Reformulating Programming Assignments: Balancing Correctness and LLM Resistance

Hsin-Chih Ho, Sin-Syuan Wu Yao-Chung Fan
Department of Computer Science and Engineering,
National Chung Hsing University, Taiwan

Abstract

This research addresses the challenge of creating effective programming assignments in the era of large language models (LLMs). Teachers can now use LLMs like ChatGPT to generate assignments, but students may also rely on these models to solve them. To mitigate this, we propose an Automated Algorithmic Question Generation (AAQG) method that augments dataset information to create scenario-based questions that challenge LLMs.

We introduce two new metrics: Topic Integrity Score (TIS) and Knowledge Similarity (KS). These evaluate the differences and core knowledge between generated and original problems. Our experiments using the LeetCode-sourced TACO dataset show improvements in TIS by up to 0.167 points and KS by up to 0.084 points over the baseline. Furthermore, Pass@k evaluation demonstrated our method's effectiveness in disrupting LLM performance, with GPT-3.5-turbo-0125 experiencing a 42-point drop in Pass@1 and CodeLlama-13b-Instruct-hf showing a 3.4-point drop in Pass@1 and a 7.21-point drop in Pass@5.

Keywords: LLM, Programming Education, Question Generation

1 Introduction

In computer programming education, teachers typically need to design programming assignments (as shown in table 1) to assist students in their learning. To improve the efficiency of assignment creation, teachers often modify existing past exam questions. However, with the advancement of large language models (such as ChatGPT and similar tools), teachers can combine past exam questions with new ideas and input them into the model to help generate new assignments. Nevertheless, when using large language models to generate programming assignments, teachers may face the following challenges.

- Correctness: The generated problem may lack a solution due to missing information or incoherent logic in the prompt. Additionally, the generated result might deviate from the original topic. For example, if the input is a sorting-related problem, the language model's output might reduce the problem to just a general array concept, straying away from the intended subject.
- Difficulty: During the learning process, it is difficult for teachers to prevent students from using large language models to solve their assignments. Therefore, teachers also hope that the problems they generate will be challenging for the language model to solve easily.

With the rise of large language models, their remarkable performance in various natural language processing tasks has garnered significant attention. In recent years, researchers (Austin et al., 2021; Hendrycks et al., 2021; Li et al., 2022; Nijkamp et al., 2022; Li et al., 2023a; Roziere et al., 2023; Li et al., 2023b) have begun focusing on the capabilities of large language models in code generation. Since the development of these models, their ability to generate code has seen notable improvement, particularly in solving algorithmic problems, where they have demonstrated great potential.

However, previous researchers have primarily focused on generating high-quality code and have yet to explore the generation of pro-

Question	Count the number of prime numbers less than a non-negative number, n. Example: Input: 10 Output: 4 Explanation: There are 4 prime numbers less than 10, they are 2, 3, 5, 7.
Skill types	Data structures, Range queries

Table 1: Example of a Programming Assignment: In the question description, in addition to the question itself, a set of test data is also included. In this example, the required problem-solving technique involves data structures and range queries.

gramming problems. Traditionally, programming problems rely on domain experts, such as teachers, to manually create them, which is both time-consuming and labor-intensive. Therefore, we aim to develop an automated algorithmic problem generation system to produce high-quality problems.

In practical terms, with the development of large language models, teachers are facing a new challenge: students are increasingly using tools like ChatGPT to complete algorithm assignments and write code. Therefore, we aim to generate problems that can effectively interfere with language models, making it difficult for them to provide easy solutions.

This study proposes an Automated Algorithmic Question Generation (AAQG) method. AAQG combines existing problems with their core problem-solving concepts and uses large language models to generate new algorithmic problems that share the same problem-solving core as the input problem. To evaluate the quality of the generated problems, we designed two assessment metrics specifically for algorithmic problems: Topic Integrity Score (TIS) and Knowledge Similarity (KS). The TIS metric assesses the differences between the generated and original problems, while the KS metric evaluates the similarity in core concepts between the generated and original problems.

In our experiments, we used a subset of the TACO dataset sourced from LeetCode and evaluated problem quality using our proposed TIS and KS metrics. We employed the Pass@k metric from (Chen et al., 2021) to observe whether large language models showed signs of interference, resulting in reduced problem-solving capabilities. The results indicate that our method improved the TIS score by up to 0.167 points and the KS score by up to 0.084 points. Furthermore, our method effectively reduced the language models' performance on Pass@k. When using CodeLlama-13b-Instruct-hf as the code generation model, Pass@1 dropped by up to 3.4 points and Pass@5 decreased by up to 7.21 points. When using GPT-3.5-turbo-0125, the reduction was even more significant, with Pass@1 dropping by up to 42 points.

