AutoLogi: Automated Generation of Logic Puzzles for Evaluating Reasoning Abilities of Large Language Models

Anonymous ACL submission

Abstract

While logical reasoning evaluation of Large Language Models (LLMs) has attracted significant attention, existing benchmarks predominantly rely on multiple-choice formats that are vulnerable to random guessing, leading to overestimated performance and substantial performance fluctuations. To obtain more accurate assessments of models' reasoning capabilities, we propose an automated method for synthesizing open-ended logic puzzles, and use it to develop a bilingual benchmark, AutoLogi. Our approach features program-based verification and controllable difficulty levels, enabling more reliable evaluation that better distinguishes models' reasoning abilities. Extensive evaluation of eight modern LLMs shows that AutoLogi can better reflect true model capabilities, with performance scores spanning from 35% to 73% compared to the narrower range of 21% to 37% on the source multiple-choice dataset. Beyond benchmark creation, this synthesis method can generate high-quality training data by incorporating program verifiers into the rejection sampling process, enabling systematic enhancement of LLMs' reasoning capabilities across diverse datasets.

1 Introduction

Large Language Models (LLMs) have demonstrated remarkable capabilities across diverse applications (OpenAI, 2024; Anthropic, 2022; Qwen et al., 2025). Among these capabilities, logical reasoning has emerged as a critical skill (Luo et al., 2023; Wei et al., 2022; Yao et al., 2024; Zhu et al., 2024). The increasing emphasis on reasoning capabilities has highlighted the pressing need for reliable evaluation methodologies.

However, the field of reasoning evaluation faces three fundamental challenges: the vulnerability to random guessing, insufficient difficulty variation to differentiate model capabilities, and high human annotation costs in dataset construction. A critical issue lies in the prevalent reliance on closed-ended questions in existing benchmarks, where models are simply required to select answers from predefined options (Liu et al., 2020; Bansal et al., 2023; Zhong et al., 2021; Han et al., 2024; Ismayilzada et al., 2023). This format allows models with minimal reasoning capabilities to achieve substantially overestimated performance, leading to deeply misleading evaluation results that mask true reasoning deficiencies. The rapid advancements in LLMs are leading to the saturation of scores on existing benchmarks, reducing their effectiveness in differentiating model capabilities. Consequently, there is a growing and continuous need for the development of more challenging and distinctive benchmarks to better evaluate model performance. Nevertheless, constructing such high-quality reasoning datasets requires extensive human annotation effort, which substantially limits the scale of available evaluation resources.

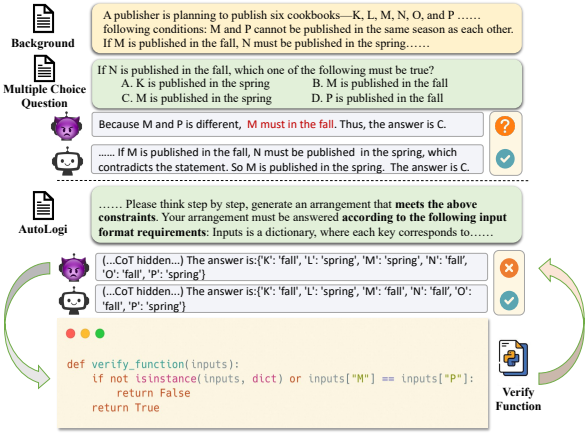

Figure 1: Comparison of evaluation processes between multiple-choice questions and our method. While multiple-choice questions may allow underperforming models to guess the correct answer, our method generates open generative questions, and utilize a verification function to validate the generated solution, providing a more accurate reflection of model performance.

To address these challenges, we introduce a novel method for automatically synthesizing open-ended logic puzzles to construct a reasoning benchmark named AutoLogi. Our approach offers three key advantages: (1) as is illustrated in Figure 1, the open-ended format requires models to construct complete solutions from scratch, significantly mitigating the performance inflation caused by random guessing; (2) automated augmentation generates puzzles with varying logical constraints, enabling balanced difficulty distribution; (3) fully automated generation with minimal human verification substantially reduces benchmark construction costs. These innovations directly address the aforementioned challenges, offering a more reliable, balanced, and scalable evaluation methodology.

The proposed method follows a three-stage pipeline: information extraction from corpora for open-ended question synthesis, program-based verifier generation using advanced LLMs, and dataset augmentation for difficulty balance. Since most generation steps are based on LLMs thus potentially introducing errors, we employ a cross-validation framework where verification functions and traversal functions verify each other. The traversal functions validate puzzle correctness through exhaustive search, while verification functions check the traversal results. Although this provides only necessary conditions, our experiments show it corrects 23% of erroneous data. Building upon this foundation, We further develop a data augmentation method that creates puzzle variants by modifying logical constraints to control difficulty levels, enabling us to expand our dataset to 1,575 English and 883 Chinese puzzles.

Based on our insights that code execution provides verifiable reward signals for reasoning tasks, we explore its utility in synthesizing high-quality training data. Through rejection-sampling with verifiers, we generate both verified correct answers for supervised fine-tuning (SFT) and correct-incorrect answer pairs for Direct Preference Optimization (DPO, Rafailov et al., 2024). This verification mechanism provides stronger guarantees of solution correctness, thereby enabling the collection of higher-quality supervised training data.

Through comprehensive experiments with eight state-of-the-art language models (including GPT-4, Claude, Qwen, and LLaMA), we evaluate our method from two aspects. First, as a benchmark, AutoLogi exhibits superior discriminative power with a wider score distribution (35.25 to 72.61) compared to traditional multiple-choice formats (21.04 to 37.39), better reflecting models' true reasoning capabilities. Second, when used for training, our synthesized dataset leads to substantial improvements on independent reasoning benchmarks, notably improving Qwen's performance on LiveBench from 30% to 35% at 7B scale and from 46% to 52% at 72B scale.

Our core contributions are:

- We develop a method to generate open-ended logic puzzles with controllable complexity, mitigating the performance inflation caused by random guessing and ensuring reliable evaluation through code-based verification.
- We introduce AutoLogi, a bilingual logical reasoning benchmark that better reflects models' reasoning capabilities compared to multiple-choice formats.
- We leverage our method to generate training data that improves model performance across multiple independent benchmarks [1].

2 Related Work

2.1 Logical Reasoning Benchmarks

Question Types Existing logical reasoning datasets primarily use multiple-choice questions (Liu et al., 2020; Bansal et al., 2023; Zhong et al., 2021; bench authors, 2023; Suzgun et al., 2022), true/false questions (Han et al., 2024; Ismayilzada et al., 2023), or classification tasks (Sinha et al., 2019; Tian et al., 2021). These formats, while enabling simple evaluation through keyword matching, are vulnerable to random guessing. Open-ended questions provide more rigorous assessment by requiring complete solution generation. While datasets like Zebra Puzzle (Prosser, 1993) have explored this format, our AutoLogi dataset leverages programmatic verification for more reliable evaluation of logical reasoning abilities.

Dataset Construction Approach The current mainstream approaches to constructing logical reasoning datasets are threefold: human annotation (Clark et al., 2020; Ismayilzada et al., 2023; Han et al., 2024), extraction from academic exams (Liu et al., 2020; Bansal et al., 2023; Zhong et al., 2021), and synthesize (Saparov and He, 2022; Sinha et al., 2019; White et al., 2024). Among the

[1] We will release our complete implementation, including code, training and evaluation datasets.

three methods, synthesize is most cost-effective but usually limited in diversity since they heavily rely on rule-based templates.

Our AutoLogi dataset is synthesized, but different from previous methods, we employ LLMs to reformulate questions from established academic reasoning assessments (such as LogiQA (Liu et al., 2020) and AR-LSAT (Zhong et al., 2021), which are derived from real human logical reasoning tests), thus ensuring rich linguistic quality and logical, real-world connected contexts. We design our synthesis process to explicitly control difficulty levels (discussed in 3.2), allowing for better discrimination of model reasoning abilities.

2.2 Program-aided Methods

LLMs, despite their capabilities, can generate factually incorrect responses. Integrating reliable feedback mechanisms, such as code interpreters, offers a potential solution. This approach, termed Program-aided Language model or Program of Thought, has been shown by Gao et al. (2023) and Chen et al. (2023) to outperform Chain of Thought methods across various domains, particularly in tasks requiring definitive answers (Wang et al., 2024b,a; Lu et al., 2023). Inspired by (Dong et al., 2024)'s use of program-based verifiers in training, we combine this approach with rejection sampling to generate our training data.

3 Method

As shown in Figure 2, our method consists of three stages: *Puzzle Formulation*, *Format & Verifiers Generation*, and *Data Augmentation*.

3.1 Puzzle Formulation

The Puzzle Formulation stage takes a source corpus containing puzzle-related content as input and leverages advanced LLMs with direct prompting (detailed prompts in Appendix A.4) to extract and restructure the text. The source corpus should provide background information suitable for constructing logical puzzles, for example, questions from existing multiple-choice reasoning benchmarks. The output consists of two key components: *Background* and *Logical Constraints*. The **Background** provides the context and basic elements of the puzzle, defining the objects to be arranged and their value ranges. The **Logical Constraints** specify the conditions limiting valid arrangements. These components, combined with a request for test LLMs to generate any valid arrangement satisfying these constraints, as illustrated in Figure 1, form the puzzle stem.

A notable strength of this transformation approach lies in its broad applicability - as long as the source text contains background information related to logical reasoning, it can be converted into our open-ended format, regardless of question types or the presence of answers.

3.2 Format & Verifiers Generation

Stage 2 focuses on establishing a reliable evaluation mechanism. We design a program-based verification scheme consisting of three tightly coupled components: **Format Requirement** specifies the expected JSON output structure (including **Arrangement Format** and **Arrangement Example**); **Verifiers** contain program-based format verifier and constraint verifier for checking format compliance and logical constraints; **Traversal Function** searches for all possible valid solutions. This code execution-based verification approach effectively overcomes limitations of traditional evaluation methods: since our logical puzzles may have multiple valid answers (which can grow exponentially when logical constraints are sparse), rule-based matching becomes impractical; meanwhile, model-based scoring (LLM-as-a-judge) introduces instability in complex tasks (discussed in Section 5.4).

Given the strong correlation between Format Requirement and Verifiers, we generate these two components simultaneously using advanced LLM, with Background and Logical Constraints from Stage 1 as input. Subsequently, we generate the Traversal Function based on the produced Format Requirement and Verifiers.

To ensure the reliability of the evaluation mechanism, we propose a cross-validation method to check the correctness of LLM-generated Verifiers and Traversal Function. We run the Traversal Function to examines all possible combinations within the value ranges defined in Background (**Domain space**), and validates them through Verifiers to determine the existence of solutions satisfying all constraints (**Solution space**). When no solution is found (empty solution space) or syntax errors occur, a regeneration is triggered. Through the verification, we ensure that each problem has at least one valid solution, which is a necessary condition for problem validity. The effectiveness of Traversal Function are discussed in Section 5.4.

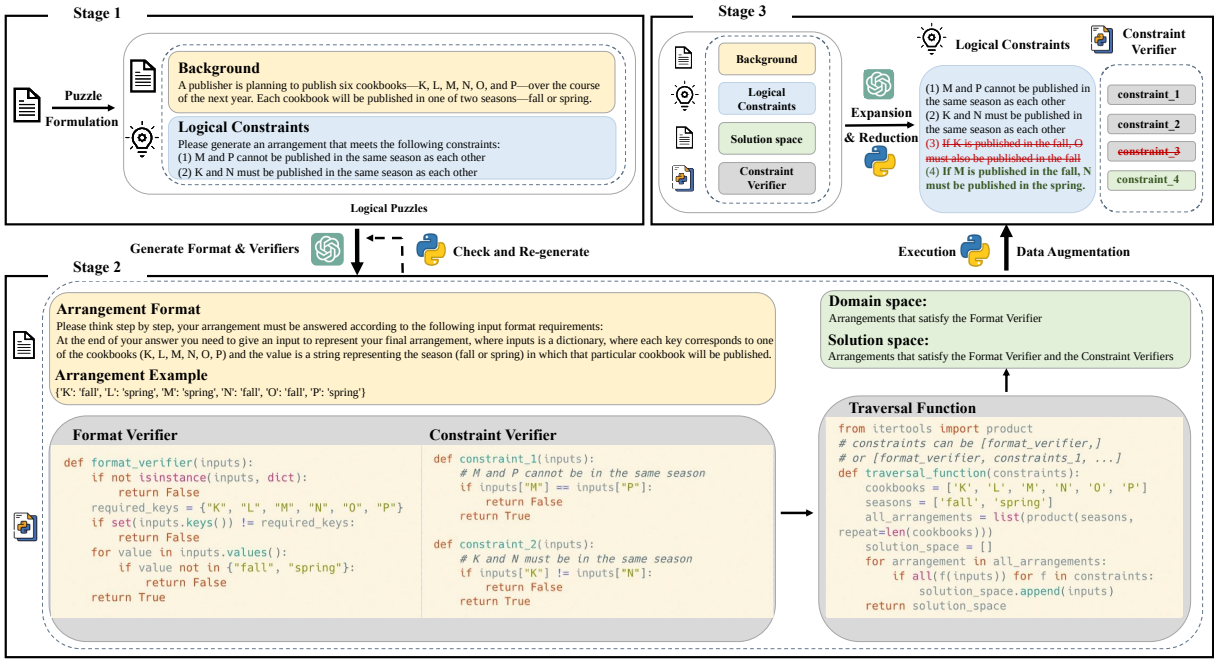

Figure 2: An overview of our method. The process consists of three stages: **Stage 1** formulates logic puzzles by extracting background information and constraints from a source corpus. **Stage 2** uses large language models (LLMs) to generate verifiers, which are programs that check puzzle solutions and ensure correct formatting. **Stage 3** augments the puzzles by adding or removing constraints to create varying difficulty levels. All three stages leverage powerful LLMs, such as GPT-4, for generation.

