Olympiads in Informatics, 2025, Vol. 19, 129–144

DOI: 10.15388/ioi.2025.09

OI-Assistant: A Retrieval Augmented System for Similar Problem Discovery and Interactive Learning in Competitive Programming

Yuhua SU[1,*], Ping NIE[2], Xin MENG[2]
[1]*International School Altdorf, Altdorf, Switzerland*
[2]*Peking University, Beijing, China*
e-mail: suyuhuahz21@gmail.com, ping.nie@pku.edu.cn, 1601214372@pku.edu.cn

Abstract. Competitive programming (CP) often requires quickly identifying relevant problems and solutions, yet current online judge (OJ) platforms offer only limited keyword or tag-based search. This makes it difficult for contestants and coaches to find past problems with similar patterns or concepts, hindering efficient practice and problem-solving.

We introduce OI-assistant, the first intelligent problem search and solution assistant based on Retrieval Augmented Generation (RAG) to bridge this gap. The proposed OI assistant provides insightful and similar problems based on our curated problem database, detailed and structured code explanation, and interactive code validation and follow-up chat.

We first collected over 11,000 programming problems from Luogu, a widely-used Chinese platform. Then we use multiple embeddings and llm-based ranker to retrieve and rank semantically similar CP problems based on user queries or code snippets. An LLM generates context-aware responses, like related problem suggestions or solution summaries, enabling more accurate discovery than traditional keyword-based searches. Additionally, it validates these solutions by automatically generating test cases, validating code in real-time and providing educational improvement feedbacks.

The proposed RAG-based search engine significantly improves the precision and recall of finding relevant problems, as evidenced by enhanced search results in our preliminary tests. When we introduced OI-Assistant to competitive programming students, their feedback was overwhelmingly positive. They rated the similar-problem recommendations highly and particularly appreciated the clear algorithm visualizations and real-time validation and improvement feedbacks. Overall, students found our platform significantly more helpful compared to traditional OJ systems or standalone ChatGPT.

By simplifying the discovery of related practice problems and enhancing real-time interactive learning, OI-Assistant significantly improves the effectiveness of competitive programming training and opens up new possibilities for the community.

Keywords: Competitive programming, Retrieval Augmented Generation, Large Language Model

* Corresponding author

1. Introduction

Competitive programming (CP) has grown remarkably worldwide in recent decades. Back in 2000, the International Olympiad in Informatics in Beijing had 278 participants from 72 countries. Last year's IOI in Egypt drew 362 participants from 91 countries. Similarly, national contests like the USACO Open have seen participation quadruple in just ten years. This surge in interest has led to a boom in problem repositories and online judge (OJ) platforms where competitors hone their skills.

Current OJs have a major weakness: they can't effectively search for problems based on their underlying mathematical concepts (C.R.A.C. Generation, 2024; Sollenberger *et al.*, 2024). Simple keyword searches miss the importance of CP problems, especially CP problems typically are wrapped in stories to hide their core algorithmic challenges. The search options on popular platforms like Codeforces, LeetCode, and Luogu only let you filter by basic methods like difficulty, algorithm tags, or problem sources. These methods often return too many irrelevant results, forcing users to spend time manually filtering through them. The problem gets worse because algorithm tagging is wrong or inconsistent across platforms.

We created OI-Assistant to solve these issues. Our system uses Retrieval Augmented Generation (RAG) specifically designed for competitive programming education (Lewis *et al.*, 2020; Shao *et al.*, 2025). It brings three key innovations. First, we built a rich database with over 11,000 quality problems from Luogu, one of the most popular competitive programming sites. This extensive collection gives our similarity search a strong foundation, helping students find inspiration from problems with related patterns. Second, we developed a multi-pronged search approach using three different strategies: question vector search, concept vector search, and summary vector search (Lewis *et al.*, 2020). We enhance these parallel searches with an LLM-based reranker that significantly boosts result quality. Our tests show this method achieves over 80% recall when finding similar problems – much better than traditional keyword searches. Third, OI-Assistant generates comprehensive solutions that include detailed algorithm explanations, clear flowcharts, and a real-time code validation system (Kumar, 2025; Fakhoury *et al.*, 2024). This validation feature uses GPT to automatically create test cases that cover even edge scenarios (Sollenberger *et al.*, 2024). Users can run their code right in our platform, get immediate feedback, and receive helpful suggestions when errors occur (Zhou *et al.*, 2024; Nicol and Macfarlane-Dick, 2006). The system can also regenerate improved solutions that address specific issues, creating a feedback loop that enhances learning (Zhang *et al.*, 2024).