2 Related Work

In this section, we primarily discuss the details of how previous researchers have utilized large language models for code generation.

Code Generation Model In the past, there have been numerous studies on code generation, such as (Austin et al., 2021; Hendrycks et al., 2021; Li et al., 2022; Nijkamp et al., 2022; Li et al., 2023a; Roziere et al., 2023), which focus on training new models for the task of code generation. Among these, (Roziere et al., 2023) introduced Code Llama, an open foundational model specifically designed for code generation.

Code Llama is notable for its robust multi-turn code generation capabilities, which make it particularly effective at handling complex algorithmic tasks. Built on the latest deep learning technologies, Code Llama can generate high-quality code and support a wide range of

code generation tasks, from simple to complex. Its open nature provides broad applicability in the research community and facilitates easy integration into various programming-related applications.

In (Li et al., 2022), the focus is on generating code for competitive programming problems. The paper introduces AlphaCode, a system designed to generate high-quality competition-level code. AlphaCode aims to participate in programming contests and produce code that meets competition standards. The authors present the model architecture, training process, and performance of AlphaCode in programming contests. Experimental results demonstrate that AlphaCode excels in generating code of competitive programming quality.

Code Generation Dataset In (Li et al., 2023b), a novel dataset called TACO was introduced for the code generation task. This dataset includes problems sourced from major online programming platforms such as LeetCode and Codewars, and it also features difficulty levels for the problems. The paper evaluates various large language models on this dataset, with GPT-4 showing the best performance, exceeding other large language models by up to 22.5 points.

Since the TACO dataset provides comprehensive information on algorithmic problem generation, we utilized it for our problem generation experiments. We also referenced the comparison subjects from the research to guide our experiments. For code generation models in our experiments, we used Code Llama and GPT.

3 Method

Algorithmic problem generation is an emerging research field, and as a result, there has been a lack of robust evaluation methods to assess the quality of generated problems. This section will introduce our proposed problem generation method and the design of automated metrics to evaluate different aspects of problem quality.

Therefore, in this section, we will cover: (1) how our method, Automated Algorithmic Question Generation (AAQG), automatically generates algorithmic problems, and (2) how we validate the quality of the generated problems.

3.1 AAQG

In this paper, we primarily use existing problems and their problem-solving techniques as inputs to automatically generate new problems related to the input through our automated process.

Figure 1 presents an overview of the entire AAQG process. We utilize a large language model (LLM) to accomplish our generation task. AAQG is divided into three main components: Query Expansion, Retrieve, and Question Generation.

3.1.1 Query Expansion

In this section, we use large language models to expand the input problems and their problem-solving techniques through prompting. This process is mainly divided into Question Expansion and Skill Type Expansion, as illustrated in Figure 2.

Question Expansion Since the ultimate goal is to generate new problems related to the input problem, we first input the problem into the LLM for data expansion. The LLM generates application scenarios related to the problem, including a ”title” and ”description.” This approach aims to provide context for the subsequent problem generation, making the generated problems richer and more contextualized.

Skill Type Expansion Since the problem-solving techniques in the dataset are often broad topics, such as data structures or dynamic programming, we aim to make the generated problems more focused. To achieve this, we input the problem and its associated techniques into the LLM. The LLM then supplements this information by identifying additional problem-solving techniques involved or specifying subtopics within the given technique, and describes the relevant problem-solving knowledge.

3.1.2 Retrieve

There have many past studies discuss the issue of hallucinations in large language models. To ensure that the generated problems are more realistic, we use the ”title” from the application scenarios generated by the LLM as input