3.3 Data Augmentation

Stage 3 performs data augmentation through two complementary techniques, Reduction and Expansion, to construct a dataset with balanced difficulty distribution, enabling better discrimination of models' logical reasoning capabilities.

In the **Reduction** process, we randomly select and remove one logical constraint along with its corresponding components in the Verifier. By reducing the number of constraints, the problem is simplified, yielding more problems of lower difficulty. For each problem, we iteratively remove logical constraints to generate new problems until only one constraint remains.

In the **Expansion** process, we utilize advanced LLMs to generate additional constraints and their corresponding Verifiers, taking as input the Background and Logical Constraints from Stage 1, along with the solution space and Verifier from Stage 2. To ensure data quality, we leverage the Traversal Function developed in Stage 2 to verify the solvability of newly generated problems. The expansion process terminates when either the maximum number of attempts is reached or the solution space size reduces to one.

3.4 Synthesizing Training Data

Beyond benchmark construction, our data synthesis method can also be used to generate model training data through rejection-sampling with a verifier, naturally obtaining two categories: verified correct responses for Supervised Fine-Tuning (SFT) and pairs of correct-incorrect responses for DPO.

A significant advantage of this approach is that rejection sampling with program-based verifiers produces more accurate training data. Unlike multiple-choice questions that risk accepting responses where models guess correctly despite flawed reasoning, potentially introducing noise into the training data, our method ensures accuracy by using program-based verifier to examine model responses.

4 Dataset Construction

To construct the AutoLogi dataset, we leveraged two established logical reasoning datasets: AR-LSAT (Zhong et al., 2021) and LogiQA (Liu et al., 2020). Both datasets comprise multiple-choice questions designed to assess logical reasoning capabilities. The complete set of prompts utilized in our study is detailed in Section A.4.

	Source Corpus		Stage 1	Stage 2	Stage3
	# Backgrounds	# Samples	# Sample	# Samples	# Samples
EN	40	230	210	206	1575
ZH	90	123	147	139	883

Table 1: Statistics of our testing data. The AutoLogi benchmark corresponds to the sample number of stage 2, and the AutoLogi (Augumented) corresponds to stage 3.

Testing Set Construction We applied our proposed method to construct the testing set through three stages, with detailed statistics presented in Table 1.

First, we employed GPT-4 to filter suitable samples from the source corpus, resulting in 230 English questions from AR-LSAT (40 distinct backgrounds) and 123 Chinese questions from LogiQA (90 backgrounds). In stage 1, we applied Puzzle Formulation to transform these multiple-choice questions into open-ended puzzles. To maximize coverage, we created separate cases by pairing each question stem with individual options. After deduplication, this yielded 210 English and 147 Chinese open-ended logic puzzles.

In stage 2, we utilized both GPT-4 and GPT-4o to generate format requirements and verifiers, as their complementary capabilities helped address model-specific limitations. Finally, in stage 3, we applied data augmentation techniques to expand our dataset to 1,575 English and 883 Chinese puzzles.

Training Set Construction We utilize the cleaned training sets from LogiQA and AR-LSAT as source corpora, applying our proposed synthesis method to generate 1,675 Chinese samples and 5,064 English samples as $D_{training}$. The rejection sampling process involves two models: Qwen2.5-7b-instruct and Qwen2.5-72b-instruct, each performing rejection sampling with 8 rounds per input to obtain corresponding candidate responses.

By employing our verifier to evaluate these candidate responses, we naturally derive two types of training datasets:

- D_{sft}: A collection of responses verified as correct, serving as high-quality demonstrations for supervised fine-tuning.
- D_{dpo}: Pairs of correct and incorrect responses, forming natural positive-negative sample pairs (x, y_w, y_l) for DPO.

Through eight rounds of rejection sampling, as presented in Table 2, we constructed two datasets: D_{sft} and D_{pref}. D_{sft} was formed by excluding examples without correct solutions, while D_{pref} was further refined by removing instances lacking contrastive pairs. Given their trivial nature, single-constraint puzzles were excluded from both datasets.

	D_{training}	D_{dpo}(7b)	D_{dpo}(72b)	D_{sft}(72b)
EN	5064	2877	2349	3724
CN	1675	901	621	1170

Table 2: Statistics of our training data.

5 Experiment

5.1 Experimental Setup

Models We evaluate 8 models in our experiments: Qwen2.5-7B/72B-Instruct (Qwen et al., 2025), LLama3.1-8B/70B/405B-Instruct (MetaAI, 2024), GPT-3.5-Turbo (OpenAI, 2022), GPT-4o-2024-08-06 (OpenAI, 2024), and Claude-3.5-Sonnet-20240620 (Anthropic, 2022).

Evaluation Datasets We conduct evaluations on two versions of our AutoLogi benchmark: the base version (AutoLogi) and its augmented version (AutoLogi Augmented). We also evaluate on the original multiple-choice datasets that served as our source corpus - AR-LSAT (Zhong et al., 2021) and LogiQA (Liu et al., 2020). To analyze the reliability of our benchmark, we compare the performance patterns with two established reasoning benchmarks: MUSR (Sprague et al., 2024) and LiveBench (Reasoning, 2024-08-31) (White et al., 2024). For all evaluations, we report the mean and standard deviation across 5 independent runs to ensure reliable results.

5.2 Benchmark Results

We evaluate the effectiveness of our synthesis method in constructing high-quality logical reasoning benchmarks through extensive experiments across eight modern LLMs.

Superior Assessment Capability Experimental results in Table 3 demonstrate AutoLogi's

Model	AutoLogi		AutoLogi(Augmented)		AR-LSAT	LogiQA	MUSR	LiveBench
	EN	CN	EN	CN				
Qwen2.5-7b-instruct	$27.28_{\pm 2.41}$	$30.94_{\pm 1.76}$	$43.64_{\pm 1.25}$	$42.08_{\pm 1.50}$	$22.70_{\pm 0.84}$	$34.42_{\pm 1.38}$	$47.14_{\pm 0.73}$	30.67†
Qwen2.5-72b-instruct	$53.50_{\pm 0.93}$	$54.10_{\pm 0.28}$	$68.18_{\pm 0.77}$	$63.92_{\pm 0.56}$	$31.65_{\pm 1.89}$	**47.92**$_{\pm 1.37}$	$54.21_{\pm 0.41}$	46.00†
LLama3.1-8b-instruct	$25.53_{\pm 3.64}$	$17.41_{\pm 3.61}$	$37.96_{\pm 1.41}$	$23.69_{\pm 1.33}$	$22.61_{\pm 2.47}$	$25.04_{\pm 1.40}$	$49.63_{\pm 0.58}$	15.33†
LLama3.1-70b-instruct	$53.79_{\pm 2.92}$	$42.59_{\pm 2.41}$	$62.47_{\pm 0.96}$	$53.77_{\pm 0.95}$	$30.26_{\pm 1.36}$	$36.25_{\pm 2.55}$	$57.28_{\pm 0.76}$	40.67†
LLama3.1-405b-instruct	$59.42_{\pm 3.04}$	$56.98_{\pm 1.79}$	$70.43_{\pm 1.39}$	$65.39_{\pm 1.07}$	$32.61_{\pm 1.94}$	$39.35_{\pm 1.10}$	$57.68_{\pm 0.67}$	53.33†
GPT-3.5-Turbo	$18.93_{\pm 3.83}$	$22.30_{\pm 2.39}$	$35.25_{\pm 0.81}$	$34.47_{\pm 1.39}$	$21.04_{\pm 1.71}$	$26.34_{\pm 4.13}$	$48.33_{\pm 1.07}$	26.67†
GPT-4o-2024-08-06	**62.04**$_{\pm 1.59}$	$55.54_{\pm 3.78}$	**72.61**$_{\pm 0.76}$	$66.70_{\pm 1.25}$	**37.39**$_{\pm 0.91}$	$43.74_{\pm 1.30}$	59.95 ± 1.13	54.67†
Claude-3.5-sonnet	$60.97_{\pm 0.88}$	**60.29**$_{\pm 3.19}$	$72.53_{\pm 0.82}$	**68.24**$_{\pm 0.98}$	$34.35_{\pm 1.76}$	$46.83_{\pm 3.24}$	**60.53**$_{\pm 0.63}$	**58.68**†

Table 3: Results on the original multiple-choice datasets (AR-LSAT, LogiQA), and our benchmarks (AutoLogi and Augmented AutoLogi). The accuracy was reported with a standard deviation (std) of 5 trials. Results marked with † are sourced from the LiveBench leaderboard(2024-08-31).

Model	AutoLogi		AR-LSAT	LogiQA	MUSR	LiveBench
	EN	CN				
Baseline Qwen2.5-7b-instruct	43.64	42.08	22.70	34.42	47.14	30.67†
+*Self-Alignment DPO*	$48.80_{+5.16}$	$45.39_{+3.31}$	$26.09_{+3.39}$	$38.05_{+3.63}$	$47.57_{+0.43}$	$35.73_{+5.06}$
+*Strong-to-Weak RFT*	$48.33_{+4.69}$	$47.54_{+5.46}$	$27.30_{+4.60}$	$35.93_{+1.51}$	$49.15_{+2.01}$	$30.07_{-0.60}$
Baseline Qwen2.5-72b-instruct	68.18	63.92	31.65	47.92	54.21	46.00†
+*Self-Alignment DPO*	$74.79_{+6.61}$	$69.54_{+5.62}$	$38.70_{+7.05}$	$48.14_{+0.22}$	$56.48_{+2.27}$	$52.13_{+6.13}$

Table 4: Performance comparison across different training settings. Results report the average of five trials, with standard deviations shown in Section A.8. Subscripted values indicate performance gains over the baseline model, and the best accuracy for each setup is highlighted in green. Results marked with † are sourced from the LiveBench leaderboard(2024-08-31).

distinct advantage in model capability assessment compared to traditional benchmarks. On conventional multiple-choice datasets (AR-LSAT, LogiQA), models achieve similar scores within a narrow range, making it difficult to distinguish their true capabilities. In contrast, both AutoLogi and Augmented AutoLogi reveal more pronounced performance gaps between models, enabling finer-grained capability assessment. For instance, while Qwen2.5-72b achieves unexpectedly high scores on LogiQA, this performance anomaly diverges from its relative standings across other benchmarks, suggesting potential evaluation bias in traditional tests. Notably, AutoLogi yields clear performance stratification among models, and this hierarchical distribution strongly correlates with established benchmarks (MUSR, LiveBench), validating its reliability as an evaluation framework.

Bilingual Evaluation Capability A key feature of AutoLogi is its provision of parallel English and Chinese evaluation benchmarks. Our analysis reveals that most models maintain comparable performance across both languages, indicating robust bilingual capabilities. However, we observe notable variations in cross-lingual performance: LLaMA-series models demonstrate significantly lower performance on Chinese tests compared to English, indicating limitations in Chinese language processing. In contrast, Claude, Qwen, and GPT3.5 exhibit more balanced performance across both languages, suggesting superior cross-lingual generalization ability. This bilingual evaluation dimension provides crucial insights into models' language-agnostic reasoning capabilities.

5.3 Training Results

We investigate the utility of our synthesized data in enhancing models' logical reasoning abilities through training experiments.

Training Settings We investigate two training settings. The first is **Self-Alignment**, where we use the training model itself to perform rejection sampling to generate preference data D_{pref} (containing pairs of correct and incorrect answers), followed by reinforcement learning training using Direct Preference Optimization (DPO, Rafailov et al., 2024). The second setting is **Strong-to-weak Distillation** based on RFT (Rejection sampling Fine-Tuning, Yuan et al., 2023). In this approach, we leverage a more powerful model (Qwen2.5-72b-instruct) to

perform rejection sampling to generate SFT data D_{sft}, which is then used for fine-tuning the target model.

Significant In-domain Improvements As shown in Table 4, our optimization methods achieve substantial improvements on the AutoLogi benchmark. Notably, the DPO method brings significant performance gains of 6.61% and 5.62% on English and Chinese tests respectively for Qwen2.5-72b, demonstrating the effectiveness of our optimization approach on the target tasks.

Broad Cross-domain Generalization Models trained on our synthesized data demonstrate strong generalization capability on tasks with similar problem context but entirely different problem formats. For instance, Qwen2.5-72b achieves a 7.05% improvement on the AR-LSAT test set. The enhanced performance extends to tasks with substantially different source distributions, as evidenced by a 6.13% performance gain on the LiveBench test set, thoroughly validating the immense potential of our proposed synthesis approach in generating high-quality training data to enhance models' general logical reasoning capabilities.

