Beyond the Surface: A Solution-Aware Retrieval Model for Competition-level Code Generation

Shiwen Zhang[1,2], **Lingxiang Wang**[1,2*], **Hainan Zhang**[1,2†],
Ziwei Wang[1,2], **Sijia Wen**[1,2], **Zhiming Zheng**[1,2]
[1]Beijing Advanced Innovation Center for Future Blockchain and Privacy Computing
[2]Institute of Artificial Intelligence, Beihang University, China
{zhangshiwen, wanglingxiang, zhanghainan}@buaa.edu.cn

Abstract

In competitive programming task, problem statements are often embedded within elaborate narrative backgrounds, requiring deep understanding of the underlying solutions to successfully complete the tasks. Current code generation models primarily focus on token-level semantic modeling, highly susceptible to distractions from irrelevant narrative statements. Inspired by RAG, retrieving reference code with similar solutions may help enhance model performance on difficult problems. However, existing retrieval models also emphasize surface-level semantic similarity, neglecting the deeper solution-level logical similarities that are critical in competitive programming. Therefore, designing ranking models capable of accurately identifying and retrieving problems and corresponding codes remains an urgent research problem in competitive code generation. In this paper, we propose SolveRank, a solution-aware ranking model empowered by synthetic data for competitive programming tasks. Specifically, we leverage the DeepSeek-R1 model to generate logically equivalent but differently phrased new problems, verified by GPT-4o for solution consistency. Then, we train SolveRank with these as positive samples and BM25/random-retrieved problems as negatives. During inference, SolveRank retrieves relevant problems and corresponding code from the corpus to assist a downstream code generator. Experiments on the xCodeEval dataset demonstrate that SolveRank outperforms SOTA ranking methods in precision and recall metrics, and boosts code generation performance for difficult problems.

1 Introduction

Recent large language models (LLMs) achieve human-level performance on simple programming tasks (Zheng et al., 2023; Wang et al., 2025), but struggle with competitive problems (Li et al., 2022). This discrepancy arises because simple programming tasks rely on surface-level instruction following which can be addressed by aligning with human preferences, while competitive problem statements are often embedded within elaborate narrative backgrounds, demanding deeper understanding of the underlying solutions to successfully complete the programming tasks.

Since current code generation models are typically trained using token-level semantic information, they are highly susceptible to being influenced by the narrative problem statements. As illustrated in Figure 1, Problems 1 and 2 differ only slightly in wording but require entirely different solutions:

Problem 1

You are browsing a store with n gifts, each priced at $p_1, p_2, ..., p_n$ sorted in ascending order. You have a fixed budget A and want to buy the most expensive gift you can afford. That is, among all gifts priced at or below A, select the one with the maximum price.

Problem 2

In a store, there are n gifts, each with a price p_i. The gift prices are sorted in ascending order. You are given a budget B and want to buy a set of gifts. Each gift may be selected at most once. Your goal is to maximize the total price of the selected gifts without exceeding the budget.

Problem 3

A spaceship cargo bay contains n energy crystals, each emitting a power level of $e_1, e_2, ..., e_n$, ordered from weakest to strongest. To stabilize the reactor, you must insert a crystal whose power does not exceed a safety threshold T. Select the strongest crystal whose emission level is at most T.

Figure 1: Examples of solution-level logical similar and semantically similar problems. Problems 1 and Problems 3 differ in surface descriptions but share the same algorithm (binary search). Problem 2 has similar background and vocabulary to Problem 1, yet requires a different solution approach.

*Equally Contribution.
†Corresponding author.

Findings of the Association for Computational Linguistics: EMNLP 2025, pages 5237–5246
November 4-9, 2025 ©2025 Association for Computational Linguistics

Problem 1 is an upper bound search problem in a sorted array, whereas Problem 2 is a 0/1 Knapsack Problem. This semantic similarity can mislead LLMs into generating incorrect solutions by recalling analogous but inappropriate training examples. Therefore, current LLMs still face a significant challenges in solving competitive programming.

Inspired by Retrieval-Augmented Generation (RAG), reference code that shares the same underlying solution as the current problem may help LLMs generate correct code in competitive programming tasks. As shown in Figure 2, we observe that for simple problems(<=1400), LLMs can directly generate correct code without RAG. But for more challenging problems(>1400), RAG proves to be highly beneficial. Specifically, for difficult problems, retrieving reference code improves the pass@1 rate from 30.00% to 35.84%, highlighting the effectiveness of RAG in enhancing LLM performance on competitive programming problems.