Our system builds on recent advances in large language models. As our experiments show, modern GPT models like O3-mini can score at bronze medal levels in IOI competitions even without retrieval augmentation (El-Kishky *et al.*, 2025). But OI-Assistant's real value isn't about beating these baseline capabilities – it's about educational impact (Marouf *et al.*, 2024; Alyoshyna, 2024).

By combining strong problem-solving abilities with our innovative retrieval system, OI-Assistant creates a powerful learning tool that helps students find relevant historical problems matching their current needs (Kazemitabaar *et al.*, 2024; Denny

et al., 2023). Our retrieval system's 80%+ recall rate ensures students efficiently find appropriate practice materials, while the real-time validation provides the immediate feedback crucial for learning (Price *et al.*, 2016; Price *et al.*, 2017). User studies confirm that this feature combination significantly improves problem-solving skill development and learning efficiency compared to traditional approaches (Anderson *et al.*, 1995; Marouf *et al.*, 2024).

2. Related Work

2.1. *LLMs in Computer Science Education*

Generative AI is changing how we teach computer science. Researchers are exploring ways to use Large Language Models (LLMs) in educational settings, with promising results for helping students learn programming (Kazemitabaar *et al.*, 2024; Zhang *et al.*, 2024; Alyoshyna, 2024). Recent studies show LLMs can solve programming problems quite well. Denny and colleagues (Denny *et al.*, 2023) tested GitHub Copilot on 166 programming problems. It solved about half of them on the first try. With better prompts, that success rate jumped to 60% for the remaining problems. Even more impressive, OpenAI's ChatGPT-o3 earned a gold medal at the 2024 International Olympiad in Informatics and achieved a rating on Codeforces similar to top human competitors (El-Kishky *et al.*, 2025). These results show that today's best LLMs can perform at high levels in competitive programming.

LLMs can do more than just solve problems – they can create educational content too. Kazemitabaar and team (Kazemitabaar *et al.*, 2024) built CodeAid, a coding assistant based on ChatGPT. It has six functions, including writing code, explaining concepts, and fixing errors. They tested it with 700 students over a full semester. The feedback was mostly positive. This shows how carefully designed prompts can make LLMs much more useful for teaching. These studies lay the groundwork for using LLMs in programming education.

Fakhoury (Fakhoury *et al.*, 2024) and Sollenberger (Sollenberger *et al.*, 2024) shows these models can generate test cases and check if code is correct. This creates interactive learning environments that both generate and validate code. Zhang (Zhang *et al.*, 2024) studied what students want from AI feedback. They found that detailed explanations in context are what students value most when learning to program.

2.2. *Current Online Judge Status*

As shown in Table 1, Online Judge (OJ) platforms have grown from simple grading tools into full-fledged competitive programming communities (El-Kishky *et al.*, 2025; C.R.A.C. Generation, 2024). Today's platforms like Codeforces let users join contests, participate in forums, and study other coders' solutions. Luogu offers similar features.

Table 1
Different OJ Platforms

OJ	Search by difficulty	Algorithm	Source	Title search	Content search	Similarity search
Codeforces	800–3500	37 parallel tags	Yes	No	No	No
LeetCode	3 categories	71 parallel tags	Yes	Yes	No	No
Luogu	7 categories	Tags in 22 categories	Yes	Yes	Yes	No
USACO	7 categories	162 parallel tags	Yes	Yes	No	No
SPOJ	Rating for conceptual and implementational difficulties	119 Tags	Yes	No	No	No
UVa	NO	No	Yes	No	No	No

These platforms have become essential for competitive programmers to learn and connect with others.