Relationship between Model Scale and Optimization Effects Larger models demonstrate more substantial improvements in performance. Across various benchmarks, calculating the mean improvements under the DPO setting reveals that 72B models achieve higher average performance gains (4.65%) compared to their 7B counterparts (3.50%). This suggests that models with stronger foundational capabilities can further unlock their logical reasoning potential when leveraging high-quality synthetic training data.

5.4 Analysis

Augmentation Effectiveness Analysis As illustrated in Figure 3, experimental results demonstrate that all models experience accuracy degradation with increasing number of constraints (the number of logical rules per puzzle) and decreasing Solution Space Ratio (the proportion of valid solutions within the total possibility space, representing theoretical random-guess probability). The consistent decline in model accuracy indicates increasing problem difficulty, and the significant correlation between these metrics validates that our augmentation strategy successfully generates problems across **diverse difficulty levels**. The baseline

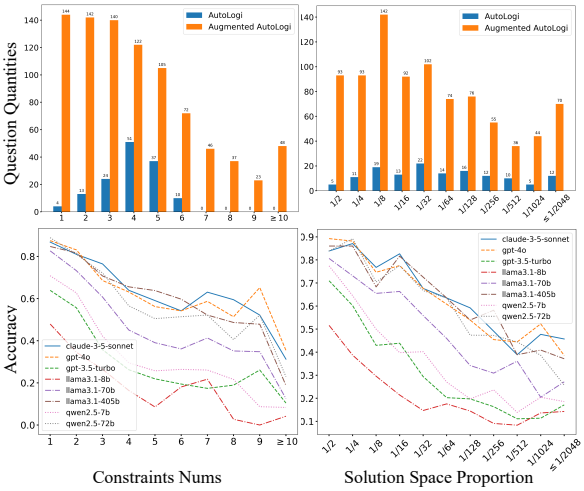

Figure 3: The question quantities on the Chinese subset of AutoLogi before and after data augmentation, and the accuracy of eight models across different constraints and solution space proportions. The figure of English subset can be found in Appendix A.1.

Autologi dataset exhibited inherent limitations in its difficulty distribution, as problems were exclusively derived from the source corpus. This dependency resulted in an unbalanced complexity spectrum, with a maximum of 6 constraints per problem and a sparse representation of higher-difficulty instances. Our augmentation methodology addresses these limitations by introducing a more **uniform distribution** of problem complexities, thereby creating a more balanced and representative dataset for logical reasoning evaluation.

Cross-Validated Solution Existence Analysis To ensure dataset quality, we incorporate traversal functions in Stage 2 to verify the existence of valid solution paths. This verification mechanism proves essential in detecting incorrect generated verifiers and invalid problems before they enter our dataset. For instance, in our experiments with the AutoLogi Chinese subset, among the 139 problems initially generated from 147 information units, 23% were identified as unsolvable(11% in English subset). Furthermore, during Stage 3's augmentation process, where additional logical constraints are introduced, the traversal function becomes indispensable as each constraint addition risks creating unsolvable problems, with approximately 30% of LLM-generated additional constraints leading to unsolvable puzzles. Through this cross-validation method, we effectively maintain dataset integrity by eliminating invalid problems throughout our pipeline.

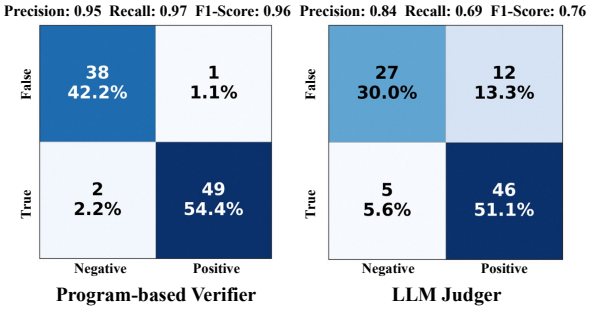

Figure 4: The precision and recall of evaluations using our verification function (Program-based Verifier) and GPT-4 as the evaluator (LLM Judger). True/False indicates the ground truth correctness of the answer. Positive/Negative is the output label predicted by Program-based Verifier or LLM Judger.

Human Alignment Analysis To evaluate the effectiveness of program-based Verifier as an evaluation tool, we conducted a comparative experiment focusing on human alignment. We randomly selected 90 AutoLogi reasoning problems from diverse backgrounds and utilized Claude-3.5-Sonnet to generate responses as the model outputs to be evaluated. To ensure evaluation reliability, we collected voting results from three human evaluators as the ground truth. In our evaluation methodology, we employed both the program-based Verifier for direct answer validation and GPT-4-Turbo as an LLM judge. The experimental results, illustrated through the confusion matrix in Figure 4, demonstrate a significant performance disparity between the two evaluation methods: the program-based Verifier showed only 3 mismatches out of 90 cases (F1 score: 0.96), while the LLM judge exhibited 17 mismatches (F1 score: 0.76). Figure 5 also presents a representative case where the LLM judge made an incorrect assessment while the program-based Verifier provided accurate evaluation.

Beyond superior accuracy, the program-based Verifier demonstrates substantial advantages in computational efficiency and cost-effectiveness, particularly crucial for large-scale evaluations. These experimental findings strongly validate the reliability and practicality of utilizing program-based Verifier as an evaluation tool.

Error Analysis Table 3 reveals that even advanced LLMs exhibit error rates exceeding 30% in logical reasoning tasks. To gain a comprehensive understanding of these limitations, we conducted an in-depth error analysis using Claude-3.5-Sonnet as our representative model. Based on the 90 problems from our human alignment experiment in Section 5.4, we enlisted human annotators to provide detailed explanations for all failure cases. Through systematic analysis of these explanations, we categorized the error sources into four primary types: Incorrect Logical Inference (incorrect deductive steps, 81%), Contradictory Conclusion (inconsistencies between inference and conclusions, 13%), Unanalyzed Condition (failure to consider certain given logical constraint, 3%), and Inconsistent Format (structural or presentational issues in responses, 3%). Detailed statistical analysis of these error patterns and specific case studies for each category are thoroughly discussed in Appendix A.2 and Appendix A.3, respectively.

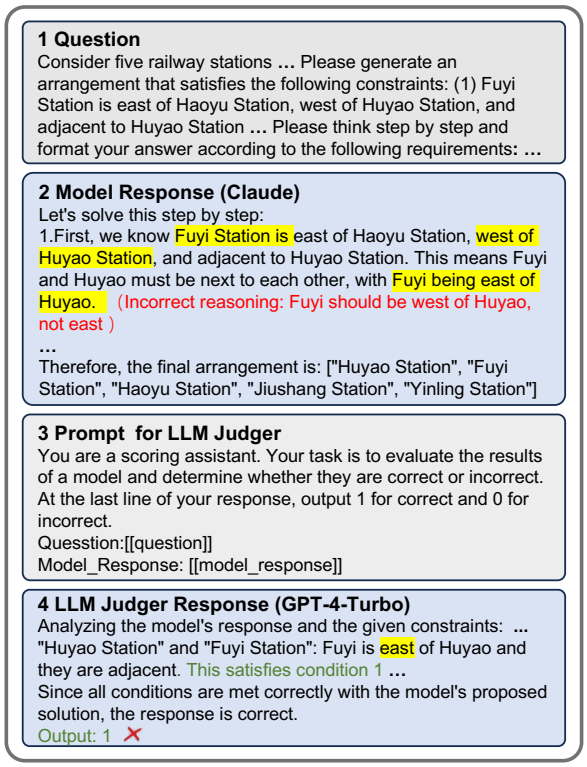

Figure 5: An example of the LLM Judger making mistakes in verifying model responses.

6 Conclusion

In conclusion, we present a method for automatically synthesizing open-ended logic puzzles, which we use to create the bilingual benchmark AutoLogi and high-quality training data. Our approach provides more accurate assessment of LLMs' reasoning capabilities by mitigating random guessing and enabling controllable difficulty levels, while the synthesized training data proves effective in enhancing models' reasoning abilities.

Limitations

Our methods may contains the following limitations:

Dependence on LLMs Our method relies on advanced large language models (LLMs) such as GPT-4 for generating logic puzzles and verification functions. If the only accessible LLMs have limited capacity, our method may suffer from a high failure rate in synthesizing the questions. However, as LLMs continue to advance, we expect the effectiveness and reliability of our approach to improve accordingly.

Verification Function Limitations Although AutoLogi uses program-based verification to handle multiple valid solutions, this approach does not guarantee that the verification functions are perfect, especially in complex conditions. In our experiment, we still find there is about 3% of validation results contain errors (Section 5.4).

References

Anthropic. 2022. Claude 3.5 sonnet.

Srijan Bansal, Semih Yavuz, Bo Pang, Meghana Bhat, and Yingbo Zhou. 2023. Few-shot unified question answering: Tuning models or prompts? *Preprint*, arXiv:2305.14569.

BIG bench authors. 2023. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *Transactions on Machine Learning Research*.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2023. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Preprint*, arXiv:2211.12588.

Peter Clark, Oyvind Tafjord, and Kyle Richardson. 2020. Transformers as soft reasoners over language. *Preprint*, arXiv:2002.05867.

Guanting Dong, Keming Lu, Chengpeng Li, Tingyu Xia, Bowen Yu, Chang Zhou, and Jingren Zhou. 2024. Self-play with execution feedback: Improving instruction-following capabilities of large language models. *Preprint*, arXiv:2406.13542.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. *Preprint*, arXiv:2211.10435.

Simeng Han, Hailey Schoelkopf, Yilun Zhao, Zhenting Qi, Martin Riddell, Wenfei Zhou, James Coady, David Peng, Yujie Qiao, Luke Benson, Lucy Sun, Alex Wardle-Solano, Hannah Szabo, Ekaterina Zubova, Matthew Burtell, Jonathan Fan, Yixin Liu, Brian Wong, Malcolm Sailor, Ansong Ni, Linyong Nan, Jungo Kasai, Tao Yu, Rui Zhang, Alexander R. Fabbri, Wojciech Kryscinski, Semih Yavuz, Ye Liu, Xi Victoria Lin, Shafiq Joty, Yingbo Zhou, Caiming Xiong, Rex Ying, Arman Cohan, and Dragomir Radev. 2024. Folio: Natural language reasoning with first-order logic. *Preprint*, arXiv:2209.00840.

Mete Ismayilzada, Debjit Paul, Syrielle Montariol, Mor Geva, and Antoine Bosselut. 2023. Crow: Benchmarking commonsense reasoning in real-world tasks. *Preprint*, arXiv:2310.15239.

Jian Liu, Leyang Cui, Hanmeng Liu, Dandan Huang, Yile Wang, and Yue Zhang. 2020. Logiqa: A challenge dataset for machine reading comprehension with logical reasoning. *Preprint*, arXiv:2007.08124.

Pan Lu, Baolin Peng, Hao Cheng, Michel Galley, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, and Jianfeng Gao. 2023. Chameleon: Plug-and-play compositional reasoning with large language models. *Preprint*, arXiv:2304.09842.

Man Luo, Shrinidhi Kumbhar, Mihir Parmar, Neeraj Varshney, Pratyay Banerjee, Somak Aditya, Chitta Baral, et al. 2023. Towards logiglue: A brief survey and a benchmark for analyzing logical reasoning capabilities of language models. *arXiv preprint arXiv:2310.00836*.

MetaAI. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

OpenAI. 2022. Introducing chatgpt.

OpenAI. 2024. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

Patrick Prosser. 1993. Hybrid algorithms for the constraint satisfaction problem. *Computational intelligence*, 9(3):268–299.

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. 2025. Qwen2.5 technical report. *Preprint*, arXiv:2412.15115.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. 2024. Direct preference optimization: Your language model is secretly a reward model. *Preprint*, arXiv:2305.18290.

Abulhair Saparov and He He. 2022. Language models are greedy reasoners: A systematic formal analysis of chain-of-thought. *arXiv preprint arXiv:2210.01240*.

Koustuv Sinha, Shagun Sodhani, Jin Dong, Joelle Pineau, and William L. Hamilton. 2019. CLUTRR: A diagnostic benchmark for inductive reasoning from text. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 4506–4515, Hong Kong, China. Association for Computational Linguistics.

Zayne Sprague, Xi Ye, Kaj Bostrom, Swarat Chaudhuri, and Greg Durrett. 2024. Musr: Testing the limits of chain-of-thought with multistep soft reasoning. *Preprint*, arXiv:2310.16049.

Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V Le, Ed H Chi, Denny Zhou, et al. 2022. Challenging big-bench tasks and whether chain-of-thought can solve them. *arXiv preprint arXiv:2210.09261*.

Jidong Tian, Yitian Li, Wenqing Chen, Liqiang Xiao, Hao He, and Yaohui Jin. 2021. Diagnosing the first-order logical reasoning ability through logicnli. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 3738–3747.

Xingyao Wang, Hao Peng, Reyhaneh Jabbarvand, and Heng Ji. 2024a. Leti: Learning to generate from textual interactions. *Preprint*, arXiv:2305.10314.