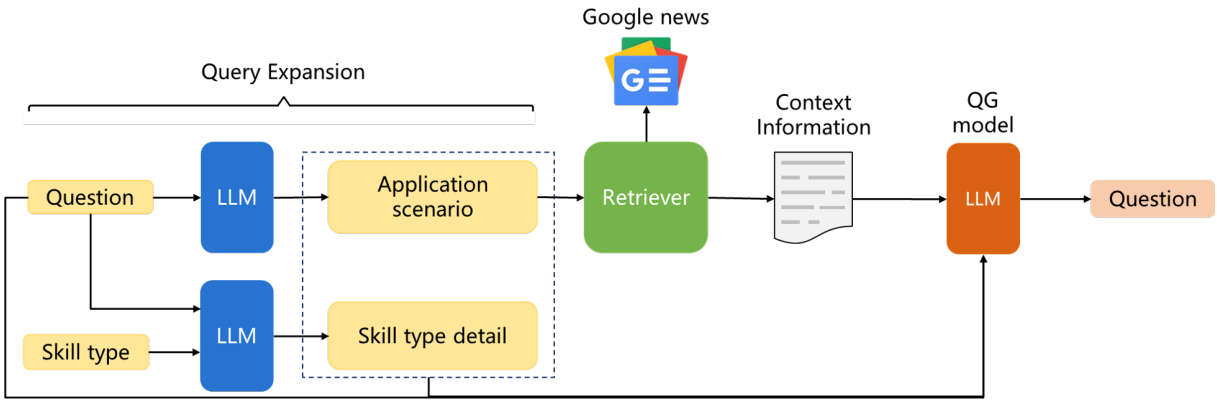

Figure 1: Overview of Methodology

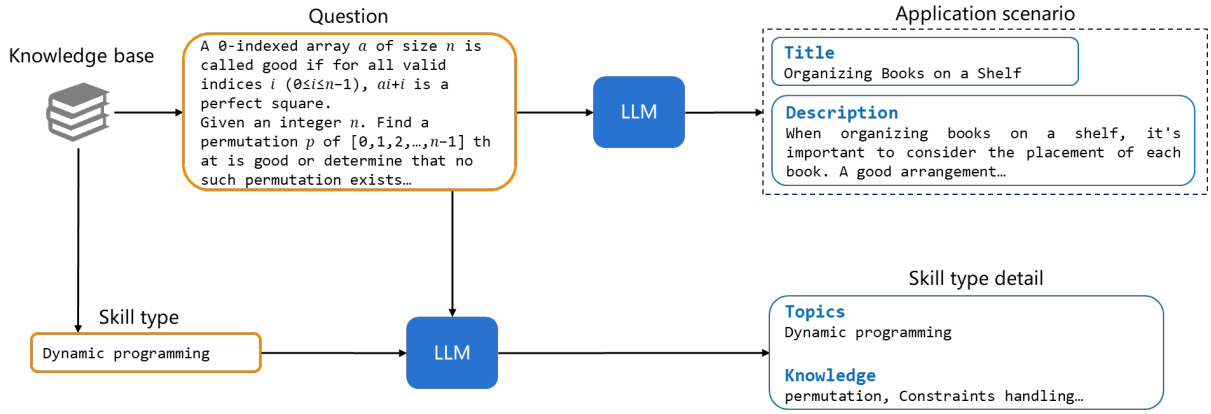

Figure 2: Query expansion

to retrieve the most relevant article. The goal is to make the content more grounded in reality. We use Google News as the retriever database to obtain a news article related to the problem, which serves as a reference for the subsequent problem generation.

3.1.3 Question Generation

In this section, we use the problems from the dataset, previously generated application scenarios, skill type detail, and a context information as inputs. We then employ the LLM to generate problems related to these inputs and create test data for the problems, as illustrated in Figure 4.

In the question generation phase, we use a two-stage generation approach. The first stage focuses on generating the questions, while the second stage is responsible for generating the test data for the questions. This two-stage approach is employed because generating both questions and test data simultaneously might distract the model from the current task, potentially leading to incoherent test data. Therefore, we choose to separate the generation of questions and test data.

Since subsequent validation will be conducted using unit tests, the LLM generates test data focusing on the following aspects: baseline testing, boundary testing, and random testing. As the generated problems do not currently account for execution time or memory requirements, stress testing is not specifically included in the test data generation. Ultimately, this process generates 3 to 10 test cases, with at least one test case for each of the baseline, boundary, and random tests.

3.2 Token Score For Question Generation

This section introduces the two evaluation metrics, Topic Integrity Score and Knowledge Similarity. These metrics assess the similarity between the generated problem and the original input problem in terms of their core problem-solving concepts.