Despite these improvements, OJs still struggle with organizing problem databases and providing good search tools, even though users really want these features (C.R.A.C. Generation, 2024; Sollenberger *et al.*, 2024). We surveyed six major OJ platforms and found several patterns in how they handle searching:

- **Search by origin:** Almost all OJs tag problems by where they came from, as shown in Table 1. This lets users filter problems by specific competitions or sources. Only SPOJ lacks this kind of search help.
- **Search by difficulty:** Most OJs group problems by how hard they are, but they do this differently. Codeforces uses numbers from 800–3500, while SPOJ separates difficulty into concept and implementation scores based on user votes.
- **Search by algorithms:** Algorithm tagging varies widely across platforms. Codeforces, LeetCode, and USACO use flat tag structures with no hierarchy. Luogu offers better organization with 22 algorithm categories, each containing related tags. SPOJ has the most sophisticated approach with a tree-structured system. All OJs support tag searches, but the differences between platforms make things confusing for users.
- **Search by content & problem similarity:** The biggest gap is in content-based searching (Lewis *et al.*, 2020; Asai *et al.*, 2024). While some OJs let you search by problem title or text, none offer true similarity search based on the underlying math concepts. This is a serious problem since competitive programming tasks usually come wrapped in stories during contests like IOI, ICPC, and Codeforces, which makes keyword searching pretty useless.

2.3. *Retrieval-Augmented Generation*

LLMs can do amazing things, but they still struggle with making up information and having outdated knowledge, especially in specialized fields (El-Kishky *et al.*, 2025; Tang *et al.*, 2024). Retrieval-Augmented Generation (RAG) solves these problems by

combining external information lookup with LLM generation. This greatly improves accuracy and relevance (Lewis *et al.*, 2020; Asai *et al.*, 2024).

For programming tasks, Wang and team created CODERAG-BENCH (C.R.A.C. Generation, 2024), a benchmark for testing how well RAG works for coding. Their research shows that RAG-enhanced models consistently beat regular LLMs, especially when tasks need external libraries. This matters for competitive programming, where specialized algorithms are often needed.

Shao's team (Shao *et al.*, 2025) looked at the effects of scale in retrieval-based language models with their MassiveDS project, which has a massive 1.4 trillion-token database. They found that bigger datastores consistently improve performance across language modeling and various tasks. Interestingly, their smaller models with large datastores outperformed bigger models without retrieval in knowledge-heavy tasks. This finding applies directly to algorithm-intensive competitive programming.

In scientific applications, Asai and colleagues (Asai *et al.*, 2024) developed OpenScholar, a retrieval-enhanced model for synthesizing scientific literature. By pulling relevant passages from open-access papers, OpenScholar reduced fake citations and beat larger models like GPT-4 in factual accuracy. This shows how RAG can improve precision in technical areas.

Collectively, these studies establish that RAG significantly enhances LLM performance in tasks requiring current, domain-specific knowledge (Lewis *et al.*, 2020; Shao *et al.*, 2025; Asai *et al.*, 2024). This approach proves especially valuable for competitive programming applications, where precise algorithmic understanding and accurate code generation are essential. Implementing RAG frameworks in this context can substantially improve solution retrieval, problem explanation, and code generation capabilities (C.R.A.C. Generation, 2024; Tang *et al.*, 2024; Zhou *et al.*, 2024).

3. Framework

Our OI-Assistant helps students find the right practice problems and understand their solutions (Kazemitabaar *et al.*, 2024; Marouf *et al.*, 2024). Fig. 1 shows how our system works. It has four main parts: Data Construction, Backend Search, Solution Generation, and Frontend Display.

3.1. *Data Construction*

We started by building a large collection of programming problems (Shao *et al.*, 2025; Asai *et al.*, 2024). We created web crawlers to gather problems from Luogu, a popular Chinese programming platform. We collected over 11,000 problems along with their details and more than 10,000 community solutions.