Xingyao Wang, Zihan Wang, Jiateng Liu, Yangyi Chen, Lifan Yuan, Hao Peng, and Heng Ji. 2024b. Mint: Evaluating llms in multi-turn interaction with tools and language feedback. *Preprint*, arXiv:2309.10691.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Colin White, Samuel Dooley, Manley Roberts, Arka Pal, Ben Feuer, Siddhartha Jain, Ravid Shwartz-Ziv, Neel Jain, Khalid Saifullah, Siddartha Naidu, et al. 2024. Livebench: A challenging, contamination-free llm benchmark. *arXiv preprint arXiv:2406.19314*.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2024. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36.

Zheng Yuan, Hongyi Yuan, Chengpeng Li, Guanting Dong, Keming Lu, Chuanqi Tan, Chang Zhou, and Jingren Zhou. 2023. Scaling relationship on learning mathematical reasoning with large language models. *Preprint*, arXiv:2308.01825.

Wanjun Zhong, Siyuan Wang, Duyu Tang, Zenan Xu, Daya Guo, Jiahai Wang, Jian Yin, Ming Zhou, and Nan Duan. 2021. Ar-lsat: Investigating analytical reasoning of text. *arXiv preprint arXiv:2104.06598*.

Kaijie Zhu, Jindong Wang, Qinlin Zhao, Ruochen Xu, and Xing Xie. 2024. Dynamic evaluation of large language models by meta probing agents. In *Forty-first International Conference on Machine Learning*.

A Appendix A

A.1 Difficulty Control through Data Augmentation

The experimental results on the English subset are presented in Figure 6. Our analysis reveals a clear negative correlation between model performance and two key difficulty metrics: the number of constraints and the solution space size. Specifically, model accuracy demonstrates a consistent decline as the number of constraints increases and as the solution space becomes more constrained. These findings align closely with our previous observations on the Chinese dataset, reinforcing the language-agnostic nature of these performance patterns.

The augmented dataset achieves a more balanced distribution across these difficulty indicators, enabling a more nuanced and comprehensive evaluation of model capabilities. Through systematic manipulation of constraint density and solution space, AutoLogi demonstrates its effectiveness in generating targeted evaluation scenarios with controlled difficulty levels.

A.2 Error Analysis

To gain insights into improving model performance in logical reasoning tasks, we conducted a systematic analysis of incorrect responses. We examined 90 samples, one from each background category in AutoLogi's Chinese dataset, focusing on Claude 3.5 Sonnet's responses. Among these samples, the model failed on 37 cases. We categorized these errors into four distinct types:

1. **Unanalyzed Condition**: Cases where the model completely overlooked critical conditions in the problem statement.

2. **Incorrect Logical Inference**: Instances where the model made erroneous deductions when processing specific logical conditions.

3. **Contradictory Conclusion**: Cases where despite appearing to follow a sound reasoning process, the model's final conclusion contradicted its own analysis.

4. **Inconsistent Format**: Responses that deviated from the required output format, including issues with naming conventions and data structure consistency.

Detailed examples for each error category are provided in Appendix A.3. The distribution of these errors across the 37 failed cases is illustrated in Figure 7. Our analysis reveals that Claude 3.5 Sonnet demonstrates strong instruction-following capabilities and processes constraints systematically. Consequently, Inconsistent Format and Unanalyzed Condition errors were relatively rare. The most prevalent issue was **Incorrect Logical Inference**.

Further investigation into the Incorrect Logical Inference cases revealed two primary challenges:

1. **Complex Logic**: Errors emerged when handling logical conditions that required multiple interconnected judgments.

2. **Misinterpretation of Logical Negations**: Common mistakes included misunderstanding logical operators, particularly in cases involving negations (e.g., confusing "not less than" with "less than").

A.3 Error Cases

Following the analysis of four distinct error types and their distribution in Section A.2, here we present one representative example for each type from the AutoLogi Chinese dataset. These examples also serve as sample problems demonstrating how the system processes logical reasoning questions. While the AutoLogi dataset contains both Chinese and English problems, we focus on Chinese examples here since English problems typically have much longer problem descriptions. For each example, we provide both the original Chinese text and its English translation.

The following examples illustrate: Unanalyzed Condition (Table 5), Incorrect Logical Inference (Table 6), Contradictory Conclusion (Table 7), and Inconsistent Format (Table 8).

A.4 Prompt Design

A.4.1 Stage 1: Puzzle Formulation

The first stage focuses on extracting two essential components from the source corpus for open-ended logic puzzle generation: Background information and Logical Constraints. To automate this extraction process, we employ GPT-4o with a one-shot learning approach, providing a single comprehensive example. Table 9 details the prompt instructions in both Chinese and English.

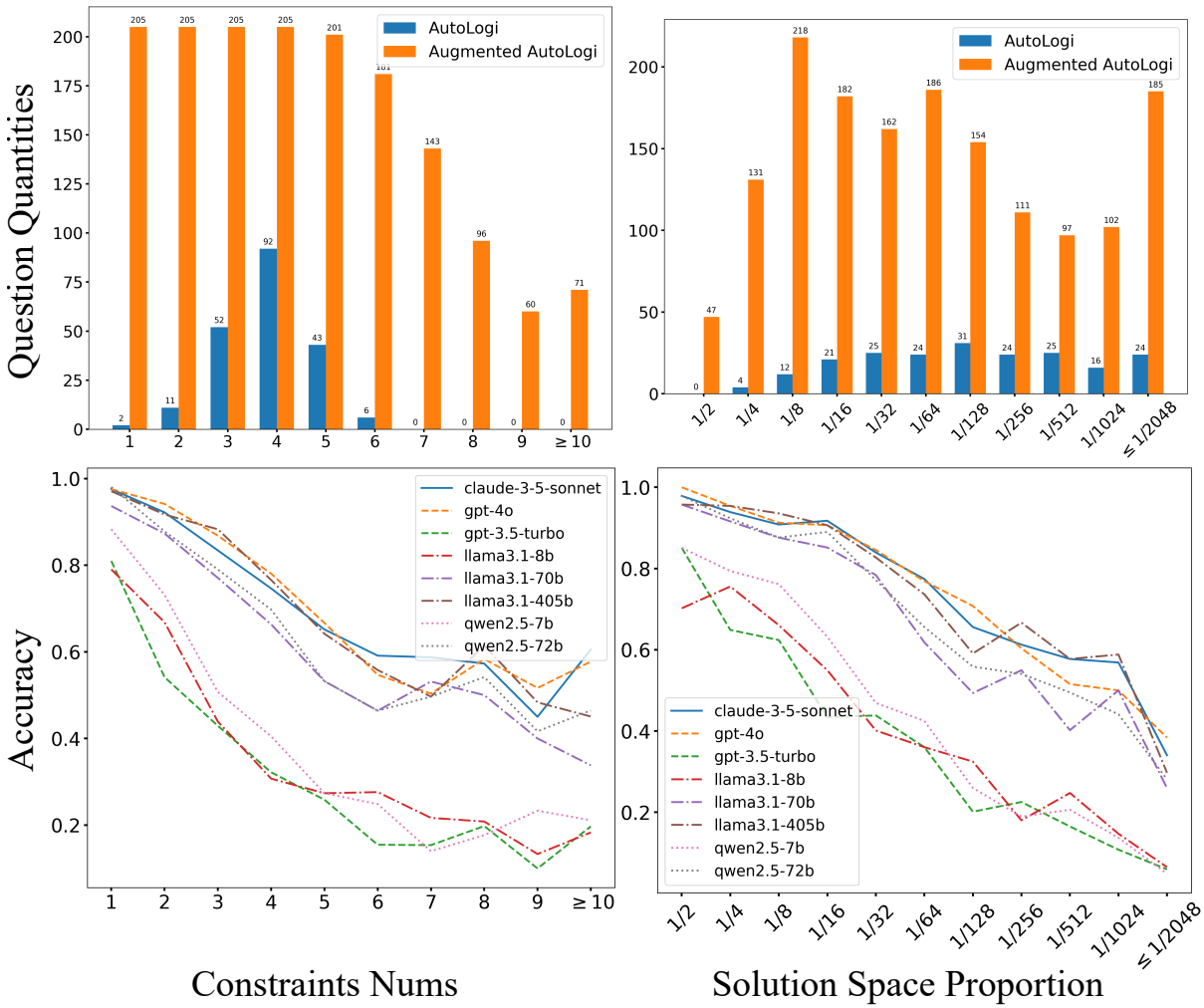

Figure 6: Distribution of questions in AutoLogi's English subset before and after data augmentation, alongside the performance analysis of eight models across varying constraints and solution space proportions.

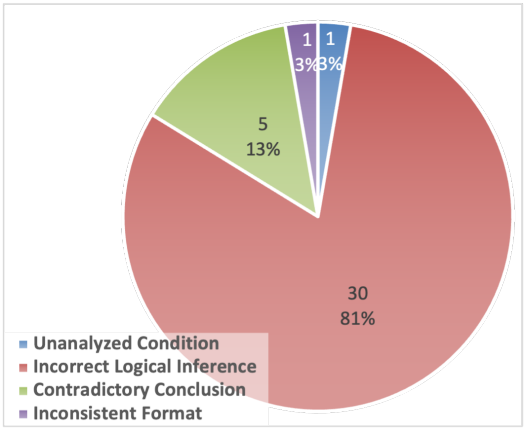

Figure 7: Distribution of error types in model responses, analyzing 37 incorrect answers from Claude across 90 test questions.

A.5 Stage 2: Generation of Format and Verifiers

This stage utilizes prompts to instruct GPT-4o to generate two key components based on the previously extracted Background information and Logical Constraints. The first component is the synthesized puzzle, consisting of the Arrangement Format and an Arrangement Example. The second component comprises two code-based verifiers: a Format Verifier to ensure structural correctness and a Constraint Verifier to validate logical consistency. The detailed prompts in both Chinese and English are presented in Figures 8, 9, 10, 11, 12, and 13.

A.6 Stage 2: Validation and Regeneration

A crucial verification step involves the implementation of a Traversal Function to validate the correctness of the generated puzzles. This function employs an enumeration approach to verify the

PROMPT (CHINESE): 有7名运动员参加男子5千米的决赛，他们是:S、T、U、W、X、Y和Z。运动员穿的服装不是红色，就是绿色，没有运动员同时到达终点。请你生成一个安排方案满足以下约束： (1) 相继到达终点的运动员，他们的服装不全是红色的。 (2) Y在T和W之前的某一时刻到达了终点。 (3) 在Y之前到达终点的运动员，恰好有两位穿的是红色服装。 (4) S是第六个到达终点的运动员。 (5) Z在U之前的某一时刻到达终点。
请你一步一步思考，你的安排方案必须按照为以下输入格式要求来进行回答：在回答的最后，你需要给出一个输入，以表示你的最终安排方案, 其中inputs 是一个字典，包含两个键："order" 和"colors"。- inputs["order"] 是一个包含7个元素的list，元素为运动员的名字（字符串），表示运动员的到达顺序。所有可能的元素为["S", "T", "U", "W", "X", "Y", "Z"]。- inputs["colors"] 是一个字典，键为运动员的名字（字符串），值为"red" 或"green"（字符串），表示运动员的衣服颜色。下面提供一个输入样例（注意这只是一个合法输入的例子，不一定是正确方案）：
{ "order": ["T", "Y", "Z", "U", "W", "S", "X"], "colors": {S": "red", "T": "green", "U": "red", "W": "green", "X": "red", "Y": "green", "Z": "green"} }

PROMPT (ENGLISH,TRANSLATED): There are 7 athletes participating in the men's 5-kilometer final, and they are: S, T, U, W, X, Y, and Z. The athletes' outfits are either red or green, and no athletes reached the finish line simultaneously. Please generate an arrangement that meets the following constraints:
(1) The athletes who reached the finish line consecutively do not all wear red outfits. (2) Y reached the finish line at some point before T and W. (3) Among the athletes who reached the finish line before Y, exactly two wore red outfits. (4) S is the sixth athlete to reach the finish line. (5) Z reached the finish line at some point before U.
Please think step by step, and your arrangement must be answered according to the following input format requirements: At the end of your answer, you need to provide an input to represent your final arrangement. The inputs should be a dictionary containing two keys: "order" and "colors". - ‘inputs["order"]‘ is a list containing 7 elements, with each element being the name of an athlete (string) representing the order in which the athletes reached the finish line. Possible elements are ‘["S", "T", "U", "W", "X", "Y", "Z"]‘. - ‘inputs["colors"]‘ is a dictionary where the key is the name of an athlete (string) and the value is "red" or "green" (string), representing the color of the athlete's outfit. Below is an example input (note this is just an example of valid input, not necessarily the correct arrangement):
{ "order": ["T", "Y", "Z", "U", "W", "S", "X"], "colors": {S": "red", "T": "green", "U": "red", "W": "green", "X": "red", "Y": "green", "Z": "green"} }

MODEL RESPONSE (CHINESE): 好的,让我们一步步思考来满足所有约束条件: 1. 首先,我们知道S是第六个到达终点的,所以我们可以先把S放在第六位。 2. Y在T和W之前到达终点。 （这里进行了简单重复没有分析） 3. Z在U之前到达终点,所以Z必须在U前面。 4. 在Y之前到达终点的运动员,恰好有两位穿的是红色服装。这意味着Y之前的运动员中,必须有两个红色,一个绿色。 5. 相继到达终点的运动员,他们的服装不全是红色的。这意味着不能有两个连续的红色。
根据这些条件,我们可以尝试以下安排: 1. X (红色) 2. T (绿色) 3. Y (绿色) 4. Z (红色) 5. W (绿色) 6. S (红色) 7. U (绿色)
这个安排满足所有条件: - S在第六位 - Y在T和W之前 - Z在U之前 - Y之前有两个红色(X和T) - 没有连续的红色 - 总共有3个红色,4个绿色。因此,最终的输入格式如下:

```
{"order": ["X", "T", "Y", "Z", "W", "S", "U"],
"colors": {S": "red","T": "green","U": "green","W": "green","X": "red","Y":"green","Z": "red"}}
```

MODEL RESPONSE(ENGLISH,TRANSLATED): Let's think step by step to meet all the constraints: 1. First, we know S is the sixth to reach the finish line 2. Y reached the finish line before T and W (This was merely repeated without proper analysis) 3. Z reached the finish line before U 4. Among the athletes before Y, exactly two wore red outfits 5. No consecutive athletes can wear red
Based on these conditions, we can arrange as follows: 1. X (red) 2. T (green) 3. Y (green) 4. Z (red) 5. W (green) 6. S (red) 7. U (green)
Final solution in required format:

```
{"order": ["X", "T", "Y", "Z", "W", "S", "U"],
"colors": {"S": "red", "T": "green", "U": "green","W": "green", "X": "red", "Y": "green","Z": "red"} }
```

Error Type: Unanalyzed Condition

There was no analysis of the constraint "(2) Y reached the finish line at some point before T and W".