However, existing retrieval models primarily focus on surface-level semantic similarity, often retrieving problems with similar wording but lacking deep solution relevance. Downstream performance analysis shows that solution-aware retrieval models significantly outperform DPR, achieving a pass@1 of 35.84% vs. 31.67% in Figure 2. Notably, inaccurate retrieval in RAG can mislead code generation in competitive programming. Therefore, designing ranking models that can accurately identify and retrieve solution-relevant problems remains an urgent challenge in competitive programming tasks.

In this paper, we propose SolveRank, a solution-aware retrieval model empowered by synthetic data for competitive programming tasks. Specifically, we first leverage the DeepSeek-R1 model to generate multiple synthetic problem statements for current problem that are logically equivalent but differ in surface phrasing. We then use GPT-4o as a discriminator to verify that the generated problems indeed share the same solution with the current problem. Next, we employ contrastive learning to train SolveRank, using the synthetic problems as positive samples, and BM25/random-retrieved real problems as negative. Finally, we apply our SolveRank model to retrieve solution-relevant problems and their reference code from the corpus, which are then provided as context to a downstream code generation model for difficult problems.

Experiments on the xCodeEval (Khan et al., 2024) dataset demonstrate that SolveRank significantly outperforms existing retrieval models(about 406% MRR increment), and improves code generation performance(about 20% pass@1 increment) for difficult problems, validating the effectiveness of SolveRank on competitive code generation.[1]

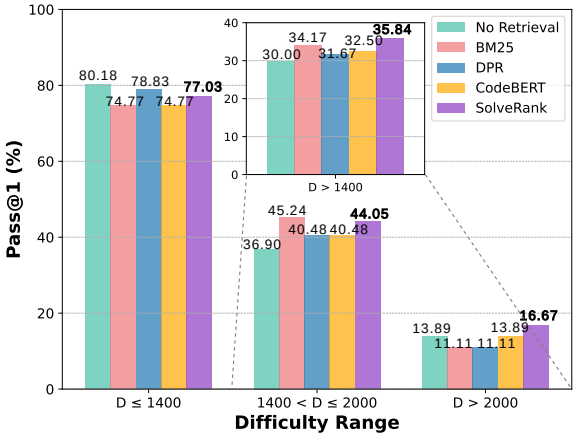

Figure 2: Comparison results of RAG with different ranking methods and without RAG on xCodeEval-python test set. The x-axis is the difficulty score of problems and the y-axis is the pass@1 rate.

The innovations in this paper are as follows:

- We find that RAG is particularly beneficial for solving difficult problems in competitive programming, and solution-aware retrievers outperform semantic-based retrievers.
- We propose a solution-level ranking task focused on assessing whether candidate solutions truly address the query's intent over surface-level language matching, and release two supporting datasets.
- We introduce SolveRank to retrieve relevant solutions to help LLMs generate correct code, significantly improving performance on difficult problems of competitive programming.

2 Method

The framework of SolveRank consists of three stages, as illustrated in Figure 3. We use the DeepSeek-R1 model to generate logically equivalent but differently phrased problems, which are verified with GPT-4o for consistency. Then, we use these as positive samples to train SolveRank, with BM25/random-retrieved problems as negative ones. During inference, we retrieve relevant problems and code to assist the downstream code generator.

[1]Our code and data are available at: https://github.com/lotus-0216/SolveRank

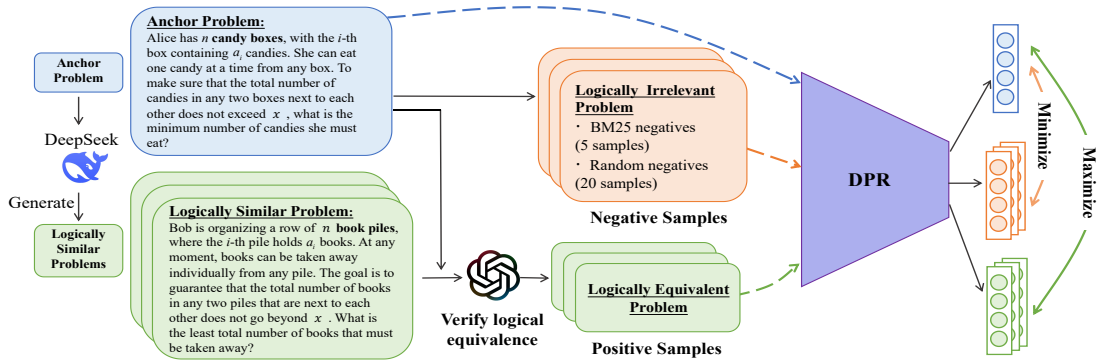

Figure 3: Training pipeline for Solution-Aware Retriever **SolveRank**.