Programming contests often present problems as stories (El-Kishky *et al.*, 2025). While these narratives make problems interesting, they hide the core math concepts,

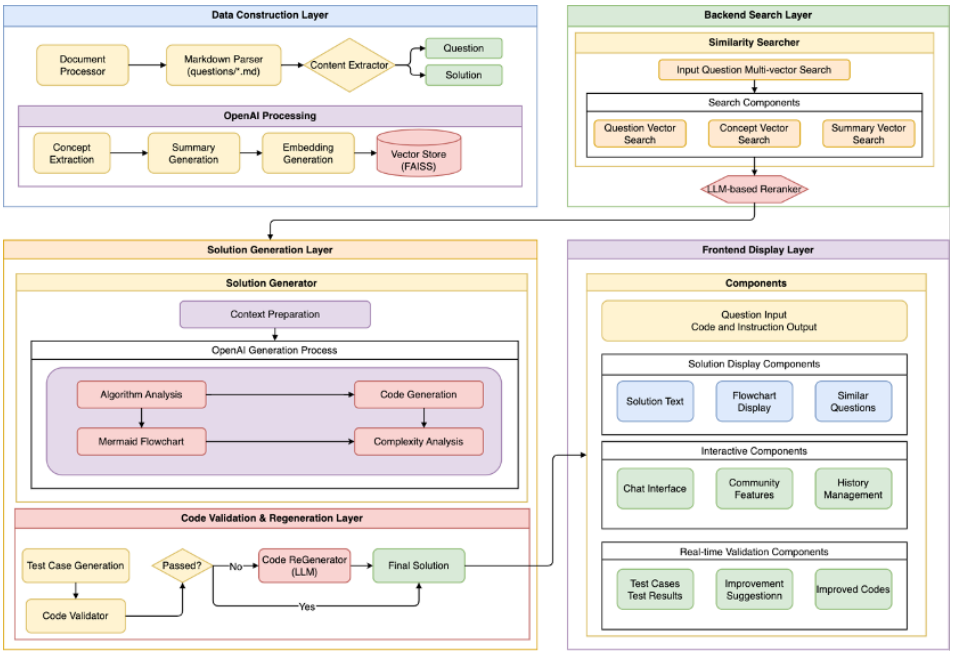

Fig. 1. OI-Assistant Framework.

which makes keyword searches pretty useless. We solved this by using large language models with carefully designed prompts to extract the mathematical essence from each problem, separating the algorithm from the story (Denny *et al.*, 2023; El-Kishky *et al.*, 2025).

We also use Luogu's tagging system, which groups problems by categories like Dynamic Programming, Graph Theory, Math, and Data Structures. Each category has more specific tags – for example, Dynamic Programming breaks down into 1D DP, Interval DP, Tree DP, and so on. Luogu rates problem difficulty on a 7-level scale:

Level 1: Beginner problems teaching basic concepts
Level 2: Easy problems needing simple algorithms
Level 3: Medium problems combining multiple ideas
Level 4: Hard problems requiring complex algorithms
Level 5: Provincial competition level
Level 6: National competition (NOI) level
Level 7: International Olympiad (IOI) level

After cleaning the data, each problem has a standard format with the problem statement, algorithm tags, difficulty level, and solution. For each problem, we create three different vector embeddings using OpenAI's API: Statement embedding that captures the math concepts, Concept embedding that represents the algorithms needed, Solution embedding that encodes how to implement the answer. We store these in FAISS indexes so we can search them quickly.

3.2. *Assistant Response Generation*

Finding similar problems is helpful, but students also need to understand how to solve them (Kazemitabaar *et al.*, 2024; Zhang *et al.*, 2024. As shown in Fig. 1 (bottom left), our Solution Generation Layer transforms a user's problem query into a comprehensive educational solution through several interconnected stages.

3.2.1. *User Input Processing and Retrieval.*

When a user submits a problem, the system first processes it through the Backend Search Layer shown in Fig. 1 (top right). The input question undergoes multi-vector search, which splits into three parallel components:

1. **Question Vector Search** – finds problems with similar statement structures,
2. **Concept Vector Search** – identifies problems using similar algorithmic techniques,
3. **Summary Vector Search** – locates problems with similar high-level approaches

The results from these three search components feed into the LLM-based Reranker (shown in the pink oval in Fig. 1), which evaluates each candidate problem in context and identifies the most top 50 relevant matches.