Table 5: Error Case: Unanalyzed Condition

existence of solutions that satisfy both verifiers established in the previous stage. By exhaustively exploring the value ranges of puzzle parameters, the Traversal Function helps ensure the quality and solvability of the synthesized puzzles. We utilize a two-shot prompting approach to guide GPT-4o in generating this function based on the value domains specified in the puzzle. The detailed prompts in both English and Chinese can be found in Figures 14, 15, 16, 17, 18, and 19.

A.7 Stage 3: Data Augmentation

The third stage implements data augmentation through two approaches: Reduction and Expansion. In the Expansion process, we leverage GPT-4o to generate additional constraints by learning from

PROMPT (CHINESE): 某国东部沿海有5个火山岛E、F、G、H、I，它们由北至南排成一条直线。 请你生成一个安排方案满足以下约束： (1) F与H相邻并且在H的北边 (2) I和E相邻 (3) G在F的北边某个位置 (4) G和E相邻
请你一步一步思考，你的安排方案必须按照为以下输入格式要求来进行回答：在回答的最后，你需要给出一个输入，以表示你的最终安排方案, 其中inputs 是一个包含5个元素的list，元素为字符串，取值范围是["E","F","G","H","I"]。下面提供一个输入样例（注意这只是一个合法输入的例子，不一定是正确方案）：['F', 'H', 'I', 'E', 'G']

PROMPT (ENGLISH,TRANSLATED): There are 5 volcanic islands, E, F, G, H, and I, located along the eastern coast of a country, arranged in a straight line from north to south. Please generate an arrangement that satisfies the following constraints: 1. F is adjacent to H and is located north of H. 2. I is adjacent to E. 3. G is located somewhere north of F. 4. G is adjacent to E. Please think step by step, and your arrangement must follow the input format requirements below: At the end of your answer, you need to provide an input to represent your final arrangement. The input should be a list containing 5 elements, with each element being a string from the range ‘["E", "F", "G", "H", "I"]‘. Below is an example input (note that this is just an example of valid input and may not be the correct arrangement): ['F', 'H', 'I', 'E', 'G']

MODEL RESPONSE (CHINESE): 让我们一步步思考这个问题： 1. 根据约束(1)，F与H相邻并且在H的北边。所以我们知道F和H必须相邻，且F在H的北边。 2. 根据约束(3)，G在F的北边某个位置。这意味着G必须在F的北边，但不一定紧邻F。 3. 根据约束(4)，G和E相邻。结合约束(3)，我们可以推断E必须在G的北边，因为G已经在F的北边了。(推理错误：G不一定在F北边) 4. 根据约束(2)，I和E相邻。考虑到前面的推论，I只能在E的北边或者H的南边。 5. 综合以上分析，我们可以得出一个可能的排列：G-E-I-F-H 或I-G-E-F-H。 6. 考虑到题目要求从北到南排列，I-G-E-F-H 是符合所有约束的一个有效排列。因此，最终的安排方案如下：

```
['I', 'G', 'E', 'F', 'H']
```

这个排列满足所有给定的约束条件：- F与H相邻并且在H的北边- I和E相邻- G在F的北边- G和E相邻

MODEL RESPONSE (ENGLISH,TRANSLATED): Let’s think through this problem step by step: 1. According to constraint (1), F is adjacent to H and is north of H. So, we know F and H must be adjacent, with F to the north of H. 2. According to constraint (3), G is located somewhere north of F. This means G must be north of F, but not necessarily immediately adjacent. 3. According to constraint (4), G is adjacent to E. Combining this with constraint (3), we can infer that E must be north of G because G is already north of F.(Logical error: G is not necessarily north of F) 4. According to constraint (2), I is adjacent to E. Considering the previous inferences, I can only be either north of E or south of H. 5. Combining all the above analysis, one possible arrangement is G-E-I-F-H or I-G-E-F-H. 6. Considering the requirement that the arrangement is from north to south, I-G-E-F-H is a valid arrangement that satisfies all constraints. Therefore, the final arrangement is as follows:

```
['I', 'G', 'E', 'F', 'H']
```

This arrangement satisfies all the given constraints: - F is adjacent to H and is north of H. - I is adjacent to E. - G is north of F. - G is adjacent to E

Error Type: Incorrect Logical Inference

Error in step 3 of the reasoning: The conclusion that "E must be north of G" cannot be derived from the premises "G is north of F" and "G is adjacent to E"

Table 6: Error Case: Incorrect Logical Inference

existing ones and their known solution sets. The model analyzes the patterns in current constraints and their corresponding solutions to synthesize new, compatible constraints. To ensure the validity of the augmented puzzles, we employ the Traversal Function for verification, confirming that each expanded puzzle maintains solvability. The complete prompts for this stage are presented in Figures 20, 21, 22, 23, 24, and 25.

A.8 Detailed Training Results

The experimental results presented in Table 10 report the accuracy scores with their corresponding standard deviations (std), calculated across five independent trials.

PROMPT (CHINESE): G、H、J、K、L、M和O七人结伴出游。在他们4天的露营生活中，每天要安排两个人值班为大家做饭。要求这4天的人员安排不能重复，且恰好有一人值班两天。请你生成一个安排方案满足以下约束：(1) J要排在H值班的次日。(2) 值班两天的那个人必须排在第四天，不能排在第三天。(3) G必须与J和O中的某一个人排在同一天。(4) 第一天或第二天至少有一天安排K。(5) O必须排在第三天。
请你一步一步思考，你的安排方案必须按照为以下输入格式要求来进行回答：在回答的最后，你需要给出一个输入，以表示你的最终安排方案，其中inputs是一个字典，包含四个键：‘"day1"‘，‘"day2"‘，‘"day3"‘和‘"day4"‘。‘inputs["day1"]‘，‘inputs["day2"]‘，‘inputs["day3"]‘，‘inputs["day4"]‘都是一个包含两个元素的set，包含值班人员名字（字符型）。取值范围是['G', 'H', 'J', 'K', 'L', 'M', 'O']。下面提供一个输入样例（注意这只是一个合法输入的例子，不一定是正确方案）：{'day1': {'K', 'G'}, 'day2': {'K', 'H'}, 'day3': {'O', 'L'}, 'day4': {'J', 'M'}}

PROMPT (ENGLISH,TRANSLATED): Seven people, G, H, J, K, L, M, and O, go on a trip together. During their 4-day camping, two people are assigned each day to cook for the group. The assignment must not be repeated over the 4 days, and exactly one person must be on duty for two days. Please generate an arrangement that satisfies the following constraints: 1. J must be on duty the day after H. 2. The person who is on duty for two days must be assigned on the fourth day and cannot be assigned on the third day. 3. G must be on the same day as either J or O. 4. K must be scheduled on either the first or the second day. 5. O must be scheduled on the third day.
Please think step by step, and your arrangement must follow the input format requirements below: At the end of your answer, you need to provide an input to represent your final arrangement. The input should be a dictionary containing four keys: "day1", "day2", "day3", and "day4". ‘inputs["day1"]‘, ‘inputs["day2"]‘, ‘inputs["day3"]‘, and ‘inputs["day4"]‘ are each *sets* containing two elements representing the names of the people on duty (*strings*). The range of values is ‘['G', 'H', 'J', 'K', 'L', 'M', 'O']‘. Below is an example input (note that this is just an example of valid input and may not be the correct arrangement): {'day1': {'K', 'G'}, 'day2': {'K', 'H'}, 'day3': {'O', 'L'}, 'day4': {'J', 'M'}}

MODEL RESPONSE (CHINESE): 好的，我会一步步思考来制定一个满足所有约束条件的安排方案。 1. O必须排在第三天，所以先确定： - inputs["day3"] = 'O', 某人 2. J要排在H值班的次日，所以H必须在第1或第2天，J在第2或第3天。但J不能和O同天，所以： - inputs["day2"] = 'H', 某人 - inputs["day3"] = 'O', 'J' (这里要求'J'安排在第三天，与最终给出的结论是相互矛盾的) 3. 值班两天的人必须在第四天，不能在第三天。G必须和J或O同一天，但G不能是值班两天的人(因为G不在第四天)，所以G必须和O同一天： - inputs["day3"] = 'O', 'G' - inputs["day4"] = 'J', 值班两天的人 4. 第一天或第二天至少有一天安排K。我们可以把K安排在第一天： - inputs["day1"] = 'K', 某人 5. 剩下L和M还没安排。L可以安排在第一天，M可以安排在第二天： - inputs["day1"] = 'K', 'L' - inputs["day2"] = 'H', 'M' 6. 现在，值班两天的人只能是M，因为其他人都只值班了一次。综上所述，最终的安排方案是：

```
{'day1': {'K', 'L'}, 'day2': {'H', 'M'}, 'day3': {'O', 'G'}, 'day4': {'J', 'M'}}
```

这个方案满足了所有给定的约束条件。

MODEL RESPONSE (ENGLISH,TRANSLATED): Let's think step by step to create an arrangement that satisfies all the constraints. 1. O must be scheduled on the third day, so first we set: - inputs["day3"] = 'O', someone 2. J must be on duty the day after H, so H must be on the 1st or 2nd day, and J on the 2nd or 3rd day. But J cannot be on the same day as O, so: - inputs["day2"] = 'H', someone - inputs["day3"] = 'O', 'J' (There's a contradiction between requiring 'J' to be on day 3 here and the final arrangement presented) 3. The person on duty for two days must be on the fourth day, and not on the third day. G must be on the same day as either J or O, but G cannot be the person on duty for two days (because G is not on the fourth day). So G must be with O: - inputs["day3"] = 'O', 'G' - inputs["day4"] = 'J', the person on duty for two days 4. K must be scheduled on either the first or second day. We can place K on the first day: - inputs["day1"] = 'K', someone 5. L and M are left. L can be scheduled on the first day, and M on the second day: - inputs["day1"] = 'K', 'L' - inputs["day2"] = 'H', 'M' 6. Now, the person on duty for two days can only be M, because everyone else is only on duty once. Therefore, the final arrangement is:

```
{'day1': {'K', 'L'}, 'day2': {'H', 'M'}, 'day3': {'O', 'G'}, 'day4': {'J', 'M'}}
```

This arrangement satisfies all the given constraints.

Error Type: Contradictory Conclusion

There's a contradiction: While step 2 of the reasoning process states that 'J' must be on day 3, the final solution puts it on day 4!

