) adopted a clustering-based approach to identify the number of algorithmically unique solutions using binary classification obtained from `Llama-3.1-8B` (Grattafiori et al., 2024). They found that increasing the temperature and sampling using heterogeneous models can help improve the diversity of model generated outputs. Complementing them, Narula et al. (2026) kept the sampling fixed, and investigated the impact of varying incidental prompt cues carrying no algorithmic content (such as persona, team names, project codenames, color themes). They found that these cues can substantially shift the algorithm-family distributions, and the most reliable mitigation strategy was direct algorithm naming. Their approach for detecting algorithm families relied on manually specified abstract syntax tree (McCarthy, 1964; Knuth, 1968) patterns for each task to capture the structural signals reliably, limiting the investigation to 11 tasks. Zhang & Lu (2026) instead investigated algorithmic similarity from execution behaviors, executing the programs and capturing the alignment of the intermediate states. While precise, this approach cannot be extended to problems without tractable intermediate structure (refer to Figure 1 for an example). Beyond programming problems, Almeida et al. (2026) connects this research direction with the software-diversity literature, extending classical models of correlated failures Avizienis et al. (2004); Lyons & Vanderkulk (1962) to LLM-generated code. They find that heterogeneity across models, temperatures, and programming languages yields measurable reliability gains, but failure independence is not guaranteed with this approach.

Prior work examined the problem of algorithmic diversity for very simple problems (Lu et al., 2025), or at a very small scale (Narula et al., 2026), for a particular class of problems (Zhang & Lu, 2026), or at a very coarse level (Lee et al., 2025; Le Bronnec et al., 2026). Our work extends this literature by developing a large-scale algorithmic diversity benchmark spanning problems of varying difficulty, with fine-grained control over the algorithmic overlap between solutions. We distilled a lightweight algorithmic similarity metric to comprehensively examine the diversity of model generated code against human solutions across several diverse code generation strategies.

3 DivAlgo Benchmark Construction

We start with the CodeContests+ benchmark, that comprises of $11,690$ programming problems from CodeForces, LeetCode and AtCoder, with 4.36M correct submissions. To prevent any confounds from inter-language similarity measurements (Utpala et al., 2024), we limit our investigation to the 1.73M C++ solutions in the dataset[1]. We filtered $1,326$ instances that had less than 20 C++ correct submissions, and dropped 205 instances that did not have a problem statement. We additionally dropped 966 duplicate instances[2]. This filtration left us with a canonical set of $9,193$ instances, with mean 176.4 C++ correct submissions.

To build `DivAlgo` we need to identify algorithmically different solutions among the human provided solutions. A recent human-annotation study (Lee et al., 2025) has shown the difficulty of obtaining high quality human annotations for algorithmic diversity, particularly at scale. Thus, we conduct a systematic investigation of automatic metrics and a two-stage diversity selection process. We focus on similarity metrics that go beyond superficial syntactic attributes (such as variable

[1]We decide to use C++ because it is the most popular language in the dataset, spanning 39.7% solutions, compared to 33.6% for Python-3, 20.6% for Java, and 6.1% Python-2.

[2]CodeForces platform has two divisions, Div1 and Div2, and duplicate problems are used across divisions with changed difficulty. To only consider unique problems, we dropped duplicate problems by matching normalized prefix hashes over the first 1,000 characters of the problem statement.

Table 1: Dataset statistics of the algorithmic diversity benchmark.

			Human solutions		**Ontology-based algorithmic diversity**			
Split	*instances*	*C++ sol.*	*total*	*median*	*strong*	*medium*	*weak*	*identical*
train	5,623	11,246	148,242	25	2,429	1,463	1,229	502
val	1,000	2,000	25,915	24	419	257	231	93
test	1,000	2,000	26,145	25	428	280	212	80
all	7,623	15,246	200,302	25	3,276	2,000	1,672	675

names): 1) program dependency graph (`PDG`) that captures the control and data dependencies for a program, 2) a code embedding model, `CodeXEmbed-2b` Liu et al. (2024b), which was the best model on the code retrieval (CoIR) leaderboard Li et al. (2025) in July 2026, and 3) six proprietary LLMs (`Gemini-3.1-pro`, `Gemini-3.1-flash-lite`, `GPT-5.5`, `GPT-5.4-mini`, `Claude-opus-4.8` and `Claude-Haiku-4.5`) used as LLM-as-judge. The models were prompted to give the similarity classification out of four choices: *identical*, *variation*, *distinct* and *very distinct* (prompt in Appendix E).

An analysis on 100 randomly selected solution pairs (Appendix A.1) reveals that LLM-judges have high pairwise AUC ($\sim$0.85) despite modest rank correlation indicating their disagreement arises from the decision boundaries of algorithmic similarity. Additionally both lightweight measures (`CodeXEmbed` and `PDG`) align with this aggregate well above chance (0.709 and 0.657), confirming their capability in coarsely capturing algorithmic differences in their respective space. Therefore, we adopt a two-stage approach to obtain the algorithmically diverse solutions. As the dataset contains 1.6M correct C++ submissions after the primary filtration, we cannot rely on LLM-based pairwise measurements to obtain the algorithmically diverse solutions due to cost. Thus, we first use the two lighweight similarity measures `CodeXEmbed` and `PDG` to filter similar solutions, and then select the final diverse solution pairs using the best LLM-judges.

For the first stage, we filter solutions from human submissions (median 185 solutions) to only retain solutions with higher algorithmic diversity, as captured by the lightweight metrics `CodeXEmbed` and `PDG`. In each metric space, we kept instances using a medoid-seeded farthest-first traversal strategy to capture distant solutions. We set the number of solutions filtered per strategy from the covering-radius curve, stopping when an additional solution pick reduced this normalized radius by less than 2 percentage points. We ended up sampling 10 solutions from `PDG` space, 6 solutions from the `CodeXEmbed` space, and also included 5 uniform random solutions for control to ensure appropriate representatives per solution. As the samples were done independently, this filtration stage yielded a median of 19 solutions per instance.

For the subsequent LLM-based filtering step, we shortlist the disjoint top-3 pairs of solutions, ranked in ascending order of similarity computed as the mean `CodeXEmbed` and `PDG` similarity scores. We used the cheaper `Gemini-3.1-flash-lite` and `gpt-5.4-mini` to establish algorithmic dissimilarity between the pairs, using `Claude-Haiku-4.5` to break the ties.

Ontology-driven Program Classification for Interpretable Labels. To collect labels for the program description, we propose a three-layer ontology to describe a program's algorithmic paradigm, the techniques used in the problem solving, and the data structures used. The ontology spans 13 paradigms, 64 techniques and 22 data structures. We used `Gemini-3.1-flash-lite` for this ontology labeling, as it had the highest agreement with the other six proprietary models we considered (see Appendix A.2). We provide the distribution of the ontology labels in Figure 8.

Final dataset statistics. We split our dataset into training, validation and test split with stratified sampling based on the programming paradigms and the average length of candidate C++ solutions (see Table 1). We additionally classify each solution pair's algorithmic diversity into four categories depending on their ontology label overlap - *strong*: different programming paradigms, *medium*: same programming paradigms but different techniques, *weak*: same programming paradigm and technique, but different data structure, and *identical*: same ontology for both the solutions. We provide the examples for each classification type in the Figures 9, 10, 11 and 12.

4 EXPERIMENTAL SETTINGS

Section 4.1 defines the problem, covering two types of algorithmic diversity, population-level and their algorithm coverage over human solutions. Section 4.2 describes the evaluation metrics and Section 4.3 presents the models and generation strategies used to answers our 3 research questions.

4.1 PROBLEM DEFINITION

Given a problem P, let $S_P^h = \{s_1^h, s_2^h, \ldots, s_N^h\}$ be a population of N correct human-written solutions to P. We assume an algorithmic diversity function $D : 2^{\mathcal{S}} \to \mathbb{R}$ that maps a population of programs to a scalar measuring the distinct underlying algorithms in $\mathcal{S}$. Then, a code generation model $\mathbf{M}$ that generates N solutions $S_P^{\mathbf{M}} = \{s_1^{\mathbf{M}}, \ldots, s_N^{\mathbf{M}}\}$, is considered as algorithmically diverse as humans, if $D(S_P^{\mathbf{M}}) \geq D(S_P^h)$ provided $D(S_P^h) \geq 1$ (*i.e.,* more than one solution exists for this problem). Additionally, assuming a labeling function $\mathcal{A} : \mathcal{S} \to 2^{\mathcal{L}}$ that maps each program to a set of algorithmic approaches over a label space $\mathcal{L}$; a model $\mathbf{M}$ captures human algorithmic diversity on P exactly when $\mathcal{A}(S_P^{\mathbf{M}}) = \mathcal{A}(S_P^h)$, a strictly stronger condition than $D(S_P^{\mathbf{M}}) \geq D(S_P^h)$.

4.2 EVALUATION METRICS

Vendi$_{\texttt{CodeX-SFT}}$: algorithmic diversity of a set of solutions. To measure the number of algorithmically diverse solutions in a set of solutions, we distilled the `CodeXEmbed` embedding model Liu et al. (2024b) on solution pairs that were labeled with the three proprietary LLM ensemble in the second stage filtering (Section 3). The training data comprised of 14,195 solution pairs from the training and validation instances, and achieved improved alignment with six LLM-ensemble (0.727 AUC compared to 0.686 AUC on the base model). We also validate the quality of the metric on a small human-curated algorithmic diversity dataset from Zhang & Lu (2026). We show that even for out-of-distribution python programs, the trained model outperforms the base model in distinguishing behaviorally distinct solutions (see Appendix B). Being a small model with moderately high alignment with the frontier models, we used this similarity measure as our primary algorithmic similarity measurement and refer to the trained model as `CodeX-SFT`. We adopt Vendi score Friedman & Dieng (2022) to evaluate the diversity of a population of solutions. For each instance with a set of n solutions, we embed them with `CodeX-SFT`, and L2-normalize them to obtain the cosine Gram matrix. The Vendi score is then obtained over this matrix, with outputs ranging between 1 (all solutions are identical) and n (all solutions are mutually dissimilar) as follows:

$$\text{Vendi}_{\texttt{CodeX-SFT}} \;=\; \exp\left(-\sum_i \lambda_i \log \lambda_i\right) \tag{1}$$

where $\lambda_1 \ldots \lambda_n$ are the eigenvalues of the cosine Gram matrix.