3.2.2. *Solution Generation Process*

As depicted in Fig. 1 (bottom left), the Solution Generator begins with Context Preparation. This crucial step combines the original problem with the retrieved similar problems and their solutions from our database. This context gives our system concrete examples of approaches that worked for similar challenges. The OpenAI Generation Process (shown in the central box) consists of four key components working in tandem:

1. **Algorithm Analysis:** (shown in the pink box) – The system creates a detailed explanation of the solution approach, key insights, and reasoning steps (Zhang *et al.*, 2024). This forms the conceptual foundation of the solution.
2. **Mermaid Flowchart:** (shown in the pink box) – In parallel, the system generates a visual flowchart using Mermaid syntax. This visualization helps students understand the algorithm's workflow intuitively, making complex concepts easier to grasp.
3. **Code Generation** (shown in the pink box): The system produces implementation code in the student's preferred programming language based on the algorithm analysis.
4. **Complexity Analysis** (shown in the pink box): The system explains the time and space complexity of the solution, helping students understand efficiency considerations (El-Kishky *et al.*, 2025).

3.2.3. *Code Validation and Regeneration*

What makes our system especially valuable is the Code Validation & Regeneration Layer shown at the bottom of Fig. 1 (bottom left). This layer includes: 1. Test Case Genera-

tion: The system automatically creates diverse test cases covering both normal scenarios and edge cases (Sollenberger *et al.*, 2024). 2. Code Validator: These test cases are fed into the Code Validator, which executes the generated code against the tests. 3. Feedback Loop: If the code passes all tests (the "Yes" path in Fig. 1), the system produces the Final Solution. If any tests fail (the "No" path), the Code Regenerator (LLM) analyzes what went wrong and creates an improved solution.

This validation-regeneration cycle continues until a correct solution is found, mimicking how a teacher might guide a student through debugging (Anderson *et al.*, 1995; Price *et al.*, 2017). When tests fail, the system doesn't just flag errors – it analyzes the specific issues and suggests targeted improvements (Zhou *et al.*, 2024; Nicol and Macfarlane-Dick, 2006).

3.3. *Frontend Interface*

As shown on the right side of Fig. 1, the Frontend Display Layer shows the system's functions through nice user interface. Fig. 2 and Fig. 3 illustrate actual screenshots of the UI interface.

3.3.1. *Solution Display Components*

The middle right side of Fig. 1 highlights components of the solution display:

- **Solution:** Clearly shows the algorithm and codes using clean description.
- **Flowchart:** Provides visual diagram that illustrates the algorithm's workflow. This is helping students learn the solution process easier.
- **Similar Questions:** gives related practice problems from our database. This is helping students recognize patterns and connections across multiple problems (Price *et al.*, 2016). Each similar problem also has a reason of its relevance to the current user input.

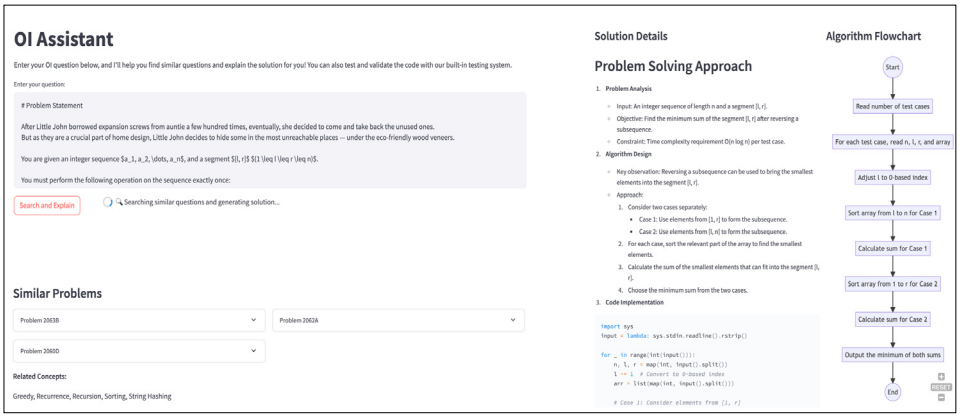

Fig. 2. OI-Assistant Frontend for Response.