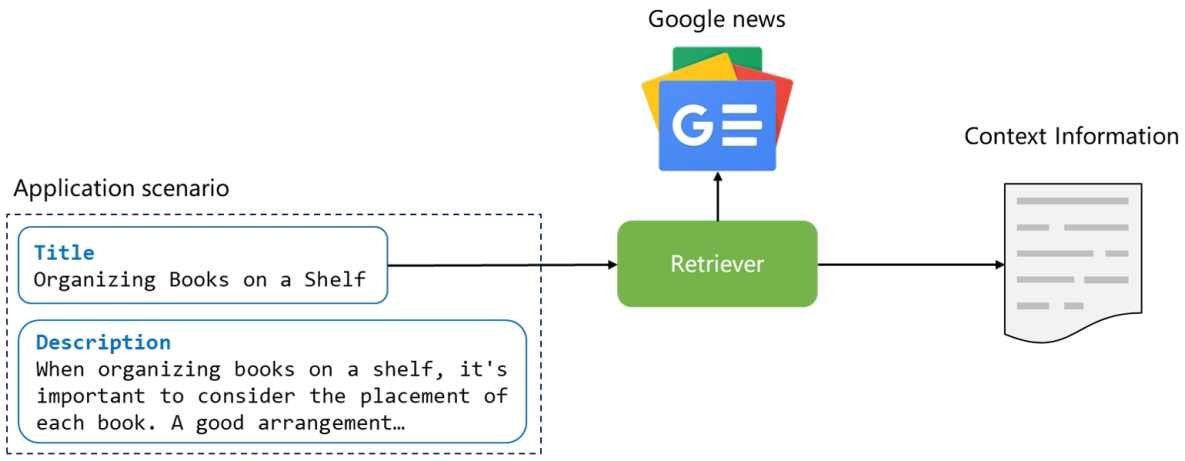

Figure 3: Retrieve

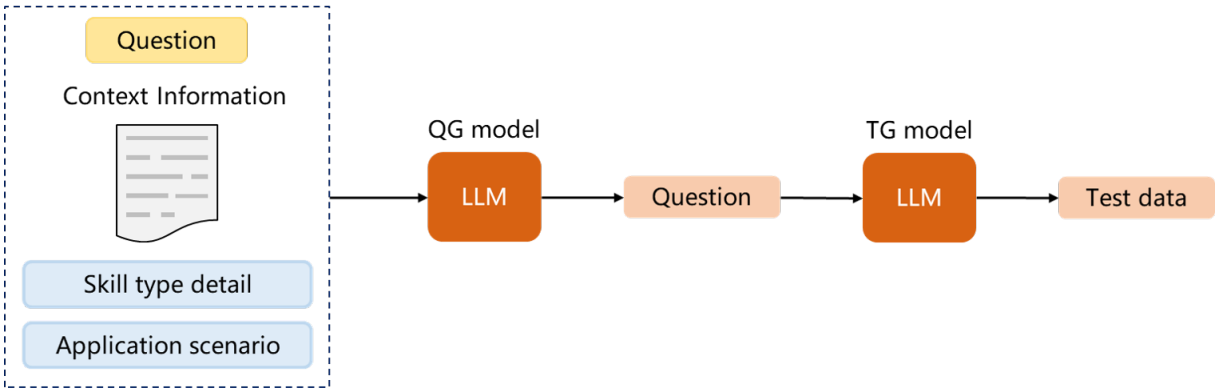

Figure 4: Query generation

3.2.1 Topic Integrity Score

Topic Integrity Score (TIS) is a comprehensive evaluation metric used to calculate the lexical and intrinsic differences between the generated problem and the original problem. Its goal is to ensure that the generated problem retains core similarity to the original problem while maximizing differences in wording, as shown in Formula 1.If the generated problem is identical to the original problem, the TIS score will be 0. This is because we aim for significant lexical differences between the generated problem and the original problem, while still maintaining similarity in core content.The TIS score ranges from 0 to 1.

$$TIS = \begin{cases} 0 & \text{if } S^Q = 1 \\ \frac{1-S^Q+D^Q}{2} & \text{otherwise} \end{cases} \quad (1)$$

In the formula 2, q_o represents the original input problem, while q_g denotes the generated problem. The score D^Q reflects the similarity of the intrinsic core between the generated problem and the original problem, specifically the dense score of the two problems, calculated using cosine similarity.