AlgoCov: algorithmic coverage of representative human solutions. To evaluate whether a set of solutions $S = \{s_1, s_2, \ldots, s_N\}$ is able to express the two representative human solutions (referred to as A and B), we propose the metric Coverage. We first obtain the ontology-labels for the solutions using the `Gemini-flash-lite-3.1` model $\{D_{s_1}, \ldots, D_{s_N}\}$ and $\{D_A, D_B\}$. We compute the solution set's algorithmic coverage over the two solutions as follows:

$$\begin{aligned} \text{Match}_X(s_i) &= \mathbb{1}[D_{s_i} \cap D_X \neq \emptyset], \quad X \in \{A, B\} \\ \text{AlgoCov}(S) &= \max_{s_i \in S} \text{Match}_A(s_i) \cdot \max_{s_j \in S} \text{Match}_B(s_j) \end{aligned} \tag{2}$$

4.3 CODE GENERATION MODELS AND GENERATION STRATEGIES

To evaluate uncontrolled diversity of LLM-generated solutions, we tested 24 code generation models: 2 closed-source proprietary models `Gemini-3.1-Flash-Lite` and `Gemini-3.8-Flash` and 22 open-source models `Gemma-4` (`E4B`, `12B`, `26B`, `31B`) (Team et al., 2026), `codegemma-7b-it` (Team et al., 2024), `Phi-4` (Abdin et al., 2024), `CodeQwen-1.5-7B` (Bai et al., 2023), `Qwen2.5` (`0.5B`, `7B`, `Coder-7B`) (Qwen et al., 2025), `Qwen3-Coder` (`30B` and `32B`) (Yang et al., 2025), `Deepseek-V2-Lite` (Liu et al., 2024a), `Codestral-22B`, `Starcoder2-15B` (Lozhkov

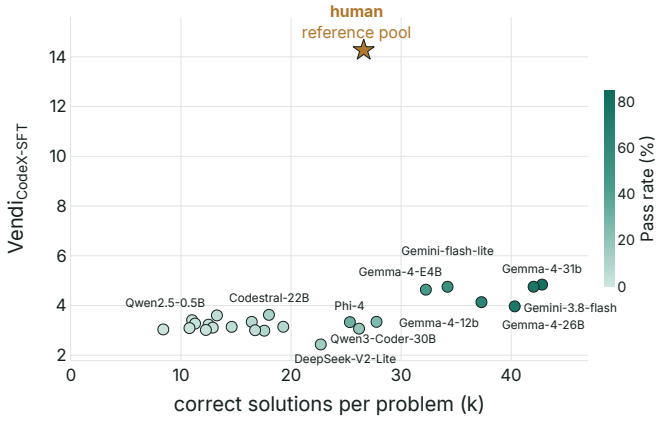


Figure 2: Algorithmic diversity ($\text{Vendi}_{\texttt{CodeX-SFT}}$) of 50 uncontrolled LLM-generated solutions against human-written solutions for the test set problems (detailed results in Table 4 Appendix D).

et al., 2024), `Mistral-7B-v0.3` (Jiang et al., 2023), `Mistral-Nemo`, `Llama` (`3.2-3B`, `3.1-8B`) (Grattafiori et al., 2024) and `CodeLlama` (`7B`, `13B`, `34B`) (Roziere et al., 2023). We then select three open-weight models (`Phi-14B`, `Gemma-4-E4B` and `Gemma-4-31B`) for our diverse code generation experiments, based on high pass rates with diverse models on the uncontrolled generations (31.4%, 44.7% and 76.8% respectively; Table 4). The diverse code generation strategies include temperature variations on the uncontrolled generation prompt, *verbalized* sampling (Zhang et al., 2025) and *iterative* constrained generation (Ye et al., 2025), where the model was prompted to generate a valid solution different from what it had already generated. These uncontrolled and controlled generation experiments, evaluated with $\text{Vendi}_{\texttt{CodeX-SFT}}$, help us answer our first two research questions, *how diverse are single model generated solutions* (**RQ1**; Section 5.1), and *how algorithmically distinct solutions from different model generated solutions are* (**RQ2**; Section 5.2). We also conduct steering experiments to examine whether *models can generate solutions following the human strategies* (**RQ3**; Section 5.3). For this experiment, we provide the ontology labels from the reference solutions A and B (called "steer (A+B)"), and evaluated coverage using AlgoCov to estimate the oracle performance. We provide the hyperparameters in Appendix C and the prompts in Appendix E.

5 Results and Discussion

5.1 How diverse are LLM-generated solution?

To answer **RQ1**, we generate 50 unconditioned solutions by zero-shot prompting the LLMs over the 1,000 test set instances. Figure 2 illustrates the algorithmic diversity of 24 LLMs and the human solution diversity using $\text{Vendi}_{\texttt{CodeX-SFT}}$ metric (Eq. 1). We found that unconditioned generations do not yield very diverse solutions, even after sampling almost twice as many solutions as the available human reference pool. Compared to the Vendi score of 14.27 for human solutions, the most diverse LLM (`Gemini-flash-3.8`) achieved 4.84 Vendi score.

To further investigate the extent of LLM-generated diverse solutions, we selected three representative open-weight models `Gemma-4-31b`, `Gemma-4-E4b` and `Phi-4-14b` to inspect the impact of temperature variation, *verbalized* sampling and *iterative* generation (Section 4.3). Figure 3 (left) presents the results from this experiment. We found that increasing temperature increased the diversity of the correctly generated solutions, with $\sim +1$ increase from T=0.2 to T=0.8, and $+0.5$ increase from T=0.8 to T=1.2. However, increasing the temperature affected the models differently, with weaker models yielding less correct solutions compared to stronger models with higher pass rates. In comparison, the other two diverse output generation strategies were significantly more effective, with verbalized sampling increasing the diversity from $+0.38$ to $+4.13$ and iterative generation from $+0.98$ to $+6.22$. While these strategies outperform the zero-shot baseline, their higher Vendi score was partially attributable to the higher samples compared to the human reference. For

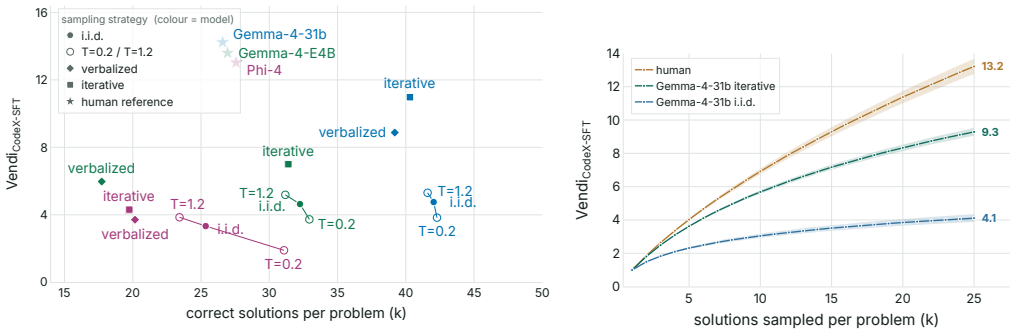


Figure 3: Algorithmic diversity ($\text{Vendi}_{\texttt{CodeX-SFT}}$) of three diverse text generation strategies for three LLMs. (Left) temperature variation, *verbalized* sampling and *iterative* constrained generation on the test set problems and the corresponding scores for human solutions for matched problems. (Right) increase in algorithmic diversity of human solutions and the best performing LLM, `Gemma-4-31b` for 412 problems where at least 25 correct solutions are available for both human and LLM solutions. T=0.8 is denoted as *i.i.d.*, independent and identically distributed samples (same as the results in Figure 2). We report the human solutions corresponding to each model's success rate for a fair comparison within model generations. For the detailed results, refer to Table 5 in Appendix D.

a fair comparison, we examined the best performing model, `Gemma-4-31b` with iterative generation, against the human-written solutions. We selected the subset of problems (N=412) with at least 25 solutions for both populations. We found that while iterative generation did significantly improve the solution diversity compared to the zero-shot (*i.i.d.*) baseline, it also steadily remained below the human solutions, achieving 70.5% of the Vendi score for the sample size of 25 solutions on this subset (Figure 3 right).

Discussion. These results indicate that algorithmic diversity is not a property that emerges from sampling alone, irrespective of the number of *i.i.d.* samples generated. The most diverse model barely reaches a third of the human Vendi score. Simple strategies like temperature increase yields diminishing returns while degrading the correctness for weaker models. Prompting strategies that elicit variation are considerably more effective, suggesting that the diversity deficit stems from how the solutions are sampled rather than the model choice. The strongest single-model approach still captures only 70% of the human diversity at matched sample size, expressing the limitations in LLMs to elicit human-level algorithmic variety.

5.2 Do Models Converge on the Same Algorithms?

To answer **RQ2**, we investigate the algorithmic preference of LLM-generated solutions juxtaposed against the popularity of the approach within the human reference solutions. We measured algorithmic similarity of the solution population against the two representative solutions using the trained `CodeX-SFT` with a 0.23 cosine distance threshold[3]. Our analysis (Figure 4 left) indicated a clear affinity of LLM-generated solutions to align with the more popular representative solution (mean Pearson's $r = +0.695$).