2.1 Task Definition

Let $\mathcal{Q} = \{(q_i, c_i)\}_{i=1}^{N}$ denote a corpus of natural language programming problems and their corresponding reference codes. Given a target problem $q' \in \mathcal{Q}_{\text{test}}$, the objective of solution-level ranking task is to retrieve a list of problem-code pairs $\mathcal{R}_q = \{(r_1, c_1), ..., (r_K, c_K)\} \subset \mathcal{Q}_{\text{train}}$ such that each problem r_j is logically equivalent to q'. During code generation, the top-K logically equivalent problem–code pairs $\{(r_j, c_j)\}_{j=1}^{K}$ are retrieved and concatenated with the target problem q to create an input prompt for code generation models.

2.2 Synthetic Data Conduction

Due to the lack of solution-relevant retrieval data, we construct the training set using synthetic data. Specifically, for each anchor problem $q \in \mathcal{Q}_{\text{train}}$, we use the DeepSeek-R1 model to generate new problems $\mathcal{P}_q = \{q_i^+\}_{i=1}^{5}$. The prompt (see Appendix A.1) is crafted to preserve the original solution logic while encouraging diversity in the problem background.

To ensure true logical equivalence, we apply GPT-4o as an automatic verifier. For each generated variant $q_i^+ \in \mathcal{P}_q$, GPT-4o is prompted to assess whether $\text{Logic}(q^+) \equiv \text{Logic}(q)$, focusing strictly on algorithm class and solution decomposition while ignoring superficial narrative or vocabulary differences(see in Appendix A.2).A statistical comparison of synthetic and original problems is provided in Appendix 6.

2.3 Solution-Aware Retriever

We adopt a contrastive learning approach to train a DPR model (Karpukhin et al., 2020). The **positive samples** are drawn from the synthetic dataset $\mathcal{P}_q$, while the **negative samples** $\mathcal{N}_q = \{q_j^-\}_{j=1}^{25}$ consist of the top-5 retrieved by BM25 and 20 randomly sampled problems from the training corpus.

We use the **InfoNCE** loss to encourage the encoder to bring logic-equivalent problems closer in the embedding space while pushing apart logical distractors. The loss is defined as:

$$\mathcal{L} = -\log \frac{\exp\left(\frac{\text{sim}(q,q^+)}{\tau}\right)}{\exp\left(\frac{\text{sim}(q,q^+)}{\tau}\right) + \sum_{q^- \in \mathcal{N}_q} \exp\left(\frac{\text{sim}(q,q^-)}{\tau}\right)}, \tag{1}$$

$$\text{sim}(q, r) = E_Q(q)^\top E_P(r), \tag{2}$$

where τ is a hyperparameter, $E_Q(\cdot)$ and $E_P(\cdot)$ are query and passage encoders from DPR model.

After that, given a target problem q' and its top-K retrieved candidates $\{r_1, r_2, ..., r_K\}$, we verify whether each r_j satisfies logical equivalence with q with GPT-4o judgment(see in Appendix A.2).

2.4 Retrieval-Augmented Code Generation

Given the top-K problem–code pairs $\{(r_j, c_j)\}_{j=1}^{K}$ retained for the target problem q' , we concatenate the verified examples and the target problem into a single input, formatted as:

$$\text{Prompt}(q) = \text{Concat}\left(\{(r_1, c_1), \ldots, (r_K, c_K)\},\ q\right).$$

We use large language models to generate Python code in an autoregressive manner. The model operates in a zero-shot setting, generating the code until an end-of-function token is produced or a maximum length is reached.

3 Experiments

3.1 Experimental Setups

Dataset We use the **xCodeEval** benchmark for experiments, which includes competitive programming problems in areas like dynamic programming, graph traversal, greedy algorithms, and simulation. The official **test set** of the NL-Code Retrieval task

Table 1: Pass@1 (%) for different difficulty levels on xCodeEval using GPT-3.5(a) and GPT-4o(b). D denotes the official difficulty score.

Method	$D \leq 1400$	$1400 < D \leq 2000$	$D > 2000$
No Retrieval	**49.77**	10.71	5.56
Random	38.29	10.71	5.56
BM25	38.74	14.29	8.33
DPR	45.50	11.90	5.56
ReACC	43.69	13.10	5.56
CodeBERT	41.18	**17.86**	**11.11**
SolveRank	40.54	13.10	**11.11**

(a)

Method	$D \leq 1400$	$1400 < D \leq 2000$	$D > 2000$
No Retrieval	**80.18**	36.90	13.89
Random	72.52	42.86	11.11
BM25	74.77	**45.24**	11.11
DPR	78.83	40.48	11.11
ReACC	72.52	38.10	13.04
CodeBERT	74.77	40.48	13.89
SolveRank	77.03	**44.05**	**16.67**

(b)

Table 2: Ranking performance by retrieving synthetic solution-relevant problems from xCodeEval-python.