$$D^Q = \frac{\vec{q_o} \cdot \vec{q_g}}{|\vec{q_o}||\vec{q_g}|} \quad (2)$$

In the formula 3, S^Q represents the lexical similarity between the original input problem and the generated problem, specifically the sparse score of the two problems. This is based on the ROUGE-L score from (Lin, 2004), with Formulas 4 and 5 corresponding to the recall and precision scores, respectively.

$$S^Q = \frac{(1+\beta)^2 R_{LCS} P_{LCS}}{R_{LCS} + \beta^2 P_{LCS}} \quad (3)$$

$$R_{LCS} = \frac{LCS(q_o, q_g)}{len(q_o)} \quad (4)$$

$$P_{LCS} = \frac{LCS(q_o, q_g)}{len(q_g)} \quad (5)$$

3.2.2 Knowledge Similarity

Knowledge Similarity (KS) is an evaluation metric designed to calculate the similarity of problem-solving knowledge between the generated problem and the original problem, as shown in Formula 6.The KS score ranges from 0 to 1.

$$KS = \frac{1 + w \times S^K + D^K}{2 + w} \tag{6}$$

In Formula 7, k_o refers to the detailed problem-solving knowledge obtained from the skill type expansion of the original problem, while k_g represents the problem-solving knowledge details generated directly by the LLM using the same prompt as in the skill type expansion. The D^K score aims to measure the internal similarity of the problem-solving knowledge between the two, specifically the dense score of the knowledge, calculated using cosine similarity.

$$D^K = \frac{\vec{k_o} \cdot \vec{k_g}}{|\vec{k_o}||\vec{k_g}|} \tag{7}$$

Given the extensive specialized knowledge involved in the algorithm domain, we also consider the sparse score S^K between k_o and k_g, as shown in Formula 8. In the formula , w represents the proportion of the sparse score to be considered, with w ranging between 0 and 1. Since the results of each generation may vary, we place more emphasis on the dense score.

$$S^K = \frac{(1+\beta)^2 R_{LCS} P_{LCS}}{R_{LCS} + \beta^2 P_{LCS}} \tag{8}$$

$$R_{LCS} = \frac{LCS(k_o, k_g)}{len(k_o)} \tag{9}$$

$$P_{LCS} = \frac{LCS(k_o, k_g)}{len(k_g)} \tag{10}$$

4 Experiment

In this section, we will introduce the dataset we used, the experimental setup details, the evaluation metrics, and the experimental results.

4.1 Dataset

Our experiments primarily utilized data from the TACO dataset, specifically focusing on problems sourced from LeetCode. The TACO dataset comprises problems from various platforms, each with its own unique difficulty definitions. To ensure the clarity and relevance of our experimental results, we chose to use data exclusively from LeetCode, categorized into three difficulty levels: Easy, Medium, and Medium_Hard, totaling 646 records. The detailed distribution is shown in table 2, which outlines the 646 problems divided into Easy, Medium, and Medium_Hard difficulty levels.

LeetCode	Easy	Medium	Medium_Hard
# of Questions	124	276	246

Table 2: The distribution of data from LeetCode within the TACO dataset, where the difficulty levels are limited to Easy, Medium, and Medium_Hard.

4.2 Implementation Details

In this section, we will provide a comprehensive description of the language models used and the code generation models employed for validation.

4.2.1 The Employed LLMs

- AAQG Model we use the GPT-3.5-turbo-0125 model developed by the OpenAI team. We access the model via API, set the temperature to 0.5, and utilize the function calling approach during operation.
- Code Generation Model we tested with the GPT-3.5-turbo-0125 model developed by OpenAI and the CodeLlama-13b-Instruct-hf model provided by Meta. The CodeLlama-13b-Instruct-hf model was downloaded and used through the Hugging Face platform.
- GPT-3.5-turbo-0125: We use the default temperature setting to generate code and unit test snippets.
- CodeLlama-13b-Instruct-hf: We set the temperature to 0.9 and top_p to 0.95 for generating code and unit test snippets.

4.3 Automatic Metric

We use our proposed TIS and KS metrics, as well as the pass@k metric introduced by (Chen et al., 2021), as evaluation indicators.

- TIS: This evaluation metric is used to assess the diversity of the generated questions in terms of format and their consistency in content.

- KS: This evaluation metric is used to assess the similarity in problem-solving knowledge between the generated questions and the original problems.

- Pass@k: This metric calculates the probability of the LLM correctly answering the questions.