Additionally, we computed the inter-model solution diversity by sampling solutions from different unions of LLM-generated solutions (Figure 4 right). Since Vendi score depends on the number of samples, we evaluated the diversity for a subset of solutions for all compared solutions ($k^* = min(k_{human}, k_{LLM})$). We found that sampling from heterogeneous model generations does increase the solution diversity ($\text{Vendi}(\rho_{all}^{i.i.d.}) = 6.69$) compared to the most diverse individual LLM ($\text{Vendi}(\texttt{Gemini-flash-3.8}) = 4.84$), it still sits far below the diverse code generation strategies

[3]The threshold was obtained by calibrating the algorithmic similarity within the metric space of `CodeX-SFT` using the two representative human solutions (A and B) from 150 randomly selected instances from the dataset. The cutoff of 10th-percentile was chosen to separate the *matched* and *novel* solution pairs on the known algorithmically distinct pairs. This conservative threshold helped establish the boundary of algorithmically similar solutions for the experiment.

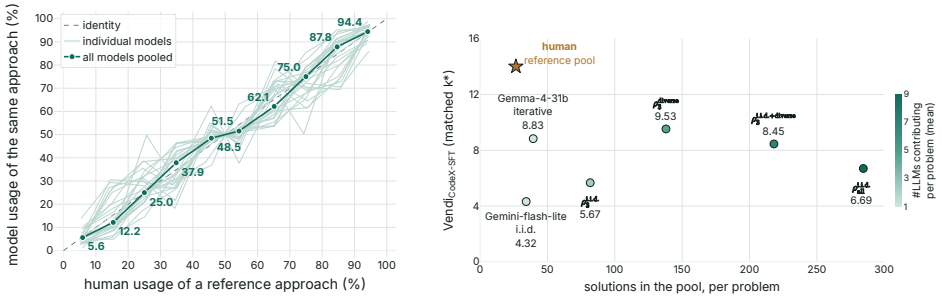


Figure 4: (left) LLM-generated solution's preference of using one of the two reference approaches vs the popularity of the same approach in human-written solutions. Alignments for model and human preferences were obtained using the cosine similarities obtained from the `CodeX-SFT` embedding model. (right) Algorithmic diversity of ensemble of LLM-generated programs ρ at the matched k^* to the human reference solutions along with the most diverse LLMs for *i.i.d.* setting and the *diverse* code generation settings. ρ_{all} denotes all 24 LLMs used in this work, ρ_3 denotes the three open-source models used for diverse code generation experiments (Figure 3 left).

($\text{Vendi}(\texttt{Gemma-4-31b}_{iterative}) = 8.83$). Similarly, sampling from ensemble of the three models prompted to generated diverse solutions also increased the diversity ($\text{Vendi}(\rho_3^{diverse}) = 9.53$); however, the gain was significantly less for the computational cost incurred to obtain the additional solutions compared to a single model.

Discussion. Our results corroborate the findings from Lee et al. (2025), who show that merging solutions from heterogeneous models increases algorithmic diversity. While we found that a union of multiple models does increase the diversity over a single model (from 4.32 to 6.69), it is not a substitute for diverse code generation. Combining multiple diversely generated outputs does increase the diversity further (from 8.83 to 9.53), the additional +0.70 gain over the best single-model approach is marginal compared to the threefold increase in the generation costs. This finding, along with the strong alignment between model preferences and the more popular human approach ($r = +0.695$) suggest that heterogeneous models share an algorithmic prior that cannot be overcome by ensembling the model outputs.

5.3 Can LLM-generated solutions follow human solution strategies?

To answer **RQ3**, we use our algorithmic diversity benchmark to investigate whether LLMs are capable of capturing the two representative human solutions using AlgoCov (Eq. 2). To specifically measure the coverage of algorithms expressed in the reference solutions (depicted as A and B), we limit this investigation to a subset of 484 problems in the test set with mutually exclusive solutions in *strong* (274), *medium* (104) and *weak* (106) overlap granularities. In addition to the diverse generations strategies discussed earlier, we include the "steer (A+B)" approach that explicitly provides the ontology labels from the reference solutions to steer the model to generate the corresponding solutions (prompt in Figure 18). This oracle-style baseline serves as an upper bound on how well LLMs can reproduce the reference solutions when given explicit guidance. To contextualize model performance using AlgoCov, we additionally measure the coverage of non-reference human solutions for a grounded-comparison.

Table 2 shows the results from this study for a subset of 10 solutions per strategy. From the algorithmic coverage of unconditioned solutions (*i.i.d.*) to the human coverage ceiling, we observe some clear trends across different solution generation strategies. For the best model (`Gemma-4-31b`), the trends are the most evident, with iterative generation strategy even surpassing the coverage compared to the explicit steer (A+B) baseline in two of the three ontology granularities[4]. This indicates

[4]Verbalized sampling for the weaker models underperforms the unconditioned *i.i.d.* baseline, not because it does not yield diverse solutions, but rather because these models do not generate enough correct samples. `Gemma-4-E4B` generated 3.3 average solutions in verbalized sampling, compared to 8.0 correct solutions in 10 generations in the *i.i.d.* baseline.

Table 2: Algorithmic coverage (AlgoCov) of LLM-generated solutions and corresponding human-written solutions over the two human solutions in the final dataset (referred as A and B). We sampled 10 solutions for each strategy; limiting to 274 *strong*, 104 *medium* and 106 *weak* instances where the ontology-labels are mutually exclusive for the respective classification dimension.

Model	n	*i.i.d.*	verbalized	iterative	steer (A+B)	human
Strong — targets differ in paradigm (274 problems)						
`Gemma-4-31b`	245	28.2 [22,33]	50.2 [44,57]	**60.8** [54,67]	46.6 [41,52]	70.0 [66,75]
`Gemma-4-E4B`	179	34.1 [27,41]	31.3 [25,38]	44.1 [37,51]	**48.3** [42,55]	70.8 [65,76]
`Phi-4`	121	24.8 [17,33]	**29.8** [22,38]	24.8 [17,32]	26.9 [20,34]	67.7 [61,74]
Medium — targets differ in technique (104 problems)						
`Gemma-4-31b`	87	21.8 [14,31]	32.2 [23,43]	37.9 [28,48]	**40.9** [32,50]	59.0 [50,68]
`Gemma-4-E4B`	49	16.3 [6,27]	14.3 [6,24]	32.7 [20,47]	**36.8** [25,49]	56.8 [44,68]
`Phi-4`	26	11.5 [0,23]	15.4 [4,31]	7.7 [0,19]	**26.6** [12,43]	54.9 [38,71]
Weak — targets differ in data structure (106 problems)						
`Gemma-4-31b`	90	10.0 [4,17]	17.8 [10,26]	**23.3** [14,32]	15.5 [8,23]	79.6 [72,87]
`Gemma-4-E4B`	71	5.6 [1,11]	4.2 [0,8]	7.0 [1,13]	**10.1** [4,17]	77.8 [69,86]
`Phi-4`	59	5.1 [0,12]	3.4 [0,8]	5.1 [0,12]	**8.2** [3,15]	76.9 [67,86]

that the algorithmic strategies are–to some extent–latent rather than absent, and can be obtained without external steering. In comparison, the human-written solutions surpass all the generation strategies; with a significant gap in the weak setting where only the data structure differs (79.6% compared to 23.3%).

We also evaluate the impact of explicit steering on the LLM-generated solution's algorithmic strategies. We identified the model's default strategy using the aggregate intersection of the paradigm labels from half of the randomly selected *i.i.d.* solutions with the reference solutions; obtaining a default and not default approach. We found that specifying the model's default approach marginally increases it's paradigm overlap from 91.5% to 93.8%. However, when asked to generate the non-default approach, the overlap increases from 15.4% to 34.7%. This gain was constant across the ontology granularities (+19.8, +16.6, +19.2 pp) for the non-default steered solutions.

Discussion. These experiments suggest that the algorithmic strategies are latent rather than absent, as indicated by the iterative generation matching the oracle steer (A+B) in two granularities and steering toward a non-default paradigm more than doubles the overlap (from 15.4% to 34.7%). The gap in coverage of the default approach still remains significantly higher than the non-default approach (93.8% *vs.* 34.7%), indicating that while steerable, LLMs still struggle to express the less known strategy. In comparison, the human solutions surpass the coverage in all three granularities, with a large gap in the weak tier indicating that diversity in fine-grained data structure choice is harder to elicit than the diversity in programming paradigm. We acknowledge that $\mathrm{AlgoCov}$ as instantiated here, assumes exactly two representative solutions, and does not reward other valid strategies beyond the two solutions. The human results support this argument, with the coverage of 55-80% across the granularities, indicating there are more than two ways to solve the problems. While could be feasible to extend our benchmark beyond 2 representative human solutions, each additional solution would need to be distinct from the existing ones, compounding the number of similarities that needs to be computed.We argue that the human grounded ceiling does help situate the model results better, and we leave the exploration beyond two reference solutions to future work.

6 Conclusion

We introduced a fine-grained algorithmic diversity benchmark for code generation (`DivAlgo`) and presented a a distilled lightweight embedding model with LLM-ensemble annotations to practically measure population-level diversity of model-generated solution at scale. Our extensive examination across multiple models and generation strategies reveals that unconditioned sampling recovers only a fraction of the algorithmic variety present in human solution pools, and that models converge on the more popular human approach. Prompting the models to generate diverse solutions does close some of this gap, unlike minor gains by ensembling heterogeneous models. While steering toward a particular strategy does help improve the coverage of model generated solutions, it still remains below the human ceiling on all three granularities. We conclude that algorithmic diversity is indeed a latent property of current LLMs, but they fail to secure the full variety of human solutions.

AI USE STATEMENT

We have used generative AI tools as an aid to write the code to conduct the experiments, as judges, to generate the figures and tables, and support in early stages of paper writing. We have not used generative AI for ideation, literature review or end-to-end paper or section writing. We have reviewed all AI assisted work, validated the code written for experimentation, cross-checked the illustration artifacts, and all authors had full agency of the paper writing process. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

Marah Abdin, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J Hewett, Mojan Javaheripi, Piero Kauffmann, et al. Phi-4 technical report. *arXiv preprint arXiv:2412.08905*, 2024.