Method	P@1	R@1	P@3	R@3	P@5	R@5	MRR
BM25	0.131	0.026	0.198	0.068	0.240	0.103	0.186
DPR	0.039	0.008	0.059	0.020	0.069	0.030	0.057
ReACC	0.027	0.005	0.064	0.016	0.083	0.024	0.057
CodeBERT	0.096	0.019	0.167	0.048	0.193	0.066	0.147
SolveRank	**0.682**	**0.136**	**0.808**	**0.385**	**0.842**	**0.593**	**0.755**

is used for evaluation, while the training set serves as the retrieval corpus. Since the `ExecEval` platform of xCodeEval only supports problems from the `program_synthesis` subset for functional evaluation, we filter the NL-Code Retrieval test set to keep 342 suitable problems.

Baselines We compare SolveRank with three SOTA ranking methods: BM25 (Robertson et al., 2009), CodeBERT (Feng et al., 2020) , DPR (Karpukhin et al., 2020) and ReACC (Lu et al., 2022).

Evaluation Metrics We use Pass@1 as the primary metric for competition-level code generation, measuring the proportion of problems where the top-1 generated code passes all test cases via ExecEval platform. For the solution-level ranking task, we use Precision(P@K), Recall(R@K) and MRR to evaluate the model performance.

Implementation Details We use GPT-4o and GPT-3.5 as the code generation models in zero-shot inference mode. SolveRank is trained on a dual-GPU server (NVIDIA RTX A6000) for 10 epochs using a batch size of 4 and a learning rate of 3×10^{-5}, under CUDA 12.5.

3.2 Main Results

The study evaluates Pass@1 performance across three difficulty levels (Easy<=1400, 1400 < Medium <=2000, and Hard>2000) on the xCodeEval dataset using GPT-3.5-turbo and GPT-4o, as shown in Table 1. The results show that for easy problems, retrieval offers no significant improvement, sometimes even decreasing performance. This suggests that LLMs can already solve simple problems effectively without additional guidance. In contrast, for medium and hard tasks, all the retrieval methods enhance performance, indicating that RAG is helpful for more complex problems.

For easy problems, all retrieval models surpass the Random baseline, showing they can capture some relevance. However, compared to the No Retrieval setting, all methods perform worse, regardless of whether the retrieved examples are semantically similar, logically aligned, or randomly sampled. This suggests that the base model is already sufficient to solve simple tasks independently, and adding reference may introduce distractions and degrade performance.

For medium problems, SolveRank may perform worse as its logic-aware retrieval focuses on complex, deep examples, which can introduce unnecessary abstraction and cognitive load for simpler tasks. A more detailed analysis of this phenomenon will be presented in Section 3.3.But for hard problems, SolveRank outperforms all other baselines, especially with GPT-4o. SolveRank yields a Pass@1 of 16.67% with GPT-4o and 11.11% with GPT-3.5, while other methods (e.g., DPR, BM25, and ReACC) offer little to no improvement compared to no retrieval. These results show that logic-equivalent examples retrieved by SolveRank aid in solving complex problems, highlighting the importance of structural alignment and deep solution in retrieval-augmented code generation.Further evaluation on the APPS dataset is provided in Appendix D.

We evaluate the ranking performance by comparing SolveRank with SOTA ranking baselines, using the retrievability of synthetic solution-relevant problems from the xCodeEval-Python training dataset as the evaluation criterion. From Table 2, we can see that SolveRank outperforms all baselines, achieving a P@1 of 0.682 and an MRR of 0.755, while BM25, ReACC, and CodeBERT have much lower MRRs (0.186, 0.057 and 0.147, respectively). BM25 focuses on lexical overlap, lead-

Table 3: Algorithm distribution across difficulty levels in xCodeEval.