Table 7: Error Case: Contradictory Conclusion

PROMPT (CHINESE): 一个委员会工作两年，每年都由4 人组成，其中2 名成员来自下面4 位法官：F、G、H 和I，另外2 名成员来自下面3 位科学家：V、Y 和Z。每一年，该委员会有1 名成员做主席。 请你生成一个安排方案满足以下约束： 1. 在第一年做主席的成员在第二年必须退出该委员会。在第二年做主席的人在第一年必须是该委员会的成员。 2. G 和V 不能在同一年成为该委员会的成员。 3. H 和Y 不能在同一年成为该委员会的成员。 4. 每一年，I 和V 中有且只有一位做该委员会的成员。
请你一步一步思考，你的安排方案必须按照为以下输入格式要求来进行回答：输入是一个字典，包含四个键："committee_year1"、"chairman_year1"、"committee_year2" 和"chairman_year2"。- 在回答的最后，你需要给出一个输入，以表示你的最终安排方案, 其中‘inputs["committee_year1"]‘ 是一个包含4 个元素的set，包含第一年的委员会成员的名字。取值范围是‘["F","G","H","I","V","Y","Z"]‘。- ‘inputs["chairman_year1"]‘ 是一个字符串，是第一年的主席名字，必须是第一年的委员。- ‘inputs["committee_year2"]‘ 是一个包含4 个元素的set，包含第二年的委员会成员的名字。取值范围是‘["F","G","H","I","V","Y","Z"]‘。- ‘inputs["chairman_year2"]‘ 是一个字符串，第二年的主席名字必须是第二年的委员，并且是第一年的委员。下面提供一个输入样例（注意这只是一个合法输入的例子，不一定是正确方案）：{’committee_year1’: {’Z’, ’V’, ’F’, ’G’}, ’chairman_year1’: ’G’, ’committee_year2’: {’Y’, ’V’, ’F’, ’H’}, ’chairman_year2’: ’F’}

PROMPT (ENGLISH): A committee works for two years, and each year it consists of 4 members, including 2 judges from the following: F, G, H, and I, and 2 scientists from the following: V, Y, and Z. Each year, the committee has one member serve as chairman. Please generate an arrangement that satisfies the following constraints: 1. The member who serves as chairman in the first year must leave the committee in the second year. The member who serves as chairman in the second year must be a member of the committee in the first year. 2. G and V cannot be members of the committee in the same year. 3. H and Y cannot be members of the committee in the same year. 4. Each year, either I or V, but not both, must be a member of the committee.
Please think step by step, and your arrangement must follow the input format requirements below: The input is a dictionary containing four keys: "committee_year1", "chairman_year1", "committee_year2", and "chairman_year2". - At the end of your answer, you need to provide an input representing your final arrangement, where inputs["committee_year1"] is a set containing 4 elements, representing the members of the committee in the first year. The range of values is ["F","G","H","I","V","Y","Z"]. - inputs["chairman_year1"] is a string representing the chairman of the first year, and this person must be a committee member in the first year. - inputs["committee_year2"] is a set containing 4 elements, representing the members of the committee in the second year. The range of values is ["F","G","H","I","V","Y","Z"]. - inputs["chairman_year2"] is a string representing the chairman of the second year, and this person must be a committee member in both the current and previous year. Below is an example input (note that this is just an example of valid input and may not be the correct arrangement): {’committee_year1’: {’Z’, ’V’, ’F’, ’G’}, ’chairman_year1’: ’G’, ’committee_year2’: {’Y’, ’V’, ’F’, ’H’}, ’chairman_year2’: ’F’}

MODEL RESPONSE (CHINESE): 好的, 我会一步步思考来制定一个满足所有约束条件的安排方案。 第一步：考虑第一年的委员会成员 - 需要2 名法官和2 名科学家 - G 和V 不能同时在委员会中 - H 和Y 不能同时在委员会中 - I 和V 中必须有且只有一位 考虑到这些约束，我们可以选择： - F, I (法官) - V, Z (科学家) 第二步：选择第一年的主席 从F, I, V, Z 中选择一位作为主席。我们选择F 作为第一年的主席。 第三步：考虑第二年的委员会成员 - 第一年的主席F 必须退出 - 需要包含第二年主席的人选 - 仍然需要遵守其他约束条件 考虑到这些, 我们可以选择： - G, H (法官) - Y, Z (科学家) 第四步：选择第二年的主席 从G, H, Y, Z 中选择一位作为主席，且这个人必须是第一年的委员。唯一符合条件的是Z。最终的安排方案如下：

```python
committee_year1 = {"F", "I", "V", "Z"}
chairman_year1 = "F"
committee_year2 = {"G", "H", "Y", "Z"}
chairman_year2 = "Z"
```

(不符合格式要求)

MODEL RESPONSE (ENGLISH): Let’s think step by step to create an arrangement that satisfies all the constraints.
Step 1: Consider the members of the committee for the first year - We need 2 judges and 2 scientists - G and V cannot both be in the committee - H and Y cannot both be in the committee - One of I or V must be in the committee, but not both Considering these constraints, we can select: - Judges: F, I - Scientists: V, Z
Step 2: Select the chairman for the first year From F, I, V, Z, select one as chairman. We select F as the chairman for the first year.
Step 3: Consider the members of the committee for the second year - The first year’s chairman F must leave the committee - The second year’s chairman must be included - Still need to adhere to the other constraints Considering these, we can select: - Judges: G, H - Scientists: Y, Z
Step 4: Select the chairman for the second year From G, H, Y, Z, select one as chairman, and this person must have been a member in the first year. The only eligible person is Z. The final arrangement is as follows:

```python
committee_year1 = {"F", "I", "V", "Z"}
chairman_year1 = "F"
committee_year2 = {"G", "H", "Y", "Z"}
chairman_year2 = "Z"
```

(Does not meet the format requirements)

Error Type: Inconsistent Format

The answer uses Python assignment syntax, but a dictionary definition is required.

Table 8: Error Case: Inconsistent Format

INSTRUCTION(ENGLISH): Provide a description of a problem, which contains some background and definition of the problem (usually the first one or two sentences), as well as some logical constraints. You need to divide these two parts, you can reorganize the language, but do not change the original meaning of the problem.

INPUT: "A supermarket is neatly arranged with 7 rows of shelves from front to back, displaying 7 categories of goods: stationery, snacks, condiments, daily necessities, alcohol, grains and oils, and beverages, each occupying one row. It is known that: \nThe alcohol is placed before the condiments; \nThere are 3 rows between the stationery and the condiments; \nThe grains and oils are after the snacks, with 2 rows in between; \nThe daily necessities are either in the row before or after the stationery; \nThe snacks are in the first row."

OUTPUT:

```json
{
   "background": "A supermarket is neatly arranged with 7 rows of shelves from front to back, displaying 7 categories of goods: stationery, snacks, condiments, daily necessities, alcohol, grains and oils, and beverages, each occupying one row.",
    "logic_constraints": "The alcohol is placed before the condiments; There are 3 rows between the stationery and the condiments; The grains and oils are after the snacks, with 2 rows in between; The daily necessities are either in the row before or after the stationery; The snacks are in the first row."
}
```

INPUT: [[[question]]]

OUTPUT:

INSTRUCTION(CHINESE): 提供一个问题的描述，这个描述中有一些是问题的背景以及定义(一般就是开头一两句)，也有一些是逻辑的限制条件，你需要把这两部分划分出来，可以自己重新组织语言，但不要改变原题意。

INPUT: "某超市从前到后整齐排列着7排货架，放置着文具、零食、调料、日用品、酒、粮油和饮料7类商品，每类商品\n占据一排。已知：\n酒类排在调料类之前；\n文具类和调料类中间隔着3排；\n粮油类在零食类之后，中间隔着2排；\n日用品类紧挨在文具类前一排或者后一排。\n零食类排在第1排"

OUTPUT:

```json
{
    "background":"某超市从前到后整齐排列着7排货架，放置着文具、零食、调料、日用品、酒、粮油和饮料7类商品，每类商品占据一排。",
    "logic_constraints": "酒类排在调料类之前；文具类和调料类中间隔着3排；粮油类在零食类之后，中间隔着2排；日用品类紧挨在文具类前一排或者后一排；零食类排在第1排。"
}
```

INPUT: [[[question]]]

OUTPUT:

Table 9: Instructions for Puzzle Formulation

Model	AutoLogi		AR-LSAT	LogiQA	MUSR	LiveBench
	EN	CN				
Baseline Qwen2.5-7b-instruct	43.64±1.25	42.08±1.50	22.70±0.84	34.42±1.38	47.14±0.73	30.67†
+*Self-Alignment DPO*	48.80±0.91	45.39±0.41	26.09±2.37	38.05±4.00	47.57±0.30	35.73±2.94
+*Strong-to-Weak RFT*	48.33±0.63	47.54±0.71	27.30±0.97	35.93±2.59	49.15±0.55	30.07±2.00
Baseline Qwen2.5-72b-instruct	68.18±0.77	63.92±0.56	31.65±1.89	47.92±1.37	54.21±0.41	46.00†
+*Self-Alignment DPO*	74.79±0.41	69.54±0.87	38.70±1.98	48.14±3.98	56.48±0.47	52.13±1.48

Table 10: Results on various benchmarks. Numbers in parentheses indicate standard deviation over 5 trials. †Results reported from LiveBench leaderboard(2024-08-31).

The Input describes a problem in a real-life scenario along with some logical constraints. There may be several solutions that satisfy all the above constraints. The goal is to determine whether a particular solution meets all the conditions of the problem.

The approach is to first analyze, then define the inputs (a dict) to represent any possible solution to be verified based on the background and definitions. Therefore, you need to provide the specification of the inputs and an example. Note that the definition of the inputs should be based on the background and definitions, not on the logical constraints. Do not include the logical constraints in the specification! Also, if the order of the elements does not affect the result, prefer to use a set, but if the elements may be repeated or the order affects the result, prefer to use a list. Then write an 'inputs_check' function to verify the validity of the inputs; then extract all the logical constraints of the problem and convert them into corresponding constraint functions. You need to provide a constraint_list, which contains all the extracted constraint functions, to ensure that it does not repeat with the judgment of inputs_check; finally, form the final verify_function. This function is as follows:

```
def verify_function(inputs, inputs_check, constraint_list):
    # First check if the inputs are valid
    if not inputs_check(inputs):  # If the input format is not satisfied
            return False

    # Traverse the constraint_list and check each constraint function
    for constraint in constraint_list:
        if not constraint(inputs):  # If the constraint is not satisfied
            return False

    # All constraints have been checked and satisfied
    return True
```

Here is an example:

Input:

Background and definition: A class plans to select two girls from Emily, Sophia, and Olivia, and three boys from James, Michael, William, David, and Robert to form a five-person college student teaching team for voluntary teaching in the mountains.

Figure 8: Stage 2 prompt for English Data(1/3). This prompt is designed to simultaneously generate both puzzle requirements (Arrangement Format + Arrangement Example) and program-based verifiers for answer validation, using a one-shot demonstration approach.

Logical constraints: (1) Sophia and Robert cannot be selected at the same time; (2) James and Song Kaiwen cannot be selected at the same time; (3) Michael and Robert cannot be selected at the same time.

Output:

Analysis

First, we need to define the format of inputs based on the content of the background and definitions, and it cannot include logical constraints. We can design inputs as a dictionary, which includes two keys: "female" and "male", corresponding to the selected girls and boys list. Check it, it does not include the content of logical constraints! Elements are not repeated and the order does not matter, so use a set.

Inputs_Format

Inputs is a dictionary, containing two keys: "female" and "male".

inputs["female"] is a set containing 2 elements, including the names of the selected girls (strings). The value range is ["Emily", "Sophia", "Olivia"].

inputs["male"] is a set containing 3 elements, including the names of the selected boys (strings). The value range is ["James", "Michael", "William", "David", "Robert"].

Inputs_Example

```
{"female": {"Emily", "Sophia"},"male": {"James", "Michael", "William"}}
```

Inputs_Check_Function

```
def inputs_check(inputs):
    # Check if inputs is a dictionary
    if not isinstance(inputs, dict):
        return False
    # Check if inputs contains "female" and "male" two keys
    if "female" not in inputs or "male" not in inputs:
        return False
    # Check if the values corresponding to "female" and "male" are sets
    if not isinstance(inputs["female"], set) or not
isinstance(inputs["male"], set):
        return False
    # Check if the sets corresponding to "female" and "male" contain the
correct number of elements
```

Figure 9: Stage 2 prompt for English Data(2/3). This prompt is designed to simultaneously generate both puzzle requirements (Arrangement Format + Arrangement Example) and program-based verifiers for answer validation, using a one-shot demonstration approach.

```
    if len(inputs["female"]) != 2 or len(inputs["male"]) != 3:
        return False
    # Check if the elements of the sets corresponding to "female" and "male" are correct
    if not inputs["female"].issubset({"Emily", "Sophia", "Olivia"}) or not inputs["male"].issubset({"James", "Michael", "William", "David", "Robert"}):
        return False
    return True
```

Constraint_List

Create three constraint functions below to meet the requirements in the problem description:

Sophia and Robert cannot be selected at the same time;

James and David cannot be selected at the same time;

Michael and Robert cannot be selected at the same time.

```
def constraint_1(inputs):
    if "Sophia" in inputs["female"] and "Robert" in inputs["male"]:
        return False
    return True

def constraint_2(inputs):
    if "James" in inputs["male"] and "David" in inputs["male"]:
        return False
    return True

def constraint_3(inputs):
    if "Michael" in inputs["male"] and "Robert" in inputs["male"]:
        return False
    return True

constraint_list = [constraint_1, constraint_2, constraint_3]
```

Now it's your turn:

Input:

Background and Definitions: [[[background]]]

Logical Constraints: [[[logic_constraints]]]

Output:

Figure 10: Stage 2 prompt for English Data(3/3). This prompt is designed to simultaneously generate both puzzle requirements (Arrangement Format + Arrangement Example) and program-based verifiers for answer validation, using a one-shot demonstration approach.