Gabriel Almeida, Ilir Gashi, Vladimir Stankovic, and João R Campos. Effectiveness of llm-based software diversity for reliability improvement–an empirical study. *arXiv preprint arXiv:2607.03174*, 2026.

Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1 (1):11–33, 2004.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. Qwen technical report. *arXiv preprint arXiv:2309.16609*, 2023.

Liguo Chen, Qi Guo, Hongrui Jia, Zhengran Zeng, Xin Wang, Yijiang Xu, Jian Wu, Yidong Wang, Qing Gao, Jindong Wang, et al. A survey on evaluating large language models in code generation tasks. *arXiv preprint arXiv:2408.16498*, 2024.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Enrique Dehaerne, Bappaditya Dey, Sandip Halder, Stefan De Gendt, and Wannes Meert. Code generation using machine learning: A systematic review. *Ieee Access*, 10:82434–82455, 2022.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. In *Findings of the association for computational linguistics: EMNLP 2020*, pp. 1536–1547, 2020.

Dan Friedman and Adji Bousso Dieng. The vendi score: A diversity evaluation metric for machine learning. *arXiv preprint arXiv:2210.02410*, 2022.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Albert Q Jiang, A Sablayrolles, A Mensch, C Bamford, D Singh Chaplot, Ddl Casas, F Bressand, G Lengyel, G Lample, L Saulnier, et al. Mistral 7b. arxiv. *arXiv preprint arXiv:2310.06825*, 10 (3), 2023.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology*, 35 (2):1–72, 2026.

Dipin Khati, Yijin Liu, David N Palacio, Yixuan Zhang, and Denys Poshyvanyk. Mapping the trust terrain: Llms in software engineering-insights and perspectives. *ACM Transactions on Software Engineering and Methodology*, 2025.

Donald E Knuth. Semantics of context-free languages. *Mathematical systems theory*, 2(2):127–145, 1968.

Nathan Lambert, Jacob Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman, Lester James V Miranda, Alisa Liu, Nouha Dziri, Shane Lyu, et al. Tulu 3: Pushing frontiers in open language model post-training. *arXiv preprint arXiv:2411.15124*, 2024.

Florian Le Bronnec, Alexandre Verine, Rio Yokota, and Benjamin Negrevergne. Beyond pass@ k: Redundancy-aware rlvr for multi-sample code generation. *arXiv preprint arXiv:2605.28022*, 2026.

Seonghyeon Lee, Heejae Chon, Joonwon Jang, Dongha Lee, and Hwanjo Yu. How diversely can language models solve problems? exploring the algorithmic diversity of model-generated code. In *EMNLP (Findings)*, pp. 152–167, 2025.

Xiangyang Li, Kuicai Dong, Yi Quan Lee, Wei Xia, Hao Zhang, Xinyi Dai, Yasheng Wang, and Ruiming Tang. Coir: A comprehensive benchmark for code information retrieval models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 22074–22091, 2025.

Xin-Ye Li, Ren-Biao Liu, Yun-Ji Zhang, Hui Sun, Zheng Xie, and Ming Li. Exploring pass-rate reward in reinforcement learning for code generation. *arXiv preprint arXiv:2605.02944*, 2026.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, et al. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434*, 2024a.

Ye Liu, Rui Meng, Shafiq Joty, Silvio Savarese, Caiming Xiong, Yingbo Zhou, and Semih Yavuz. Codexembed: A generalist embedding model family for multiligual and multi-task code retrieval. *arXiv preprint arXiv:2411.12644*, 2024b.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*, 2024.

Yining Lu, Dixuan Wang, Tianjian Li, Dongwei Jiang, Sanjeev Khudanpur, Meng Jiang, and Daniel Khashabi. Benchmarking language model creativity: A case study on code generation. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 2776–2794, 2025.

Robert E Lyons and Wouter Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM journal of research and development*, 6(2):200–209, 1962.

John McCarthy. *A formal description of a subset of ALGOL*. Stanford University. Computer Science Department. Artificial Intelligence . . . , 1964.

Akanksha Narula, Mofasshara Binte Rafique, and Laurent Bindschaedler. The invisible lottery: How subtle cues steer algorithm choice in LLM code generation. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=JrVfxa6aLL.

Lutz Prechelt, Guido Malpohl, Michael Philippsen, et al. Finding plagiarisms among a set of programs with jplag. *J. Univers. Comput. Sci.*, 8(11):1016, 2002.

Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025. URL https://arxiv.org/abs/2412.15115.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Anshul Shah, Thomas Rexin, Elena Tomson, William G Griswold, Leo Porter, and Adalbert Gerald Soosai Raj. Evolution of programmers' trust in generative ai programming assistants. In *Proceedings of the 25th Koli Calling International Conference on Computing Education Research*, pp. 1–11, 2025.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Nino Shervashidze, Pascal Schweitzer, Erik Jan Van Leeuwen, Kurt Mehlhorn, and Karsten M Borgwardt. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12(9), 2011.

CodeGemma Team, Heri Zhao, Jeffrey Hui, Joshua Howland, Nam Nguyen, Siqi Zuo, Andrea Hu, Christopher A Choquette-Choo, Jingyue Shen, Joe Kelley, et al. Codegemma: Open code models based on gemma. *arXiv preprint arXiv:2406.11409*, 2024.

Gemma Team, Sherif El Abd, Vaibhav Aggarwal, Robin Algayres, Alek Andreev, Olivier Bachem, Ian Ballantyne, Cormac Brick, Victor Cărbune, Michelle Casbon, et al. Gemma 4 technical report. *arXiv preprint arXiv:2607.02770*, 2026.

Saiteja Utpala, Alex Gu, and Pin-Yu Chen. Language agnostic code embeddings. In Kevin Duh, Helena Gomez, and Steven Bethard (eds.), *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 678–691, Mexico City, Mexico, June 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.naacl-long.38. URL https://aclanthology.org/2024.naacl-long.38/.

Aozhe Wang, Yuchen Yan, Nan Zhou, Zhengxi Lu, Weiming Lu, Jun Xiao, Yueting Zhuang, and Yongliang Shen. Code-a1: Adversarial evolving of code llm and test llm via reinforcement learning. *arXiv preprint arXiv:2603.15611*, 2026.

Zihan Wang, Siyao Liu, Yang Sun, Ming Ding, and Hongyan Li. CodeContests+: High-quality test case generation for competitive programming. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 5576–5600, Suzhou, China, November 2025. Association for Computational Linguistics. ISBN 979-8-89176-335-7. doi: 10.18653/v1/2025.findings-emnlp.299. URL https://aclanthology.org/2025.findings-emnlp.299/.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Junyi Ye, Jingyi Gu, Xinyun Zhao, Wenpeng Yin, and Guiling Wang. Assessing the creativity of llms in proposing novel solutions to mathematical problems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 25687–25696, 2025.

Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhu Chen. Acecoder: Acing coder rl via automated test-case synthesis. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12023–12040, 2025.

Jiayi Zhang, Simon Yu, Derek Chong, Anthony Sicilia, Michael R Tomz, Christopher D Manning, and Weiyan Shi. Verbalized sampling: How to mitigate mode collapse and unlock llm diversity. *arXiv preprint arXiv:2510.01171*, 2025.

Rui Zhang and Zhichao Lu. Rethinking code similarity for automated algorithm design with LLMs. In *The Fourteenth International Conference on Learning Representations*, 2026. URL `https://openreview.net/forum?id=HIUqeO9OOr`.

A Additional Dataset Construction details

A.1 Preliminary investigation of algorithmic similarity metrics

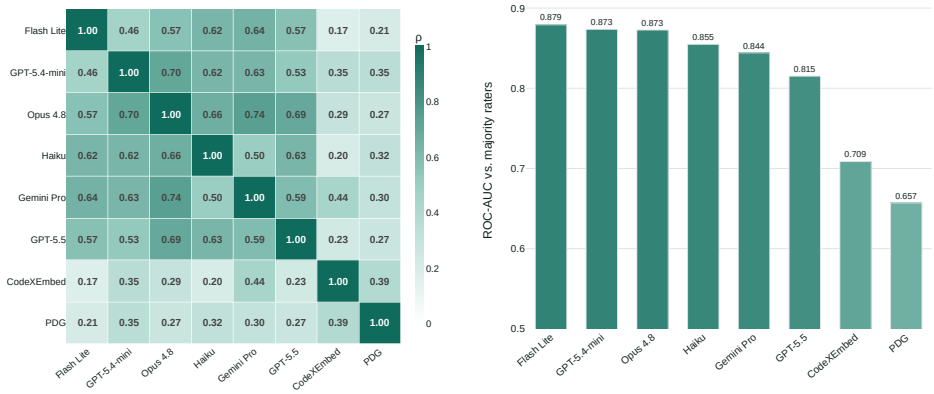


Figure 5: Algorithmic similarity metric agreements on 100 uniformly random solution pairs. (Left) Rank-based Spearman's ρ and (right) ROC-AUC against the majority of metrics; for LLM-judges, the majority of the other 5 LLMs is used, for `CodeXEmbed` and `PDG` all six LLM-judges are used.

To build `DivAlgo` we need to identify algorithmically different solutions among the human provided solutions. While obtaining human annotations for this task would be the best approach to obtain the gold-labels, a recent human-annotation study (Lee et al., 2025) has shown that human correlation dropped from $\kappa = 0.58$ to 0.25 when the edit distance increased from the $100 < x \leq 500$ to $x \geq 500$. For the `CodeContests+` dataset, the average distance between the two C++ solutions is $\geq 1,800$, indicating the difficulty of obtaining high quality human annotations, particularly at scale.