4.3.1 Embeding Model

In the scoring metrics, TIS and KS, the score D involves converting sentences into embeddings and calculating their similarity. For the embedding conversion, we use the BGE-M3 model proposed by (Multi-Granularity), which demonstrates excellent performance in similarity scoring.

4.3.2 Baselines

In the experiments on question generation, there are two main parts.

- Differences Between LLM-Generated Questions and Original Questions: Since there is no existing research focused on problem generation in the algorithm domain, we use a direct approach where problems are input into the LLM, and problems are generated through prompting as our baseline.

- Impact on Language Model Answering Capabilities: In this experiment, we use problems from the TACO dataset as input for the language model to generate code, which serves as our baseline.

4.4 Experimental results

In this section, we primarily discuss two experiments: 1. The experiment on the differences between the LLM-Generated questions and the original questions. 2. The experiment on the impact of generated questions on the language model's answering capabilities.

4.4.1 Differences Between LLM-Generated Problems and Original Problems

In table 3, we present experiments conducted across three difficulty levels: Easy Medium and Medium_Hard.

The experimental data indicates that when problems are directly generated by the LLM, the TIS score is consistently 0. According to Formula 1, this implies that the sparse score of the generated problems is identical to that of the input problems, meaning that the generated problems are identical to the original ones. We speculate that this phenomenon may be due to the dominant influence of the input problems in the prompt, which causes the model to generate problems that are identical to the input ones.

We also compared the AAQG method with different configurations of AAQG, including settings without Skill Type Expansion and Retrieve. The experimental data shows that, although the TIS score for AAQG is 0.002 points lower than the configuration without the Retrieve component in the Medium difficulty level, the KS score is higher by 0.068 points. Thus, the AAQG method demonstrates the best overall performance.

The experiments also show that removing the Skill Type Expansion step leads to a significant decrease in both TIS and KS scores, with the KS score dropping by as much as 0.073 points. Therefore, Skill Type Expansion is crucial in the AAQG method for maintaining the quality of the generated questions.

4.4.2 Impact on Language Model Answering Capabilities

In table 4, it is evident that the AAQG method results in a significant decrease in pass@1 and pass@5 scores, regardless of whether GPT-3.5-turbo-0125 or CodeLlama-13b-Instruct-hf models are used. This indicates that the AAQG method can effectively disrupt the performance of large language models. Notably, when using GPT-3.5-turbo-0125, the pass@1 score decreased by as much as 42 points.

The table also shows that as the difficulty level increases, the proportion of decline in pass scores decreases. This indicates that packaging simpler problems tends to be more effective, as harder problems are inherently more challenging for LLMs to solve. Conse-

Difficulty	Easy		Medium		Medium_Hard	
	TIS(↑)	KS(↑)	TIS(↑)	KS(↑)	TIS(↑)	KS(↑)
Baseline	0	0.761	0	0.735	0	0.74
AAQG	0.667	0.821	0.662	0.822	0.662	0.824
w/o Skill Type Expansion	0.655	0.752	0.656	0.749	0.656	0.758
w/o Retrieve	0.661	0.751	0.664	0.754	0.661	0.757

Table 3: For evaluating problem quality, the performance of the AAQG method is compared with the baseline and variations of the AAQG method where different components are removed, across different difficulty levels of generated problems.

Method	Model	Easy		Medium		Medium_Hard	
		Pass@1(↓)	Pass@5(↓)	Pass@1(↓)	Pass@5(↓)	Pass@1(↓)	Pass@5(↓)
Baseline	GPT-3.5-turbo-0125	64	-	32	-	36	-
AAQG	GPT-3.5-turbo-0125	22	-	25	-	28	-
Baseline	CodeLlama-13b-Instruct-hf	5.4	13.89	4.8	14.21	1.8	8.11
AAQG	CodeLlama-13b-Instruct-hf	2.0	8.67	1.4	7.0	1.2	5.56

Table 4: The performance of code generation pass rates across different models and difficulty levels.

quently, packaging higher-difficulty problems results in less noticeable improvements.

5 Conclusion

we are the first to explore the task of algorithmic problem generation. We introduced the Automated Algorithmic Question Generation (AAQG) method, which effectively enhances the efficiency of algorithm problem creation. Our experiments demonstrate that problems generated using AAQG can significantly disrupt the code generation capabilities of large language models. Specifically, under the GPT-3.5-turbo-0125 setting, the pass@1 performance decreased by up to 42 points, while in the CodeLlama-13b-Instruct-hf setting, pass@1 dropped by up to 3.4 points and pass@5 decreased by up to 7.21 points.