Input中描述了一个实际场景中的问题以及一些逻辑限制，可能会存在若干满足所有上述限制的方案。目标是判断某一个方案是否满足题意的所有条件。

做法是首先进行分析，然后根据背景和定义的内容来规定用inputs（一个dict）来代表任一可能的待验证方案，因此你需要给出inputs的规范，和一个示例，注意，这个inputs的定义应该基于背景和定义，而不是基于逻辑限制条件，不要把逻辑限制条件加入规范！此外如果元素顺序不影响结果优先使用set，但如果元素可能重复或者顺序影响结果优先使用list。；接着写一个'inputs_check'函数来验证inputs的合法性；然后提取出问题的所有逻辑限制转换成对应的约束函数，你需要给出一个constraint_list,这个列表中包含提取出的所有约束函数，确保不会和inputs_check的判断重复；最后组成最后的verify_fuction,这个函数完成如下：

```
def verify_function(inputs, inputs_check, constraint_list):
# 首先检查inputs是否有效
  if not inputs_check(inputs):  # 如果输入格式不满足
    return False
# 遍历constraint_list，对每个约束函数进行检查
  for constraint in constraint_list:
    if not constraint(inputs):  # 如果约束不满足
      return False
# 所有约束都已检查且满足
  return True
```

这是一个例子：

Input:

背景和定义：某班打算从方如芬、郭嫣然、何之莲三名女生中选拔两人，从彭友文、裘志节、任向阳、宋文凯、唐晓华五名男生中选拔三人组成大学生五人支教小组到山区义务支教。

逻辑限制条件：（1）郭嫣然和唐晓华不可以同时入选；（2）彭友文和宋凯文不可以同时入选；（3）裘志节和唐晓华不可以同时入选。

Output:

Analysis

首先，我们需要根据背景和定义的内容去定义inputs的格式，并且不能包含逻辑限制条件，我们可以把inputs设计为一个字典，其中包含两个键："female"和"male"，它们分别对应选中的女生和男生名单。检查一下，确实没包含逻辑限制条件的内容！元素不重复且顺序无影响，所以使用set。

Figure 11: Stage 2 prompt for Chinese Data(1/3). This prompt is designed to simultaneously generate both puzzle requirements (Arrangement Format + Arrangement Example) and program-based verifiers for answer validation, using a one-shot demonstration approach.

Inputs_Format

inputs是一个字典，包含两个键："female"和"male"。

inputs["female"]是一个包含2个元素的set，包含选中的女生名字（字符串）。取值范围是["方如芬","郭嫣然","何之莲"]。

inputs["male"]是一个包含3个元素的set，包含选中的男生名字（字符串）。取值范围是["彭友文","裘志节","任向阳","宋文凯","唐晓华"]。

Inputs_Example

```
{ "female": {"方如芬", "郭嫣然"},
  "male": {"彭友文", "裘志节", "任向阳"}}
```

Inputs_Check_Function

```
def inputs_check(inputs):
# 检查inputs是否为字典
  if not isinstance(inputs, dict):
    return False
# 检查inputs是否包含"female"和"male"两个键
  if "female" not in inputs or "male" not in inputs:
    return False
# 检查"female"和"male"对应的值是否为set
  if not isinstance(inputs["female"], set) or not isinstance(inputs["male"],
set):
    return False
# 检查"female"和"male"对应的set是否包含正确数量的元素
  if len(inputs["female"]) != 2 or len(inputs["male"]) != 3:
    return False
# 检查"female"和"male"对应的set的元素是否正确
  if not inputs["female"].issubset({"方如芬", "郭嫣然", "何之莲"}) or not
inputs["male"].issubset({"彭友文", "裘志节", "任向阳", "宋文凯", "唐晓华"}):
    return False
  return True
```

Constraint_List

下面创建三个约束函数，用来满足问题描述中的要求：

郭嫣然和唐晓华不同时入选；

彭友文和宋文凯不同时入选；

裘志节和唐晓华不同时入选。

Figure 12: Stage 2 prompt for Chinese Data(2/3). This prompt is designed to simultaneously generate both puzzle requirements (Arrangement Format + Arrangement Example) and program-based verifiers for answer validation, using a one-shot demonstration approach.

```python
def constraint_1(inputs):
  if "郭嫣然" in inputs["female"] and "唐晓华" in inputs["male"]:
    return False
  return True
def constraint_2(inputs):
  if "彭友文" in inputs["male"] and "宋文凯" in inputs["male"]:
    return False
  return True
def constraint_3(inputs):
  if "裘志节" in inputs["male"] and "唐晓华" in inputs["male"]:
    return False
  return True
constraint_list = [constraint_1, constraint_2, constraint_3]
```

下面你来完成：

Input:

背景和定义：[[[background]]]

逻辑限制条件：[[[logic_constraints]]]

Output:

Figure 13: Stage 2 prompt for Chinese Data(3/3).This prompt is designed to simultaneously generate both puzzle requirements (Arrangement Format + Arrangement Example) and program-based verifiers for answer validation, using a one-shot demonstration approach.

Provide: 1. The background and definition of a problem. 2. A previously defined function named 'verify_function'. The input requirements for this function will be provided (including the range and number of values), as well as an input example. The 'verify_function' can be directly called as 'verify_fuction(inputs, inputs_check, constraint_list)', where 'inputs' represents a solution, and 'inputs_check' and 'constraint_list' have been previously defined and can be directly called. The output of this function is the result of the solution (whether it satisfies all logical constraints). Note that before calling 'verify_fuction(inputs, inputs_check, constraint_list)', check that 'inputs_check(inputs)' is true.

Objective: To obtain the number of solutions to the problem and the size of the domain.

Output requirements: Implement the 'count_valid_arrangements' function, provide Python code, and require traversing the domain (according to the range of input requirements) and calculating the number of times 'verify_fuction' is True. The return value is a tuple, the first element is the number of times 'verify_fuction' is True, and the second element is the number of traversed domains.

Two examples:

Input:

```
{
    "background&definition": "Background and definition: A supermarket has 7 neatly arranged shelves, displaying 7 categories of goods: stationery, snacks, condiments, daily necessities, wine, grain and oil, and beverages, each category occupying one shelf.",
    "inputs_format": 'inputs is a list containing 7 elements, each element is a string, representing the category of goods on a shelf. The range of values is ["stationery","snacks","condiments","daily necessities","wine","grain and oil","beverages"].',
    "inputs_example": ["stationery", "daily necessities", "snacks", "condiments", "wine", "grain and oil", "beverages"],
}
```

Output:

```
from itertools import permutations
```

Figure 14: Traversal Function prompt for English Data(1/3). This prompt is designed to generate traversal functions that enumerate all possible solutions, using a two-shots demonstration approach.

```
2
3  def count_valid_arrangements():
4      goods = ["stationery","snacks","condiments","daily
   necessities","wine","grain and oil","beverages"]
5      all_arrangements = list(permutations(goods, 7))
6      valid_count = 0
7      total_count = 0
8      for arrangement in all_arrangements:
9          if not inputs_check(list(arrangement)):
10             continue
11         if verify_function(list(arrangement), inputs_check, constraint_list):
12             valid_count += 1
13         total_count += 1
14     return valid_count, total_count
```

Input:

```
1  {
2      "background&definition": "There are 7 heart disease patients E, F, G, H,
   I, J, K who need to be assigned to 4 doctors: Dr. Zhang, Dr. Li, Dr. Wang,
   and Dr. Liu for treatment. Each patient can only be treated by one doctor,
   and each doctor can treat up to two patients. Among them, J and K are
   children, and the remaining 5 are adults; E, F, and J are males, and the
   remaining 4 are females.",
3      "input_format": 'inputs is a dictionary containing four keys: "Dr_Zhang",
   "Dr_Li", "Dr_Wang", and "Dr_Liu".    inputs["Dr_Zhang"] is a set containing 0-
   2 elements, including the names of the patients (strings) that Dr. Zhang is
   responsible for. The range of values is ["E","F","G","H","I","J","K"].
   inputs["Dr_Li"] is a set containing 0-2 elements, including the names of the
   patients (strings) that Dr. Li is responsible for. The range of values is
   ["E","F","G","H","I","J","K"]. inputs["Dr_Wang"] is a set containing 0-2
   elements, including the names of the patients (strings) that Dr. Wang is
   responsible for. The range of values is ["E","F","G","H","I","J","K"].
   inputs["Dr_Liu"] is a set containing 0-2 elements, including the names of the
   patients (strings) that Dr. Liu is responsible for. The range of values is
   ["E","F","G","H","I","J","K"].',
4      "example_input": {
5          "Dr_Zhang": {"E", "F"}, "Dr_Li": {"G"},
6          "Dr_Wang": {"H", "J"}, "Dr_Liu": {"I", "K"}
7          },
8  }
```

Output:

Figure 15: Traversal Function prompt for English Data(2/3). This prompt is designed to generate traversal functions that enumerate all possible solutions, using a two-shots demonstration approach.

```python
from itertools import combinations, permutations
def count_valid_arrangements():
    patients = ["E","F","G","H","I","J","K"]
    doctors = ["Dr_Zhang", "Dr_Li", "Dr_Wang", "Dr_Liu"]
    all_arrangements = []
    for i in range(8):
        for comb in combinations(patients, i):
            remaining = [p for p in patients if p not in comb]
            for j in range(len(remaining) + 1):
                for comb2 in combinations(remaining, j):
                    remaining2 = [p for p in remaining if p not in comb2]
                    for k in range(len(remaining2) + 1):
                        for comb3 in combinations(remaining2, k):
                            remaining3 = [p for p in remaining2 if p not in comb3]
                            for perm in permutations([comb, comb2, comb3, remaining3]):
                                all_arrangements.append({
                                    doctors[0]: set(perm[0]),
                                    doctors[1]: set(perm[1]),
                                    doctors[2]: set(perm[2]),
                                    doctors[3]: set(perm[3]),
                                })
    valid_count = 0
    total_count = 0
    for arrangement in all_arrangements:
        if not inputs_check(arrangement):
            continue
        if verify_function(arrangement, inputs_check, constraint_list):
            valid_count += 1
        total_count += 1
    return valid_count, total_count
```

Now it's your turn:

Input:

```
{
    "background&definition": [[[background]]],
    "inputs_format": [[[inputs_format]]],
    "inputs_example": [[[inputs_example]]],
}
```

Output:

Figure 16: Traversal Function prompt for English Data(3/3). This prompt is designed to generate traversal functions that enumerate all possible solutions, using a two-shots demonstration approach.

提供:1、一道题的背景和定义 2、之前已经定义过一个函数名为'verify_function'，这里会提供它的输入要求（会提供取值范围及个数说明），以及一个输入样例， 这个'verify_function'函数直接调用'verify_fuction(inputs, inputs_check, constraint_list)'即可，其中inputs是用来代表一种方案，而inputs_check和constraint_list已经申明定义过，直接调用即可。这个函数输出就是这个方案的结果（是否满足所有逻辑限制）。注意在调用'verify_fuction(inputs, inputs_check, constraint_list)'，先检查一下'inputs_check(inputs)'为true。

目标：得到题目解的个数以及定义域的大小。

输出要求：实现'count_valid_arrangements'函数，给出python代码， 要求遍历定义域（根据输入要求的取值范围），并计算出verify_fuction为True的个数。return返回值是一个turple， 第一个元素是verify_fuction为True的个数， 第二个元素是遍历的定义域个数。

两个例子：

Input:

```
{
"background&definition": "背景和定义：某超市整齐排列着7排货架，放置着文具、零食、调料、日用品、酒、粮油和饮料7类商品，每类商品占据一排。",
"inputs_format": inputs是一个包含7个元素的list，每个元素是一个字符串，代表一排货架上的商品类别。取值范围是["文具","零食","调料","日用品","酒","粮油","饮料"]。,
"inputs_example": ["文具", "日用品", "零食", "调料", "酒", "粮油", "饮料"],
}
```

Output:

```
from itertools import permutations
def count_valid_arrangements():
    goods = ["文具","零食","调料","日用品","酒","粮油","饮料"]
    all_arrangements = list(permutations(goods, 7))
    valid_count = 0
    total_count = 0
    for arrangement in all_arrangements:
        if not inputs_check(list(arrangement)):
            continue
        if verify_function(list(arrangement), inputs_check, constraint_list):
            valid_count += 1
```

Figure 17: Traversal Function prompt for Chinese Data(1/3). This prompt is designed to generate traversal functions that enumerate all possible solutions, using a two-shots demonstration approach.

```
            total_count += 1
    return valid_count, total_count
```

Input:

```
{
"background&definition": "存在7名心脏病患者E、F、G、H、I、J、K需要被分配给4名医生张医生、李医生、王医生和刘医生进行治疗。每名患者只能由1位医生负责，每位医生最多负责两名患者的治疗。其中，J和K是儿童，其余5个是成年人；E、F和J是男性，其余4人是女性。",
"input_format": inputs是一个字典，包含四个键："Dr_Zhang", "Dr_Li", "Dr_Wang"和"Dr_Liu"。
inputs["Dr_Zhang"]是一个包含0-2个元素的set，包含张医生负责的患者名字（字符串）。取值范围是["E","F","G","H","I","J","K"]。inputs["Dr_Li"]是一个包含0-2个元素的set，包含李医生负责的患者名字（字符串）。取值范围是["E","F","G","H","I","J","K"]。
inputs["Dr_Wang"]是一个包含0-2个元素的set，包含王医生负责的患者名字（字符串）。取值范围是["E","F","G","H","I","J","K"]。inputs["Dr_Liu"]是一个包含0-2个元素的set，包含刘医生负责的患者名字（字符串）。取值范围是["E","F","G","H","I","J","K"]。,
"example_input": {
    "Dr_Zhang": {"E", "F"}, "Dr_Li": {"G"},
    "Dr_Wang": {"H", "J"}, "Dr_Liu": {"I", "K"}
},
}
```