Thus, we propose a systematic investigation of automatic metrics for algorithmic similarity and a two-stage diversity selection process (Section **??**) to build `DivAlgo`. To measure algorithmic similarity, we focus on similarity metrics that go beyond superficial syntactic attributes (such as variable names): 1) program dependency graph (`PDG`) that captures the control and data dependencies for a program[5], 2) a code embedding model, `CodeXEmbed-2b` Liu et al. (2024b), which was the best model on the code retrieval (CoIR) leaderboard Li et al. (2025) in July 2026, and 3) six proprietary LLMs (`Gemini-3.1-pro`, `Gemini-3.1-flash-lite`, `GPT-5.5`, `GPT-5.4-mini`, `Claude-opus-4.8` and `Claude-Haiku-4.5`) used as LLM-as-judge. The models were prompted to give the similarity classification out of four choices: *identical*, *variation*, *distinct* and *very distinct* (prompt in Appendix E). Figure 5 illustrates the correlations between the different similarity metrics for 100 uniformly sampled solution pairs from the `CodeContests+` dataset. We observed that while LLM-judges moderately correlated with each other on the rank-wise Spearman's ρ ($\sim 0.46 - 0.74$), they had a very high AUC (~ 0.85). This indicates that while LLMs disagreed with each other on the thresholds for what determines as the algorithmically similar, their ordering of programs is well-aligned. Comparatively, the other two similarity measures also aligned moderately with the LLM-judges; AUC for `CodeXEmbed` was 0.709, and `PDG` was 0.657.

[5]Post-hoc edits were done on the program dependency graphs to remove several stylistic attributes (see Appendix C for further details).

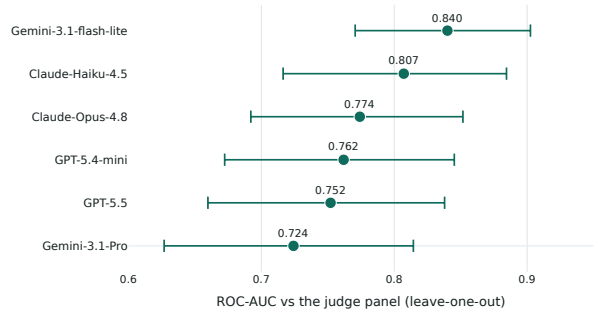


Figure 6: Leave-one-out agreement between six proprietary LLMs for the three-level ontology labeling task.

A.2 Selecting the Program Classification LLM

We used LLM-based classifier to label each programming solution with a three-level algorithm ontology, detailing the programming paradigm (13), technique (64) and data structure (22) used by the solution. To determine the best LLM for this task, we investigate six proprietary LLMs, and found `Gemini-flash-lite-3.1` to have the highest agreement with the other five LLMs, as well as was the cheapest (see Figure 6). We therefore used it to obtain the ontology-based algorithm labels for the 15,246 C++ solutions in the final dataset. We provide the distribution of the individual ontology-level specific labels in Figure 8. While the model additionally provides labels beyond the ontology, we discard them and limit our classification to the predefined ontology labels.

B Calibrating CodeXEmbed as an Evaluation Metric

To select the best evaluation metric that was both open-source and efficient to run as an metric, we first constructed a evaluation subset from our dataset. The subset comprised of 500 random solution pairs, and 200 solution pairs from the primary filtration stage using `PDG` and `CodeXEmbed` similarities (refer to Section **??**). The subset was designed intentionally, the random pairs calibrate the metric over the natural similarity distribution from the dataset, while the filtration-stage pairs oversample the more difficult near-threshold regions that random sampling might not appropriately represent. For each solution pair corresponding to a problem, we used six closed-source models (`Gemini-3.1-pro`, `Gemini-3.1-flash-lite`, `GPT-5.5`, `GPT-5.4-mini`, `Claude-opus-4.8` and `Claude-Haiku-4.5`) to obtain the ground-truth label on a four-point ordinal scale (0 - identical, 1 - variation, 2 - distinct, 3 - very distinct). The majority score from these six models was selected as the ground-truth rating for each instance, and we evaluated our light-weight metrics using the area under the ROC curve to evaluate their alignment.

We evaluated various similarity metrics, from code embeddings `CodeXEmbed` and `jina` , code-graph based `PDG`, and LLM-as-judge using `Llama-3.1-8B` following Lee et al. (2025). We additionally finetuned the two code embedding models, distilling knowledge from the LLMs used to select the most distinct solutions in the dataset curation process (Section **??**). We henceforth refer to these distilled embedding models as `CodeX-SFT` and `jina-SFT`. We trained the models on the labeled data we obtained from LLM-ensemble annotations during the second filtration stage (Section **??**), excluding the 700 instances from the evaluation subset. This training dataset comprised of 12,649 solution pairs from training and 1,546 from the validation split. The output distribution of the majority voted labels obtained from the three proprietary LLM ensemble comprised of 4,465 *identical*, 2,880 *variation*, 4,318 *distinct* and 2,532 *very distinct* solution pairs. The target labels followed the same 4-way ordinal responses from the LLM-judge responses. We used mean-squared error as the loss function for this SFT training, trained for 2 epochs with batch size of 64, $10^{-}5$ learning rate with 5% warmup using 8-bit AdamW.

We provide the meta-evaluation of similarity measures in Figure 7. All the considered evaluation metrics performed better than chance, but the only structural measure `PDG` had the weakest align-

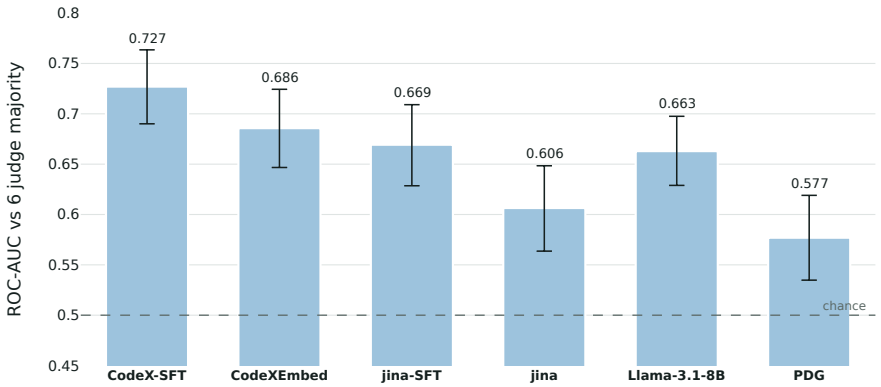


Figure 7: Evaluation of the algorithmic similarity measurement metrics on 700 instances, evaluated against the aggregated majority ground truth from six proprietary models. The confidence interval is obtained via percentile bootstrap over the solution pairs (repeated 3,000 times).

ment with the six judge ensemble, achieving a ROC of 0.577. `Llama-3.1-8B` performs similarly to the `CodeXEmbed` model, although it has more than three times the number of parameters. For both the models, finetuning helped improve the performance of the embedding metrics, with the best performing metric being the `CodeX-SFT`. We therefore used this metric as the basis of our algorithmic diversity evaluation; computing Vendi Score over the pairwise `CodeX-SFT` similarities. We also provide the self-diversity scores of `PDG` and `CodeX-SFT` to provide complementary insights into the diversity of a population of solutions, measured as the mean pairwise distance for all the solutions (see Table 4 and 5).

Validating `CodeX-SFT` on a human-curated algorithmic diversity dataset. Zhang & Lu (2026) published 30 hand-curated pairs of Python programs to measure behavioral algorithmic similarity on four types of variations. Type-1 (4 problems) and Type-2 (3 problems) focus on keeping the approach fixed, with variation in either the execution path or the result respectively. While Type-2 problems do not yield the same results, invalidating them in our algorithmic similarity measurement, that assumes a pre-requisite correctness of the solutions; not all of the four examples in Type-1 can be considered genuinely algorithmically diverse (*e.g.,* changing the order of `i, j, k` in matrix multiplication). On the other hand, Type-3 (6 problems) and Type-4 (17 problems) align well with our desired algorithmic similarity measurements, and hence we limit the emphasis on these two type of problems in this analysis. Type-3 includes two solutions, dissimilar in text, same behavior and same results (*e.g.,* recursive *vs.* iterative implementations of depth-first search) and Type-4 includes the two solutions that have dissimilar text with same result but different behavior (*e.g.,* quick sort *vs.* bubble sort). A good measurement of algorithmic similarity should give high scores for Type-3 problems, and low scores to Type-4 problems.

We present the results from Zhang & Lu (2026) as well as the results from our implementation in Table 3. While the evaluation dataset size is quite small to conduct meaningful statistical analysis, given that it is hand-curated, the signals obtained from it still bolster the quality of our trained embedding metric. We found that our trained evaluation metric, `CodeX-SFT` is able to separate the two types by 0.23 compared to the baseline `CodeXEmbed` model with 0.08 difference. Since the embedding models were trained on C++ solutions, this is a generalizable result as the evaluation solutions were in Python, indicating the impact of learning on the contrastive data. The smaller `Jina` model on the other hand underperforms, which we speculate is a result of overfitting on a small model with 161M parameters compared to the 2.61B parameters in `CodeXEmbed` model. While their proposed approach `BehaveSim` remains far ahead with a difference of 0.54, it is expected as it executes both programs and aligns their intermediate states. This approach limits their similarity measurement to problems with intermediate steps to trace the algorithm's behavior, such as search and clustering, which does not apply to the spectrum of competitive programming problems we investigated in this work.

Table 3: Measuring algorithmic overlap of our trained evaluation metrics on the hand-curated solution pairs across four types of solution pairs in Python ($n = 4, 3, 6$ and 17 pairs for each type respectively).