Algorithm Tag	≤ 1400	1400–2000	≥ 2000
implementation	146	26	18
math	91	50	18
brute force	62	25	2
greedy	50	8	4
dp	17	19	18
constructive algorithms	15	29	1
number theory	14	11	4
strings	14	1	0
sortings	11	1	0
binary search	8	5	5
bitmasks	5	6	2
combinatorics	4	12	2
probabilities	4	6	0
shortest paths	2	0	3
graphs	2	4	4
dfs and similar	2	8	3
geometry	1	6	1
games	1	6	6
matrices	1	2	3
two pointers	1	0	0
expression parsing	1	0	0
data structures	0	12	1
divide and conquer	0	2	3
flows	0	0	13
fft	0	0	1
trees	0	0	3
graph matchings	0	1	2
meet-in-the-middle	0	1	0
string suffix structures	0	0	1
total	452	241	118

ing to irrelevant results, while CodeBERT struggles with solution-level structure. ReACC focuses on surface semantic similarity through code transformations and API usage, which is less effective for competitive programming tasks.SolveRank, through contrastive learning, captures algorithmic alignment and reasoning logic, making it more effective for competitive programming.To further illustrate this distinction, a case study is provided in Appendix B.1.

3.3 Further Analysis

While SolveRank demonstrates strong performance on difficult problems, we observe that its advantage is less prominent on medium-difficulty tasks. To better understand this phenomenon, we conduct a detailed analysis of algorithm distribution, error cases, and model behaviors.

As shown in Table 3, easy problems ($D \leq 1400$) are dominated by implementation, math, brute force, greedy.Medium problems ($1400 < D \leq 2000$) are still concentrated on math, implementation, and brute force.Hard problems ($D > 2000$) exhibit a more balanced distribution and contain many algorithm classes that are rarely present in lower levels, such as flows, FFT, trees, graph matchings, string suffix structures.

Table 4: SolveRank error distribution by problem type.

Algorithm Tag	**GPT-3.5**	**GPT-4o**
math	9	7
implementation	5	6
greedy	5	4
brute force	4	3
dfs and similar	4	2
combinatorics	3	0
dp	3	0
graphs	3	0
number theory	2	2
dsu	2	0
constructive algorithms	1	0
binary search	1	1
geometry	1	2
bitmasks	1	0
data structures	0	1

We further examine the cases where SolveRank fails but at least one semantic retriever (BM25, DPR, ReACC, or CodeBERT) succeeds. Table 4 shows that most errors occur in math, implementation, greedy, and brute force categories. This highlights a limitation of solution-aware retrieval:

For math problems, which are already highly abstract, additional reference problems provide limited benefit; success depends more on the code generator's inherent mathematical reasoning ability.For implementation and greedy problems, semantic retrievers often identify problems with similar contexts or scenarios, which can better guide the generator to simulate processes correctly. In contrast, SolveRank excels at identifying deep structural and algorithmic similarities, which are more critical in complex, high-difficulty tasks.To illustrate this, we analyze a representative medium-level case in Appendix B.2.

4 Conclusion

In competitive programming, understanding problem-solving logic is crucial. Current code generation models focus on surface-level semantics, which often fail on complex problems. This paper introduces SolveRank, a solution-aware ranking model that uses synthetic data to improve code generation performance. The model outperforms semantic-based retrievers and introduces a solution-level ranking task. Future work will explore the reinforcement learning for ranking improvement.

Limitations

Our experiments are conducted solely on the xCodeEval benchmark, which focuses on competitive programming tasks. The generalizability of our framework to broader code generation domains, such as software engineering tasks or multi-language corpora, remains to be validated in future.

Acknowledgments

This work was funded by the National Natural Science Foundation of China (NSFC) under Grants No. 62406013, the Beijing Advanced Innovation Center Funds for Future Blockchain and Privacy Computing and the Fundamental Research Funds for the Central Universities.

References

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and 1 others. 2020. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7212–7225.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and 1 others. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick SH Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense passage retrieval for open-domain question answering. In *EMNLP (1)*, pages 6769–6781.

Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. 2024. Xcodeeval: An execution-based large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6766–6805.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, and 1 others. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. Reacc: A retrieval-augmented code completion framework. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6227–6240.

Liangming Pan, Alon Albalak, Xinyi Wang, and William Wang. 2023. Logic-lm: Empowering large language models with symbolic solvers for faithful logical reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 3806–3824.

Stephen Robertson, Hugo Zaragoza, and 1 others. 2009. The probabilistic relevance framework: Bm25 and beyond. *Foundations and trends® in information retrieval*, 3(4):333–389.

Guilherme Moraes Rosa, Ruan Chaves Rodrigues, Roberto Lotufo, and Rodrigo Nogueira. 2021. Yes, bm25 is a strong baseline for legal case retrieval. *arXiv preprint arXiv:2105.05686*.