We also introduced two novel automated evaluation metrics for assessing the quality of generated problems. These metrics demonstrate that problems generated using the AAQG method can maintain the same core essence as the input problems while exhibiting variations.

References

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

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

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. 2021. Measuring coding challenge competence with apps. arXiv preprint arXiv:2105.09938.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023a. Starcoder: may the source be with you! arXiv preprint arXiv:2305.06161.

Rongao Li, Jie Fu, Bo-Wen Zhang, Tao Huang, Zhihong Sun, Chen Lyu, Guang Liu, Zhi Jin, and Ge Li. 2023b. Taco: Topics in algorithmic code generation dataset. arXiv preprint arXiv:2312.14852.

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

Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In Text summarization branches out, pages 74–81.

Multi-Linguality Multi-Functionality Multi-Granularity. M3-embedding: Multi-linguality, multi-functionality, multi-granularity text embeddings through self-knowledge distillation.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. arXiv preprint arXiv:2203.13474.

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

A Data

The TACO(Li et al., 2023b) dataset, which stands for Topics in Algorithmic COde generation, is primarily designed for the code generation task. This dataset collects problems from various coding platforms such as LeetCode, Codewars, etc. The distribution of the dataset is shown in table 5 . We believe that its rich data is also well-suited for the task of algorithmic problem generation.

Dataset	Train	Test	All
# of Questions	25443	1000	26443

Table 5: This is the data distribution of the TACO dataset.

Table 6 lists the data distribution across all platforms in the TACO dataset, excluding records where the skill_types field is empty, the solution is empty, or the difficulty is labeled as UNKNOWN_DIFFICULTY.

Source	Train	Test	All	Filiter
codeforces	8193	576	8769	4264
aizu	2151	0	2151	0
geeksforgeeks	2680	0	2680	1965
codewars	2460	55	2515	646
kattis	1236	0	1236	0
codechef	3352	278	3630	1268
hackerearth	2390	45	2435	229
atcoder	1440	0	1440	0
leetcode	777	0	777	646
hackerrank	764	46	810	261

Table 6: This Table lists the data distribution for each platform in the TACO dataset.

B Code Generation Detail

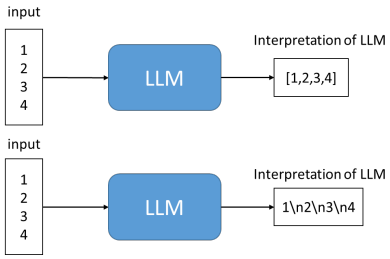

Figure 5: Interpretation of input in LLM: In the same input, the LLM may interpret the data differently, leading to variations in syntax in the output.

During the code generation process, we also use prompting to have the model generate unit test snippets. These snippets are used to guide the model on how to apply the inputs to the generated functions. In this stage of data processing, special cases such as spaces in test data may lead to different interpretations by the model, requiring various data processing methods.

As shown in Figure 5, the LLM may interpret the same input in two different ways, leading to variations in the generated code based on its interpretation. The LLM will process the input according to the format it has interpreted. Therefore, we need to ask the LLM to specify the format of the input that should be provided to its function.

C Pass Rate

As shown in formula eq. (11), where m represents all questions, n indicates the number of attempts the LLM makes to generate code for a single question, and c represents the number of code snippets among these n that pass the unit tests, i.e., the number of correct snippets out of n.

In this formula, when $n = 1$, it directly considers the total number of correct answers. For $n > 1$, it calculates the probability of getting at least one correct answer out of n attempts.

$$pass@k = 1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \tag{11}$$

When using the GPT-3.5-turbo-0125 model, due to cost considerations, it was only run once per attempt, while other models were allowed 10 attempts.

D Ablation Study

D.1 Impact on Language Model Answering Capabilities

In table 7, we used GPT-3.5-turbo-0125 as our code generation model and employed pass@1 as the evaluation metric. We compared different settings of AAQG with the baseline. The results show a significant decrease in scores for AAQG settings compared to the baseline. Among the various settings, the "without Skill Type Expansion" configuration performed best in the Medium_Hard difficulty level, surpassing AAQG by 4 points. However, considering all three difficulty levels, AAQG still demonstrates the best overall performance.