	Type-1	Type-2	Type-3	Type-4	T3 − T4
embedding metrics (our implementation)					
`CodeX-SFT`	0.93	0.85	0.87	0.65	+0.23
`CodeXEmbed`	0.99	0.97	0.97	0.89	+0.08
`jina-SFT`	0.95	0.95	0.92	0.83	+0.09
`jina-embeddings-v2-base-code`	0.93	0.94	0.92	0.78	+0.14
embedding metrics, as reported in Zhang & Lu (2026)					
`Jina-Code-Embedding`	0.99	0.99	0.90	0.84	+0.06
`Qwen3-Embedding-0.6B`	0.94	0.93	0.87	0.73	+0.14
execution-trajectory metric, as reported in Zhang & Lu (2026)					
`BehaveSim`	0.56	0.73	1.00	0.46	+0.54

C Hyperparameters and Reproducibility Details

Except for the temperature variation study, all the models were sampled at T=0.8, top_p=0.95, N=50 samples per problem over the 1,000 test split problems. The maximum token limit was varied across models depending to ensure under 10% truncation (see Table 4 for individual model hyperparameter). Every solution was compiled with `g++ -O2 -std=c++17` and run against the 15 test cases provided by the original `CodeContests+` benchmark. Program dependence graphs were extracted with the Joern library[6] and the similarity between graphs was computed using a three-iteration Weisfeiler-Lehman subtree kernel Shervashidze et al. (2011). The trained similarity metric `CodeX-SFT` was finetuned with a maximum of 2,048 input tokens. All local experiments were run on four NVIDIA A100 40GB GPUs.

D Results (Extended)

We provide the extended results for the experiments described in Section 5.1 and Section 5.2. Table 4 reports the uncontrolled zero-shot generation results for the 24 LLMs with a human reference. Table 5 reports the controlled diverse code generation results for the three open-weight LLMs for the three strategies: temperature variation, verbalized sampling and iterative constrained generation.

[6]Joern library url: `https://docs.joern.io/export/`. We post-hoc removed some stylistic attributes from the PDG graph to make it less syntax-dependent. Each PDG node was relabeled by dropping the line numbers, replacing user defined variables with 'VAR' and numeric literals by 'NUM'; and the variable name was dropped from the edge dependence to only contain the dependence type (control or data) and the direction.

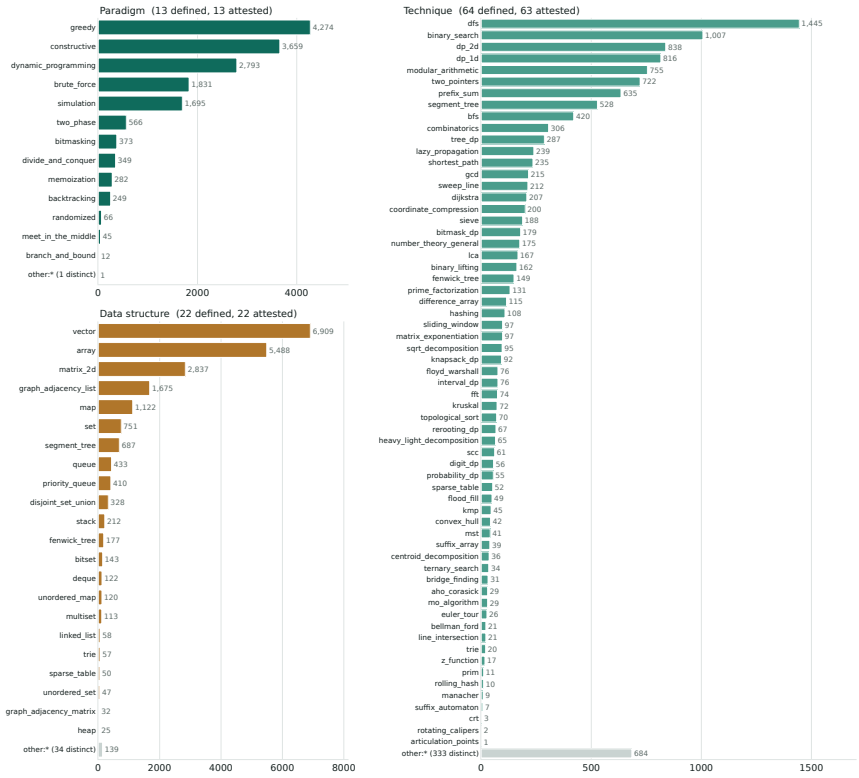


Figure 8: Distribution of ontology labels in the two C++ solutions per programming problem in the algorithmic diversity dataset.

E PROMPTS

We provide the prompts we have used throughout this work in this Section. Figure 14 illustrates the prompt used to classify the solutions' used algorithms following the proposed ontology. For evaluation, Figure 13 presents the four-way ordinal evaluation prompt. For our generation, Figures 15, 16 and 17 present the *i.i.d.*, *iterative* and *verbalized* prompts used to generate solutions. Figure 18 shows the prompt used to obtain the steer (A+B) solutions following specific solution ontology.

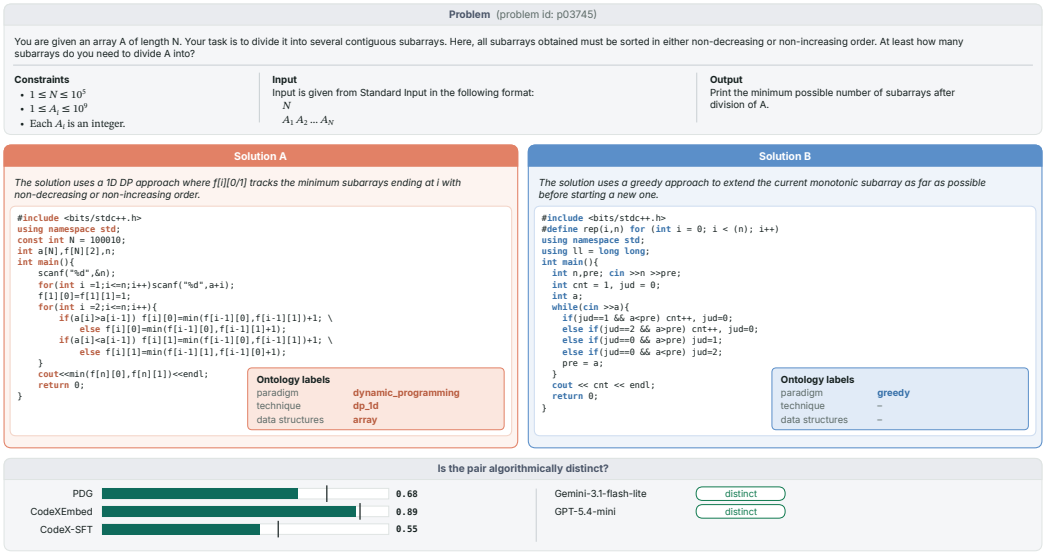


Figure 9: Illustrative example of ***strong*** algorithmic diversity from the train split. The two solutions have different programming paradigm. The bar graphs on bottom left provide the code similarity of different similarity metrics, with the ticks indicating a threshold calibrated on the 10th percentile of randomly selected 150 train instances. Similarity below the threshold is considered algorithmically diverse as per the metric. We also provide the pairwise LLM judgments from the second filtration stage on the bottom right.

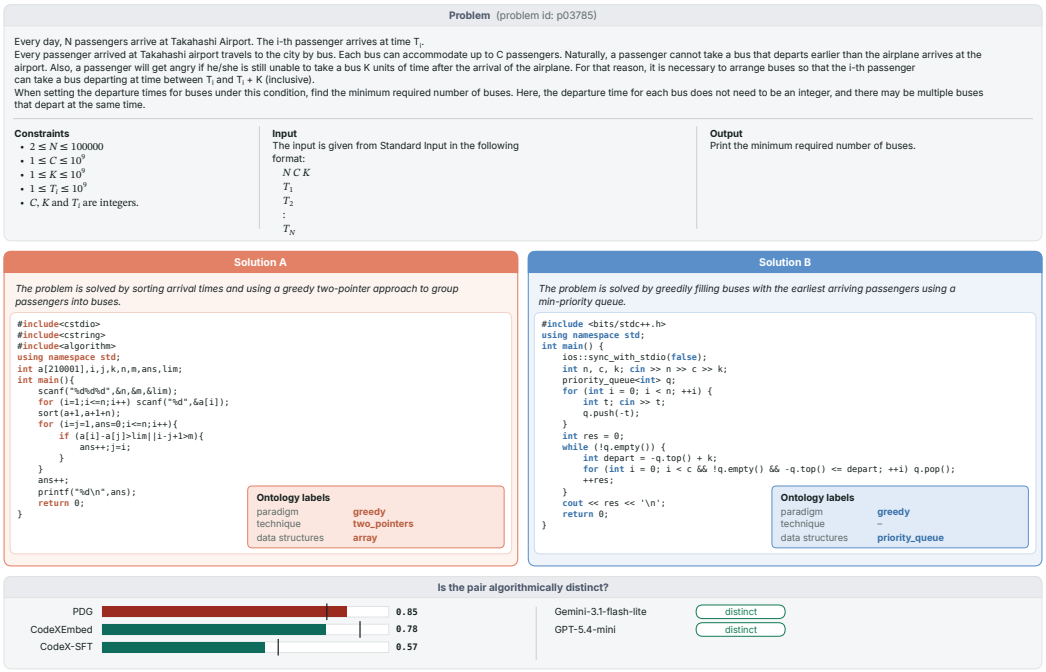


Figure 10: Illustrative example of ***medium*** algorithmic diversity from the train split. The two solutions have the same programming paradigm, but different techniques. The bar graphs on bottom left provide the code similarity of different similarity metrics, with the ticks indicating a threshold calibrated on the 10th percentile of randomly selected 150 train instances. Similarity below the threshold is considered algorithmically diverse as per the metric. We also provide the pairwise LLM judgments from the second filtration stage on the bottom right.