Lingxiang Wang, Hainan Zhang, Qinnan Zhang, Ziwei Wang, Hongwei Zheng, Jin Dong, and Zhiming Zheng. 2025. Codebc: A more secure large language model for smart contract code generation in blockchain. *arXiv preprint arXiv:2504.21043*.

Tianle Xia, Liang Ding, Guojia Wan, Yibing Zhan, Bo Du, and Dacheng Tao. 2025. Improving complex reasoning over knowledge graph with logic-aware curriculum tuning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 12881–12889.

Zhun Yang, Adam Ishay, and Joohyung Lee. 2023. Coupling large language models with logic programming for robust and general reasoning from text. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 5186–5219.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *International Conference on Mining Software Repositories*, MSR, pages 476–486. ACM.

Hanlin Zhang, Jiani Huang, Ziyang Li, Mayur Naik, and Eric Xing. 2023. Improved logical reasoning of language models via differentiable symbolic programming. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 3062–3077.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, and 1 others. 2023. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5673–5684.

A Prompt Design

The following presents the structured prompts used throughout our framework, including (1) generating logically equivalent problems, (2) verifying logical equivalence, and (3) constructing code generation input.

A.1 Prompt for Logically Equivalent Problem Generation

```
You are an algorithm engineer. Given the
    following problem: {{description}}

Change the background of the question and
    generate exactly 5 new questions that follow
     the same logic as the original question,
    but with different content and background.

Please follow this strict format:
- Output exactly 5 lines.
- Each line must contain exactly one question.
- Do not add any numbering, bullets, or
    explanations.

To encourage diversity:
- Use a variety of domains or themes such as
    education, logistics, art, nature,
    architecture, healthcare, etc.
- Ensure each question uses a different context
    and vocabulary.

Make sure each generated question is
    approximately as detailed and long as the
    original problem. Include any necessary
    conditions, definitions, and examples if
    appropriate. Avoid summarizing or
    oversimplifying the logic.
```

A.2 Prompt for Logical Equivalence Verification

```
Please determine whether the following two
    questions belong to the same category in
    terms of modeling logic and algorithmic
    abstraction. Focus only on their algorithmic
     modeling structure, core optimization
    objectives, and typical solution approaches.
     Ignore the specific real-world background
    or story.

If the problems are based on the same core
    abstraction (even with different story
    settings), answer "Yes". Otherwise, answer "
    No".

Question A:
{{query}}

Question B:
{{retrieved_question}}

Please answer only "Yes" or "No".
```

A.3 Prompt for Code Generation

```
Write a program in {{lang_cluster}} to solve
    this programming problem:
Description: {{description}}

{% if retrieved_context %}
Relevant examples (The following examples are
    selected based on their similarity to the
    current problem in terms of algorithmic
    modeling logic and abstraction. They share
    comparable modeling structures, core
    optimization objectives, or typical solution
     strategies. You may ignore the specific
    application context or surface narrative -
    focus instead on the underlying algorithmic
    structure and reasoning process. Use these
    examples as guidance to help generate code
    that aligns with the intended problem-
    solving logic.):
{{retrieved_context}}
{% endif %}

Input Specification: {{input_spec}}
Output Specification: {{output_spec}}

{% for input, output in zip(sample_inputs,
    sample_outputs) %}
Sample Input:
{{input}}
Sample Output:
{{output}}
{% endfor %}

Notes: {{notes}}

Take input from {{input_from}} and output to {{
    output_to}}.
Provide the {{lang_cluster}} code without any
    extra description or tokens. Target code: ||
    END-of-SRC||
```

B Case Study

B.1 Comparison of Solution-Aware and BM25 Retrieval

We present a case study in Figure 4 to intuitively illustrate the advantage of **SolveRank** in retrieving logic-equivalent problems beyond surface semantics, compared to the traditional sparse retriever **BM25**.