We also observed that in the absence of the retrieve component, the average score across all difficulty levels decreased by 7 points. This indicates that additional relevant articles serve effectively as interference noise in the LLM and can significantly impact the model's ability to generate correct answers.

In tables 8 to 10, we use the same settings as in table 7 but replace the code generation model with CodeLlama-13b-Instruct-hf. This table shows that under different settings of AAQG, there is a noticeable decrease in pass@1 and pass@5 scores compared to the baseline.

In the Easy and Medium difficulty levels, as shown in tables 8 and 9, although the performance in pass@1 and pass@5 is best without Retrieve, the pass@10 scores drop to 55.56% of the original baseline. This suggests that in this setting, the decline in LLM's answering ability may not be due to increased difficulty, but rather due to a lack of sufficient information in most of the generated problems, making many of them unsolvable. This indicates that relying solely on LLM-generated application scenarios can lead to information gaps, resulting in incomplete and unsolvable problems. Therefore, it is crucial to use reference articles to assist in problem generation.

In Medium_Hard difficulty problems, as shown in table 10, although the model performs best without using Retrieve, achieving results equivalent to the baseline, the pass@1 and pass@5 scores show an increase compared to the baseline. This indicates that while reference articles can provide complete information to aid in problem generation, this step may also introduce noise, which can lead to a decline in the LLM's answering ability.

Across all three difficulty levels, it is evident that the complete AAQG setup yields the best overall performance. With only a slight decrease in pass@10 scores, AAQG effectively reduces pass@1 and pass@5 scores. This indicates that problems generated using the AAQG method can significantly interfere with the language model's answering ability.

Method	Easy	Medium	Medium_Hard
Baseline	64	32	36
AAQG	22	25	28
w/o Skill Type Expansion	38	32	24
w/o Retrieve	34	36	26

Table 7: For different difficulty levels and methods, we use the GPT-3.5-turbo-0125 model and employ pass@1 as the evaluation metric to measure the pass rate performance of the generated problems.

Method	Easy		
	Pass@1 (↓)	Pass@5 (↓)	Pass@10 (↑)
Baseline	5.40	13.89	18.0
AAQG	2.0 (-62.96%)	8.67 (-37.58%)	14.0 (-22.22%)
w/o Skill Type Expansion	2.6 (-51.85%)	9.5 (-31.61%)	14.0 (-22.22%)
w/o Retrieve	0.8 (-85.19%)	4.0 (-71.2%)	8.0 (-55.56%)

Table 8: For the Easy difficulty level, we evaluated the model's pass rate performance using different methods. We employed the CodeLlama-13b-Instruct-hf model and used pass@k as the evaluation metric to measure the impact of generated problems on the model's pass rate. The numbers in parentheses next to the pass@k scores indicate the percentage decrease in performance compared to the baseline.

Method	Medium		
	Pass@1 (↓)	Pass@5 (↓)	Pass@10 (↑)
Baseline	4.80	14.21	20.0
AAQG	1.4 (-70.83%)	7.0 (-50.74%)	14.0 (-30%)
w/o Skill Type Expansion	1.6 (-66.67%)	7.54 (-46.8%)	14.0 (-30%)
w/o Retrieve	1.2 (-75%)	6.0 (-57.78%)	12.0 (-40%)

Table 9: For the Medium difficulty level, we evaluated the model's pass rate performance using different methods. We employed the CodeLlama-13b-Instruct-hf model and used pass@k as the evaluation metric to measure the impact of generated problems on the model's pass rate. The numbers in parentheses next to the pass@k scores indicate the percentage decrease in performance compared to the baseline.

Method	Medium_Hard		
	Pass@1 (↓)	Pass@5 (↓)	Pass@10 (↑)
Baseline	1.80	8.11	14.0
AAQG	1.2 (-33.33%)	5.56 (-31.44%)	10.0 (-28.57%)
w/o Skill Type Expansion	1.6 (-11.11%)	6.39 (-21.21%)	10.0 (-28.57%)
w/o Retrieve	2.0 (+11.11%)	8.67 (+6.91%)	14.0 (0%)

Table 10: For the Medium_Hard difficulty level, we evaluated the model's pass rate performance using different methods. We employed the CodeLlama-13b-Instruct-hf model and used pass@k as the evaluation metric to measure the impact of generated problems on the model's pass rate. The numbers in parentheses next to the pass@k scores indicate the percentage decrease in performance compared to the baseline.