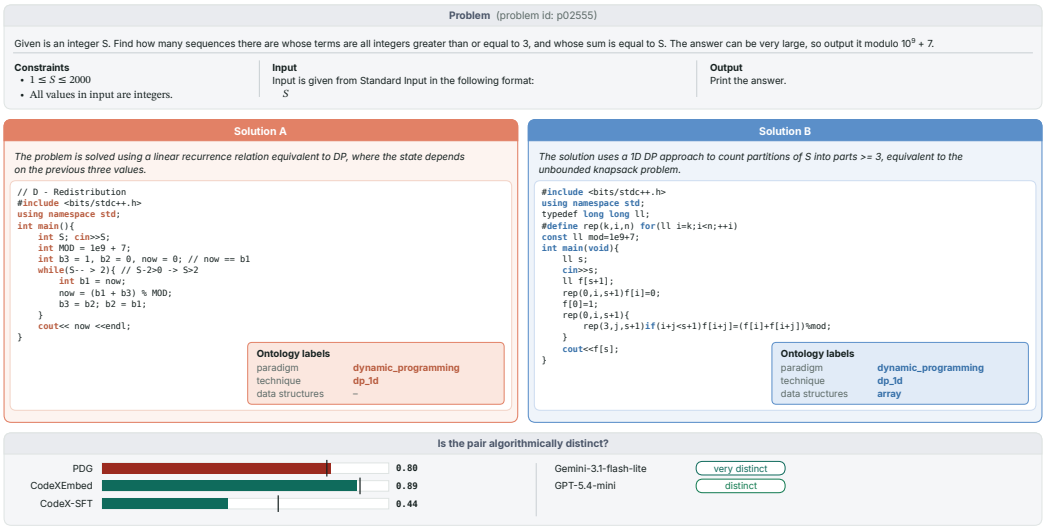


Figure 11: Illustrative example of ***weak*** algorithmic diversity from the train split. The two solutions have the same programming paradigm and techniques, but use different data structures. The bar graphs on bottom left provide the code similarity of different similarity metrics, with the ticks indicating a threshold calibrated on the 10th percentile of randomly selected 150 train instances. Similarity below the threshold is considered algorithmically diverse as per the metric. We also provide the pairwise LLM judgments from the second filtration stage on the bottom right.

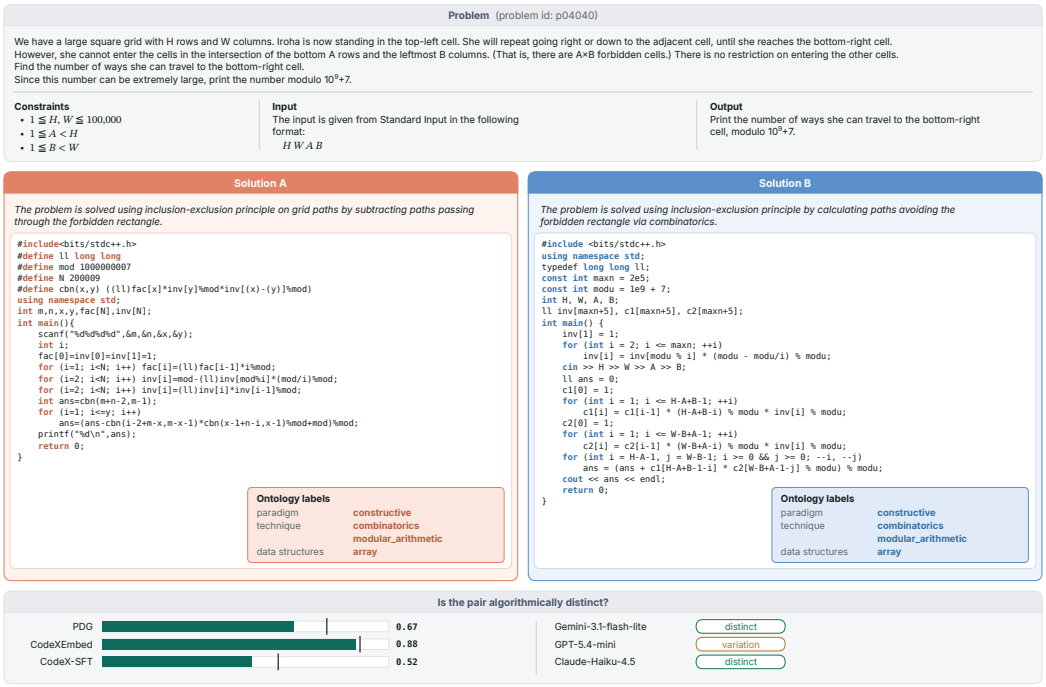


Figure 12: Illustrative example of ***identical*** algorithmic diversity from the train split. The two solutions have the same programming paradigm, techniques, and the data structures; passing through all the filtration steps. The bar graphs on bottom left provide the code similarity of different similarity metrics, with the ticks indicating a threshold calibrated on the 10th percentile of randomly selected 150 train instances. Similarity below the threshold is considered algorithmically diverse as per the metric. We also provide the pairwise LLM judgments from the second filtration stage on the bottom right.

system

```
You are a senior competitive-programming judge evaluating whether two C++ solutions to the same problem use GENUINELY
DIFFERENT algorithmic approaches, or whether they are merely stylistic / surface variations of the same approach.

Output ONLY a JSON object with these fields:
{
  "rating": "<one of: identical | variation | distinct | very_distinct>",
  "justification": "<1-3 sentences explaining the rating>",
  "key_difference": "<one sentence describing the most important algorithmic difference, or 'none' if rating is identical>",
  "complexity_A": "<best-guess time complexity of solution A>",
  "complexity_B": "<best-guess time complexity of solution B>"
}

Rating guide:
- identical: same algorithm + same data structures + same complexity. Only stylistic differences.
- variation: same core algorithm with minor variations (iterative vs recursive of same DP, equivalent traversal
  orders).
- distinct: meaningfully different algorithmic approaches (DP vs greedy; brute force vs optimized; different data
  structures driving different complexity).
- very_distinct: fundamentally different algorithms with different complexity classes or theoretical foundations.

Return raw JSON, no markdown fences.
```

user

```
Problem description:
"""
problem statement, first 2,500 chars
"""

# Solution A
```cpp
solution A source, first 4,000 chars
```

Solution B
```cpp
solution B source, first 4,000 chars
```

Return your JSON verdict now.
```

Figure 13: The LLM-judge prompt used to obtain the 4-way ordinal classification of algorithmic similarity between two programs for the same problem.

system

```
You are an expert competitive-programming reviewer. Given a problem statement and a C++ solution, classify the
solution along three ontology axes using ONLY the controlled-vocab labels listed below.

Output a single JSON object with exactly these keys:
{
  "paradigm":        [list of paradigm labels],
  "technique":       [list of technique labels],
  "data_structures": [list of data_structures labels],
  "notes":           "one short sentence (<=200 chars) explaining the labels"
}

Rules:
- Multi-label allowed; empty list is allowed if no label in that axis applies.
- Use the snake_case `id` exactly as listed. Case-sensitive.
- If a clearly relevant concept is missing from the vocab, use `other:<one_token>` (sparingly).
- Do NOT invent label aliases; use the canonical id.
- Distinguish technique.X vs data_structures.X: assign data_structures.X when the code uses X as a storage container;
  assign technique.X when the algorithm crucially depends on X-specific operations. Both may apply to the same code.
- Output JSON only. No prose before or after.

==== PARADIGM ====
13 ids, each with a one-line disambiguating description and aliases
==== TECHNIQUE ====
64 ids, grouped: search, array, dp, graph, string, math, range, geometry
==== DATA_STRUCTURES ====
22 ids
```

user

```
Problem description:
"""
problem statement, first 6,000 chars
"""

Constraints: time_limit_ms={...}, memory_limit_mb={...}

C++ solution:
```cpp
solution source, first 8,000 chars
```

Return the JSON object now.
```

Figure 14: The prompt used to obtain the ontology labels for the two representative solutions in the final dataset.

system

```
You are an expert competitive programmer. Solve the problem in C++ (C++17). Output ONLY the complete compilable C++
source code wrapped in a single ```cpp ... ``` fenced block. Do not add explanation, headers, or commentary outside
the fence.
```

user

```
Problem:
"""
problem statement, first 6,000 chars
"""

Constraints: time_limit_ms={...}, memory_limit_mb={...}

Write a correct, efficient C++ solution. Read from standard input, write to standard output.
```

Figure 15: The prompt used to generate *i.i.d.* zero-shot solutions for different temperature values.

system

```
You are an expert competitive programmer. For each problem, generate a set of five possible solutions, each within a
separate <response> tag. Each response must include a <text> containing the complete compilable C++
(C++17) source code in a single ```cpp code block, and a numeric <probability>. Please sample at random from the
full distribution of correct approaches.
```

user

```
Problem:
"""
problem statement, first 6,000 chars
"""

Constraints: time_limit_ms={...}, memory_limit_mb={...}

Give 5 solutions that use genuinely different algorithmic approaches. Each must read from standard input and write to
standard output.
```

Figure 16: The prompt used to generate the solutions using the verbalized sampling approach Zhang et al. (2025). We limit to 5 outputs per generation and re-run it 10 times to obtain 50 solutions for direct comparison with the *i.i.d.* setting.

system

```
You are an expert competitive programmer. Solve the problem in C++ (C++17). Output ONLY the complete compilable C++
source code wrapped in a single ```cpp ... ``` fenced block. Do not add explanation, headers, or commentary outside
the fence.
```

user (round 1 is this block alone)

```
Problem:
"""
problem statement, first 6,000 chars
"""

Constraints: time_limit_ms={...}, memory_limit_mb={...}

Write a correct, efficient C++ solution. Read from standard input, write to standard output.
```

user (rounds 2-5 append this)

```
# Previous solution 1
```cpp
this chain's earlier solutions, 2,500 chars each
```

The solution(s) above are already known. Write a solution that uses a FUNDAMENTALLY DIFFERENT algorithmic approach — a
different reduction, a different data structure driving a different complexity, or a different correctness argument.
Do not restate any approach shown above. Output only the new solution in a single ```cpp fenced block.
```

Figure 17: The prompt used to iteratively generate diverse solutions. We limit the iterations to 5 generations and re-run it 10 times to obtain 50 solutions for direct comparison with the *i.i.d.* setting.