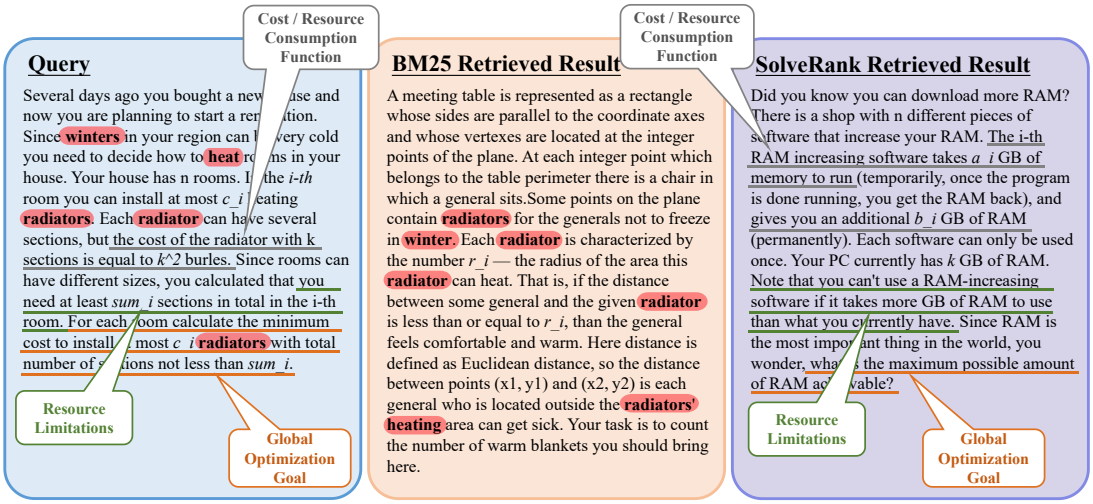

Figure 4: An example of retrieval results for a logic-intensive query. BM25 retrieves a problem with similar surface terms such as "winter", "heat", and "radiator", but diverges in algorithmic logic. In contrast, SolveRank retrieves a structurally distinct problem that shares the same underlying optimization goal and reasoning pattern, demonstrating its ability to capture logic-level similarity beyond semantics.

In the given example, the target query describes a resource allocation problem involving heating devices in multiple rooms, where the goal is to install at most c_i units in each room to achieve a required heat level sum_i, while minimizing a quadratic cost. Although this optimization task is abstract in structure, BM25 is distracted by overlapping terms such as "radiator" and "winter" in the training set and mistakenly retrieves a problem that focuses on grid coverage via Euclidean distance. The superficial term match misleads the retriever and results in a logically unrelated example that could affect subsequent generation.

In contrast, **SolveRank** retrieves a problem involving tool selection to increase memory under resource constraints—a task with a completely different surface narrative but an identical optimization structure. Both problems share the same logic: choosing from n items under a bounded constraint to reach a numeric threshold while minimizing nonlinear cost, which fits a classic dynamic programming pattern. Despite the lack of term overlap, SolveRank successfully captures this deeper alignment.

These results confirm that SolveRank can distinguish structural similarity from surface noise, enabling the retrieval of truly helpful exemplars for code generation, especially when logic alignment is more important than keyword similarity.

B.2 Medium-difficulty example: tram vs. polar bear sailing

We analyze a representative medium-level case. The "tram problem" asks Igor to reach a destination either by walking or by catching a periodically running tram, while the "polar bear sailing problem" requires computing the earliest time to reach a target under wind-driven movement. Since both problems belong to the class of time-constrained shortest reachability, SolveRank considers them logically similar. However, their structures differ: the tram problem is 1D with active boarding choices, whereas the sailing problem is 2D with passive movement:

The tram problem involves a 1D path with periodic opportunities to board.

The sailing problem involves a 2D grid with forced movement according to wind direction.

During code generation, GPT-3.5 misapplied this logic: it failed to model the tram's periodic arrivals and instead only compared the initial directions. This indicates that for medium-level tasks, directly copying reference examples may mislead weaker models. As model capability improves, however, the retrieved logic can be better adapted to the target problem.

Listing 1: Case study of medium-difficulty task: Tram vs. Polar Bear Sailing.

```
Original Question (Tram):

The tram in Berland goes along a straight line
    from the point 0 to the point s and back,
```

```
    passing 1 meter per t1 seconds in both
    directions. It means that the tram is always
     in the state of uniform rectilinear motion,
     instantly turning around at points x=0 and
    x=s.Igor is at the point x1. He should reach
     the point x2. Igor passes 1 meter per t2
    seconds. Your task is to determine the
    minimum time Igor needs to get from the
    point x1 to the point x2, if it is known
    where the tram is and in what direction it
    goes at the moment Igor comes to the point
    x1.Igor can enter the tram unlimited number
    of times at any moment when his and the tram
    's positions coincide. It is not obligatory
    that points in which Igor enter and exit the
     tram are integers. Assume that any boarding
     and unboarding happens instantly. Igor can
    move arbitrary along the line (but not
    faster than 1 meter per t2 seconds). He can
    also stand at some point for some time.

Reference Question (Polar Bear Sailing):

The polar bears are going fishing. They plan to
    sail from (sx,sy) to (ex,ey). However, the
    boat can only sail by wind. At each second,
    the wind blows in one of these directions:
    east, south, west or north. Assume the boat
    is currently at (x,y). If the wind blows to
    the east, the boat will move to (x+1,y). If
    the wind blows to the south, the boat will
    move to (x,y-1). If the wind blows to the
    west, the boat will move to (x-1,y). If the
    wind blows to the north, the boat will move
    to (x,y+1). Alternatively, they can hold the
     boat by the anchor. In this case, the boat
    stays at (x,y). Given the wind direction for
     t seconds, what is the earliest time they
    sail to (ex,ey)?
```