Output:

```
from itertools import combinations, permutations
def count_valid_arrangements():
    patients = ["E","F","G","H","I","J","K"]
    doctors = ["Dr_Zhang", "Dr_Li", "Dr_Wang", "Dr_Liu"]
    all_arrangements = []
    for i in range(8):
        for comb in combinations(patients, i):
            remaining = [p for p in patients if p not in comb]
            for j in range(len(remaining) + 1):
                for comb2 in combinations(remaining, j):
                    remaining2 = [p for p in remaining if p not in comb2]
                    for k in range(len(remaining2) + 1):
                        for comb3 in combinations(remaining2, k):
                            remaining3 = [p for p in remaining2 if p not in comb3]
                            for perm in permutations([comb, comb2, comb3, remaining3]):
                                all_arrangements.append({
```

Figure 18: Traversal Function prompt for Chinese Data(2/3). This prompt is designed to generate traversal functions that enumerate all possible solutions, using a two-shots demonstration approach.

```
                                    doctors[0]: set(perm[0]),
                                    doctors[1]: set(perm[1]),
                                    doctors[2]: set(perm[2]),
                                    doctors[3]: set(perm[3]),
                                })
    valid_count = 0
    total_count = 0
    for arrangement in all_arrangements:
        if not inputs_check(arrangement):
            continue
        if verify_function(arrangement, inputs_check, constraint_list):
            valid_count += 1
        total_count += 1
    return valid_count, total_count
```

下面你来完成：

Input:

```
{
"background&definition": "[[[background]]]",
"inputs_format": "[[[inputs_format]]]",
"inputs_example": [[[inputs_example]]],
}
```

Output:

Figure 19: Traversal Function prompt for Chinese Data(3/3). This prompt is designed to generate traversal functions that enumerate all possible solutions, using a two-shots demonstration approach.

The following will provide 1. A logical reasoning question, including some descriptions required by logic_constraint 2. All solutions that satisfy all constraints, as well as descriptions of these solution formats. 3. The verification function code corresponding to these constraints

You need to use this information to 1. Find a new, stricter constraint (which must exclude some solutions), and also require that there are solutions that can satisfy this new constraint among the provided solutions (the number of new constraints must be 1). 2. And update the corresponding verification function code so that this new constraint can be checked by the verification function.

An example (please refer to the example for the format of the title and code block in the Output):

Input:

```
{
"background": "A supermarket is neatly arranged with 7 rows of shelves from front to back, placing 7 types of goods: stationery, snacks, condiments, daily necessities, wine, grain and oil, and beverages, each type of goods occupies one row.",
"logic_constraints": "The wine is in front of the condiments; there are 3 rows between the stationery and the condiments; the grain and oil is after the snacks, with 2 rows in between; the daily necessities are either in front of or behind the stationery.",
"solution_space": [
    "['Stationery', 'Daily necessities', 'Snacks', 'Wine', 'Condiments', 'Grain and oil', 'Beverages']",
    "['Stationery', 'Daily necessities', 'Wine', 'Snacks', 'Condiments', 'Beverages', 'Grain and oil']",
    "['Snacks', 'Stationery', 'Daily necessities', 'Grain and oil', 'Wine', 'Condiments', 'Beverages']",
    "['Snacks', 'Daily necessities', 'Stationery', 'Grain and oil', 'Wine', 'Beverages', 'Condiments']",
    "['Snacks', 'Daily necessities', 'Stationery', 'Grain and oil', 'Beverages', 'Wine', 'Condiments']",
    "['Snacks', 'Wine', 'Condiments', 'Grain and oil', 'Beverages', 'Daily necessities', 'Stationery']",
    "['Daily necessities', 'Stationery', 'Wine', 'Snacks', 'Beverages', 'Condiments', 'Grain and oil']",
    "['Daily necessities', 'Stationery', 'Beverages', 'Snacks', 'Wine', 'Condiments', 'Grain and oil']",
```

Figure 20: Stage 3 prompt for English Data(1/3). This prompt is designed to augment logical constraints by generating both textual descriptions and corresponding program-based verifiers, using a one-shot demonstration approach.

```
    "['Wine', 'Stationery', 'Daily necessities', 'Snacks', 'Beverages',
'Condiments', 'Grain and oil']",
    "['Wine', 'Snacks', 'Stationery', 'Daily necessities', 'Grain and oil',
'Beverages', 'Condiments']",
    "['Wine', 'Snacks', 'Condiments', 'Beverages', 'Grain and oil', 'Daily
necessities', 'Stationery']",
    "['Wine', 'Condiments', 'Beverages', 'Snacks', 'Daily necessities',
'Stationery', 'Grain and oil']",
    "['Beverages', 'Stationery', 'Daily necessities', 'Snacks', 'Wine',
'Condiments', 'Grain and oil']",
    "['Beverages', 'Snacks', 'Stationery', 'Daily necessities', 'Grain and
oil', 'Wine', 'Condiments']"
  ],
"Constraint_List_code": "def constraint_1(inputs):\n    if
inputs.index(\"Wine\") >= inputs.index(\"Condiments\"):\n        return
False\n    return True\n\ndef constraint_2(inputs):\n    if
abs(inputs.index(\"Stationery\") - inputs.index(\"Condiments\")) != 4:\n
  return False\n    return True\n\ndef constraint_3(inputs):\n    if
inputs.index(\"Grain and oil\") <= inputs.index(\"Snacks\") or
inputs.index(\"Grain and oil\") - inputs.index(\"Snacks\") != 3:\n
return False\n    return True\n\ndef constraint_4(inputs):\n    if
abs(inputs.index(\"Daily necessities\") - inputs.index(\"Stationery\")) !=
1:\n        return False\n    return True\n\nconstraint_list = [constraint_1,
constraint_2, constraint_3, constraint_4]",
}
```

Output:

Analysis

By observing the provided solutions, we can summarize some unique features of some solutions in this solution space, and then based on these features, we can find new stricter constraints, while still having solutions that can satisfy after adding new constraints.

Constraints

xxx

```
{"new_constraints": "Beverages cannot be placed in the first and last rows"}
```

Code

```
def constraint_1(inputs):
    if inputs.index("Wine") >= inputs.index("Condiments"):
```

Figure 21: Stage 3 prompt for English Data(2/3). This prompt is designed to augment logical constraints by generating both textual descriptions and corresponding program-based verifiers, using a one-shot demonstration approach.

```
        return False
    return True

def constraint_2(inputs):
    if abs(inputs.index("Stationery") - inputs.index("Condiments")) != 4:
        return False
    return True

def constraint_3(inputs):
    if inputs.index("Grain and oil") <= inputs.index("Snacks") or
inputs.index("Grain and oil") - inputs.index("Snacks") != 3:
        return False
    return True

def constraint_4(inputs):
    if abs(inputs.index("Daily necessities") - inputs.index("Stationery")) !=
1:
        return False
    return True

def constraint_5(inputs):
    # Beverages cannot be placed in the first and last rows
    if inputs.index("Beverages") == 0 or inputs.index("Beverages") ==
len(inputs) - 1:
        return False
    return True

constraint_list = [constraint_1, constraint_2, constraint_3, constraint_4,
constraint_5]
```

Now it's your turn:

Input:

```
{
"background": "[[[background]]]",
"logic_constraints": "[[[logic_constraints]]]",
"solution_space": [[[solution_space]]],
"Constraint_List_code": "[[[Constraint_List_code]]]",
}
```

Output:

Figure 22: Stage 3 prompt for English Data(3/3). This prompt is designed to augment logical constraints by generating both textual descriptions and corresponding program-based verifiers, using a one-shot demonstration approach.

下面会提供1、一道逻辑推理题，包含一些logic_constraint要求的描述 2、满足所有constraint的所有解，以及对这些解格式的描述。 3、对应这些constraint的验证函数代码

你需要根据这些信息，1、找到一个新的更严格的constraint(必须排除掉一些解)，同时还要求提供的这些解存在能满足这个新的constraint的解（新constraint个数必须是1个）。 2、并更新对应的验证函数代码，使得这个新的constraint能够被验证函数检查到。

一个例子(Output中标题和代码块的格式请参照例子)：

Input:

```
{
"background": "某超市从前到后整齐排列着7排货架，放置着文具、零食、调料、日用品、酒、粮油和饮料7类商品，每类商品占据一排。",
"logic_constraints": "酒类排在调料类之前；文具类和调料类中间隔着3排；粮油类在零食类之后，中间隔着2排；日用品类紧挨在文具类前一排或者后一排。",
"solution_space": [
    "['文具', '日用品', '零食', '酒', '调料', '粮油', '饮料']",
    "['文具', '日用品', '酒', '零食', '调料', '饮料', '粮油']",
    "['零食', '文具', '日用品', '粮油', '酒', '调料', '饮料']",
    "['零食', '日用品', '文具', '粮油', '酒', '饮料', '调料']",
    "['零食', '日用品', '文具', '粮油', '饮料', '酒', '调料']",
    "['零食', '酒', '调料', '粮油', '饮料', '日用品', '文具']",
    "['日用品', '文具', '酒', '零食', '饮料', '调料', '粮油']",
    "['日用品', '文具', '饮料', '零食', '酒', '调料', '粮油']",
    "['酒', '文具', '日用品', '零食', '饮料', '调料', '粮油']",
    "['酒', '零食', '文具', '日用品', '粮油', '饮料', '调料']",
    "['酒', '零食', '调料', '饮料', '粮油', '日用品', '文具']",
    "['酒', '调料', '饮料', '零食', '日用品', '文具', '粮油']",
    "['饮料', '文具', '日用品', '零食', '酒', '调料', '粮油']",
    "['饮料', '零食', '文具', '日用品', '粮油', '酒', '调料']"
  ],
"Constraint_List_code": "def constraint_1(inputs):\n    if inputs.index(\"酒\") >= inputs.index(\"调料\"):\n        return False\n    return True\n\ndef constraint_2(inputs):\n    if abs(inputs.index(\"文具\") - inputs.index(\"调料\")) != 4:\n        return False\n    return True\n\ndef constraint_3(inputs):\n    if inputs.index(\"粮油\") <= inputs.index(\"零食\") or inputs.index(\"粮油\") - inputs.index(\"零食\") != 3:\n        return False\n    return True\n\ndef constraint_4(inputs):\n    if abs(inputs.index(\"日用品\") - inputs.index(\"文具\")) != 1:\n        return
```

Figure 23: Stage 3 prompt for Chinese Data(1/3). This prompt is designed to augment logical constraints by generating both textual descriptions and corresponding program-based verifiers, using a one-shot demonstration approach.

```
    False\n    return True\n\nconstraint_list = [constraint_1, constraint_2,
    constraint_3, constraint_4]",
}
```

Output:

Analysis

观察提供的解决方案，我们可以归纳出这些solution space中的一些解的独有的特点，然后根据这些特点，我们就可以找到新的更严格的约束条件，同时使新增约束条件后也仍有解能满足。

Constraints

xxx

```
{"new_constraints": "饮料类不能排在第一排和最后一排",}
```

Code

```
def constraint_1(inputs):
    if inputs.index("酒") >= inputs.index("调料"):
        return False
    return True

def constraint_2(inputs):
    if abs(inputs.index("文具") - inputs.index("调料")) != 4:
        return False
    return True

def constraint_3(inputs):
    if inputs.index("粮油") <= inputs.index("零食") or inputs.index("粮油") -
inputs.index("零食") != 3:
        return False
    return True

def constraint_4(inputs):
    if abs(inputs.index("日用品") - inputs.index("文具")) != 1:
        return False
    return True

def constraint_5(inputs):
    # 饮料类不能排在第一排和最后一排
    if inputs.index("饮料") == 0 or inputs.index("饮料") == len(inputs) - 1:
        return False
```

Figure 24: Stage 3 prompt for Chinese Data(2/3). This prompt is designed to augment logical constraints by generating both textual descriptions and corresponding program-based verifiers, using a one-shot demonstration approach.

```
    return True

constraint_list = [constraint_1, constraint_2, constraint_3, constraint_4,
constraint_5]
```

下面你来做：

Input:

```
{
"background": "[[[background]]]",
"logic_constraints": "[[[logic_constraints]]]",
"solution_space": [[[solution_space]]],
"Constraint_List_code": "[[[Constraint_List_code]]]",
}
```

Output:

Figure 25: Stage 3 prompt for Chinese Data(3/3). This prompt is designed to augment logical constraints by generating both textual descriptions and corresponding program-based verifiers, using a one-shot demonstration approach.