Table 4: Algorithmic diversity results for unconditioned zero-shot generations on the 1,000 test set programming problems. We present three success statistics, *pass rate* - percentage of generated solutions that pass all test cases, *Pass@N* - number of instances with at least one correct generation, and *Pass$_{\geq 2}$@N* - number of instances with at least two correct solutions to compute within problem diversity. We also provide the generation statistics, including the median (*med*) and maximum (*max*) number of output tokens, as well as the percentage of solutions that were truncated (*trunc*). For per-problem population diversity, we present the self diversity (**self-div**) of generated solutions as the average distance of all correct solutions using two distance measurements: program dependence graphs (*PDG*) and the trained embedding model *CodeX-SFT* (Section B). We also provide the $\text{Vendi}_{\texttt{CodeX-SFT}}$ (Eq. 1) scores on two settings, *matched* and *all solutions*. *Matched* Vendi score subsamples $k^* = \min(k_{\text{LLM}}, k_{\text{hum}})$ number of solutions from the LLM-generated solutions and the human references over 200 draws without replacement, whereas the *all solutions* Vendi score is computed at the entire pool size of each solution population. We also provide the human reference diversity scores for all 1,000 test instances for reference.

	Success Statistics			**tokens**			**self-div**		$\text{Vendi}_{\texttt{CodeX-SFT}}(k^*$ matched / all solutions)		
Model	*Pass rate*	*Pass@N*	*Pass$_{\geq 2}$@N*	*med*	*max*	*trunc (%)*	*PDG*	*CodeX-SFT*	k^*/k_{LLM}	VS^{LLM}	VS^{hum}
`Gemini-3.8-flash`	80.9	954	945	–	–	0.7	0.046	0.175	24.7 / 42.8	4.18 / 4.84	13.29 / 14.25
`Gemma-4-31b`	76.8	927	913	1,073	8,192	5.1	0.049	0.173	24.7 / 42.0	4.14 / 4.75	13.15 / 14.23
`Gemma-4-26B`	74.0	929	917	2,447	16,384	7.2	0.058	0.171	24.3 / 40.3	3.50 / 3.96	13.01 / 14.26
`Gemma-4-12b`	64.1	881	858	1,576	16,384	10.0	0.054	0.178	23.3 / 37.3	3.71 / 4.13	12.24 / 14.16
`Gemini-flash-lite`	53.6	817	782	–	–	0.0	0.050	0.187	21.9 / 34.2	4.32 / 4.75	11.29 / 13.80
`Gemma-4-E4B`	44.7	736	692	2,132	8,192	0.8	0.066	0.195	21.6 / 32.2	4.23 / 4.64	10.86 / 13.57
`Phi-4`	31.4	678	618	356	2,048	0.1	0.066	0.169	18.5 / 25.3	3.22 / 3.33	8.79 / 13.18
`Qwen3-Coder-30B`	25.8	516	463	824	8,192	5.1	0.067	0.160	19.5 / 27.8	3.22 / 3.34	9.37 / 13.05
`Qwen2.5-Coder-32B`	19.2	409	366	271	2,048	0.1	0.058	0.158	18.4 / 26.2	2.97 / 3.07	8.47 / 12.52
`DeepSeek-V2-Lite`	14.6	356	321	392	2,048	0.1	0.051	0.129	16.9 / 22.7	2.38 / 2.43	7.74 / 12.09
`Codestral-22B`	11.9	388	327	416	2,048	0.2	0.086	0.207	14.4 / 18.0	3.46 / 3.62	6.79 / 12.08
`Qwen2.5-7B`	8.8	308	246	274	2,048	0.1	0.073	0.169	14.6 / 17.6	2.93 / 2.99	6.31 / 11.73
`Qwen2.5-Coder-7B`	8.7	271	224	271	2,048	0.2	0.071	0.170	15.6 / 19.3	3.05 / 3.14	7.01 / 11.80
`Starcoder2-15b`	7.3	270	218	351	2,048	0.2	0.097	0.195	13.7 / 16.4	3.24 / 3.34	6.26 / 11.75
`Mistral-Nemo`	5.5	244	186	257	2,048	0.0	0.080	0.186	12.7 / 14.6	3.07 / 3.14	5.80 / 11.47
`Codegemma-7b-it`	5.2	196	152	236	2,048	0.0	0.074	0.169	14.1 / 16.7	2.95 / 3.01	6.26 / 11.35
`Llama-3.1-8B`	4.7	221	175	338	2,048	0.4	0.104	0.230	11.5 / 13.3	3.54 / 3.60	5.20 / 10.81
`CodeLlama-34B`	3.5	171	137	360	2,048	0.4	0.095	0.202	10.9 / 12.5	3.13 / 3.23	5.07 / 10.70
`CodeQwen1.5-7B`	3.2	186	139	403	2,048	1.6	0.114	0.245	9.8 / 11.0	3.33 / 3.41	4.49 / 11.17
`CodeLlama-13B`	3.1	148	117	303	2,048	0.2	0.092	0.207	11.4 / 12.9	3.08 / 3.11	5.24 / 10.92
`Llama-3.2-3B`	2.7	158	116	305	2,048	0.3	0.097	0.227	10.6 / 11.3	3.26 / 3.27	4.63 / 10.36
`CodeLlama-7B`	2.5	130	101	344	2,048	0.3	0.088	0.206	10.3 / 12.3	2.98 / 3.02	4.77 / 10.47
`Mistral-7B-v0.3`	1.7	118	77	455	2,048	0.1	0.086	0.222	9.6 / 10.8	3.07 / 3.09	4.20 / 9.77
`Qwen2.5-0.5B`	0.9	79	49	247	2,048	1.5	0.107	0.257	7.8 / 8.4	3.04 / 3.04	3.59 / 9.54
`Human reference`	–	1,000	1,000	–	–	–	0.215	0.374	– / 26.6	–	– / 14.27

Table 5: Algorithmic diversity of diversity conditioned generation approaches on the 1,000 test set programming problems. We present the results for three open-weight models over three strategies: temperature variation with 50 generations (*i.i.d.* denotes the standard T=0.8 setting), *verbalized* sampling (10 independent calls per problem asking to generate five solutions per call), and *iterative* generation (10 independent chains per problem with five iterative constrained generations per generation chain). All conditions are evaluated on the same 50 generated solutions, and the statistics and diversity measurements are the same as described in Table 4.

		Success Statistics			**#tokens**			**self-div**		$\text{Vendi}_{\texttt{CodeX-SFT}}(k^*$ matched / all solutions)		
Model	Sampling	*Pass rate*	*Pass$_{\geq 2}$@N*	*k*	*med*	*max*	*trunc (%)*	*PDG*	*CodeX-SFT*	k^*/k_{LLM}	VS^{LLM}	VS^{hum}
`Phi-4`	i.i.d.	31.4	618	25.3	356	2,048	0.09	0.066	0.169	18.5 / 25.3	3.22 / 3.33	8.79 / 13.18
	temp 0.2	33.5	538	31.1	347	2,048	0.08	0.032	0.087	20.6 / 31.1	1.85 / 1.89	9.56 / 12.95
	temp 1.2	27.6	587	23.4	380	2,048	1.76	0.079	0.197	17.7 / 23.4	3.73 / 3.85	8.23 / 12.96
	iterative	24.0	604	19.8	370	2,048	0.02	0.072	0.219	16.7 / 19.8	4.16 / 4.31	8.01 / 13.06
	verbalized	22.7	561	20.2	1,807	13,000	0.01	0.067	0.191	16.8 / 20.2	3.60 / 3.71	8.07 / 13.00
`Gemma-4-E4B`	i.i.d.	44.7	692	32.2	2,132	8,192	0.81	0.066	0.195	21.6 / 32.2	4.23 / 4.64	10.86 / 13.57
	temp 0.2	44.9	681	33.0	2,016	8,192	0.69	0.052	0.172	21.7 / 33.0	3.46 / 3.73	10.91 / 13.61
	temp 1.2	43.3	693	31.2	2,377	8,192	1.29	0.078	0.211	21.4 / 31.2	4.71 / 5.18	10.72 / 13.62
	iterative	44.5	707	31.4	1,760	8,192	0.20	0.071	0.252	21.9 / 31.4	6.18 / 7.00	11.09 / 13.65
	verbalized	38.4	672	17.7	6,242	16,384	2.86	0.090	0.275	15.8 / 17.7	5.79 / 5.97	8.41 / 13.55
`Gemma-4-31b`	i.i.d.	76.8	913	42.0	1,073	8,192	5.09	0.049	0.173	24.7 / 42.0	4.14 / 4.75	13.15 / 14.23
	temp 0.2	76.9	909	42.3	1,073	8,192	5.59	0.040	0.150	24.7 / 42.3	3.42 / 3.83	13.18 / 14.22
	temp 1.2	76.4	918	41.6	1,077	8,192	4.93	0.054	0.186	24.7 / 41.6	4.58 / 5.31	13.16 / 14.25
	iterative	74.2	920	40.3	1,194	8,192	1.76	0.083	0.308	24.9 / 40.3	8.83 / 10.97	13.31 / 14.22
	verbalized	73.0	907	39.2	3,546	16,384	0.17	0.077	0.280	24.4 / 39.2	7.45 / 8.88	13.01 / 14.22

system

```
You are an expert competitive programmer. Solve the problem in C++ (C++17). Output ONLY the complete compilable C++
source code wrapped in a single ```cpp ... ``` fenced block. Do not add explanation, headers, or commentary outside
the fence.
```

user — conditioned on solution A (problem 1178_B)

```
Problem:
"""
problem statement, first 6,000 chars
"""

Constraints: time_limit_ms={...}, memory_limit_mb={...}

Use this specific algorithmic approach:
  paradigm: greedy
  technique: prefix_sum
  data structures: vector

Write a correct, efficient C++ solution that follows exactly this approach. Read from standard input, write to
standard output.
```

user — conditioned on solution B (same problem)

```
Problem:
"""
identical statement and constraints
"""

Constraints: time_limit_ms={...}, memory_limit_mb={...}

Use this specific algorithmic approach:
  paradigm: dynamic_programming
  technique: dp_2d
  data structures: matrix_2d, vector

Write a correct, efficient C++ solution that follows exactly this approach. Read from standard input, write to
standard output.
```

Figure 18: The prompts used to generate the steer (A+B) solutions that are instructed to adhere to a particular algorithmic strategy.