C Related Work

C.1 Retrieval-Augmented Code Generation

The RAG paradigm has been increasingly adopted in natural language to code (NL2Code) generation. In this setting, the model retrieves code-related knowledge (e.g., semantically similar problems or code snippets) and uses it as context to generate functionally correct programs. **BM25** (Rosa et al., 2021) is a classical sparse retrieval method based on term frequency and inverse document frequency (TF-IDF). Despite its simplicity, BM25 is commonly used as a baseline due to its high precision for short queries. **CodeBERT-Retrieval** (Feng et al., 2020) leverages the CodeBERT encoder to encode NL-code pairs, building a bi-encoder retriever to retrieve semantically similar problems based on cosine similarity of embeddings. **UniXcoder-Retrieval** (Guo et al., 2022) extends CodeBERT with unified cross-modal representations, integrating NL, AST, and code tokens for richer retrieval. It supports both encoder-only and encoder-decoder settings and has shown better performance in code-related retrieval tasks. **ReACC** (Lu et al., 2022) proposes retrieval-augmented contrastive training. It retrieves code snippets as positive contexts during training, thereby improving generalization on unseen NL2Code samples.

These methods have demonstrated improvements on CodeSearchNet (Husain et al., 2019), CoNaLa (Yin et al., 2018), and HumanEval (Chen et al., 2021). However, their retrieval strategies are predominantly based on surface-level similarity, which overlooks deeper logic algorithms.

C.2 Logical Reasoning in Programming

Logical reasoning is a critical capability for solving structured programming problems that require deeper understanding beyond surface-level semantics. Recent works have proposed combining large language models (LLMs) with symbolic reasoning systems to address this limitation. Logic-LM (Pan et al., 2023) enhances the faithfulness of reasoning by translating natural language queries into formal logic representations and solving them using symbolic solvers. Similarly, DSR-LM (Zhang et al., 2023) introduces differentiable symbolic reasoning modules into LLMs, enabling fine-grained rule induction and significantly improving performance on logic-intensive tasks. In the context of knowledge-grounded reasoning, LACT (Xia et al., 2025) applies a logic-aware curriculum tuning strategy to improve the model's ability to perform multi-hop and inductive reasoning over knowledge graphs. This approach highlights the importance of reasoning difficulty control and progressive learning in complex code understanding scenarios. Moreover, coupling LLMs with logic programming frameworks such as answer set programming has shown promising generalization capabilities (Yang et al., 2023). This hybrid paradigm allows models to abstract structural logic patterns from textual descriptions and execute them robustly, even in previously unseen settings.

Existing methods often rely on surface-level similarity when retrieving examples for code generation. In contrast, we propose **SolveRank** to capture deeper solution-level similarities, thus offering more relevant and generalizable retrievals for downstream code generation tasks.

Table 5: Results on the APPS dataset (Pass@1).

Method	Pass@1
No Retrieval	24.3%
BM25	26.1%
DPR	24.3%
CodeBERT	25.2%
SolveRank	26.1%

Table 6: Statistical comparison between original and generated problems.

Metric	Original (Avg.)	Generated (Avg.)	p-value	Significance
Prompt Length	275.977	190.867	< 0.001	Significant
Vocabulary Entropy	3.863	3.635	< 0.001	Significant
Sentence Length	26.331	23.667	< 0.001	Significant

D Generalization

We further evaluate SolveRank on the **APPS** dataset (Hendrycks et al., 2021). As shown in Table 5, SolveRank achieves the highest Pass@1, tied with BM25 . A key difference from xCodeEval (Table 3) is that APPS problems are generally shorter, with weaker narrative background and a direct focus on input–output specifications. In this setting, semantic retrieval is already sufficient to capture the key information, so the advantage of logic-aware retrieval in filtering narrative noise and aligning solution structures is less apparent.

E Quality of Synthetic Data

To assess the quality of synthetic positives, we compare them with real problems in terms of prompt length, vocabulary entropy, and average sentence length. As shown in Table 6, all three metrics differ significantly ($p < 0.001$), confirming that generated problems exhibit diverse phrasing and syntax. Despite this distributional shift, the generated data remain logically consistent and suitable for training solution-level retrieval models.