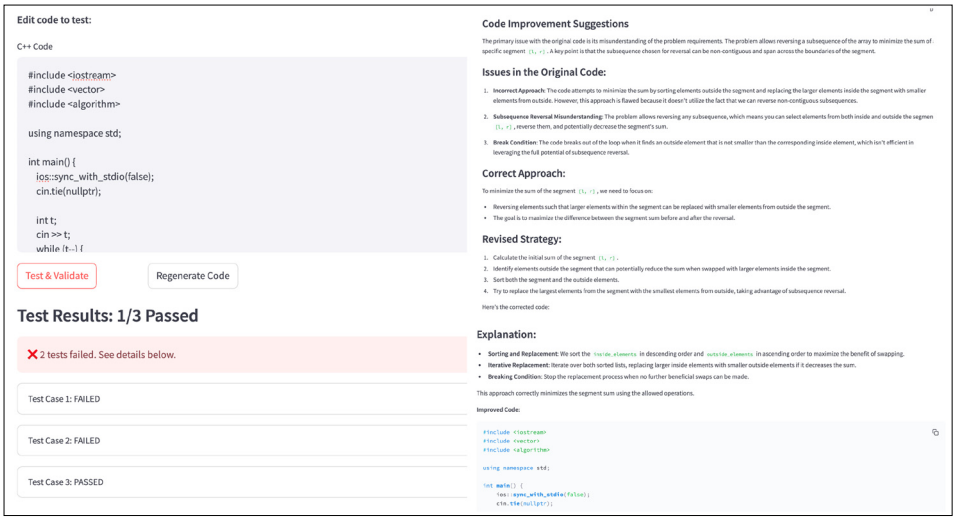

Fig. 3. OI-Assistant Frontend for Real-time Code Validation and Re-generation.

3.3.2. *Interactive Components*

Fig. 2 shows the OI-Assistant's frontend response and Fig. 3 shows the real-time code validation and re-generation feature. As shown in Fig. 2, Student can input the question statement or some code snippts to ask our system, our system will search our database to find similar statements and solutions. Also, the concepts for the user input are also displayed. Then on the right side of Fig. 2, the system will teach the student to understand the problem by text descriptions and a flowchart. After the flowchart, there is a code for the input question. In Fig. 3, there is a code box whose default code is from the system generation. The user can also edit the code box. Then the user can click the button to validate the code in the box. Our system will on the fly generate the test cases for the user input statement and the system will execute the code to check if the code can cover all generated test cases. If some test cases are not passed, the system will again look at the code and failed cases to generate some suggestions for improvement. The system will also output improved codes. The user can copy the improved code or input their own new code back to the code box to validate. This system then provides a super useful interactive process for learning.

4. Experiments and Evaluation

We built a complete evaluation framework to test OI-Assistant. Our tests cover everything from basic model abilities to how users feel about the system (Kazemitabaar *et al.*, 2024; Fakhoury *et al.*, 2024). Each experiment shows how different parts work together to create an effective learning tool.

4.1. *LLM Performance on IOI 2024*

As Fig. 4 shows, all models get better results when they try more times. The O3-mini model starts with okay performance. Yet with enough attempts, it reaches scores similar to human bronze medalists (around 215 points). This matters for real-world use. Even smaller models can do well if you let them try multiple times (El-Kishky *et al.*, 2025; Zhou *et al.*, 2024). We found that performance tends to level off after 10–20 attempts, suggesting this is a practical limit by computation.

Our system needs LLMs that can solve programming problems well (El-Kishky *et al.*, 2025). We tested several GPT models on IOI-2024 problems using methods from the Hugging Face IOI repository. One key question: how does performance improve with multiple solution attempts?

4.2. *Dataset Characteristics and Analysis*

After we know we can get good performance with SOTA LLMs, we next check our own dataset. This knowledge base powers our retrieval system, so it directly affects performance.

Our collected dataset includes 11,000 competitive programming problems from Luogu, one of China's busiest programming platforms. These problems come with 11,000

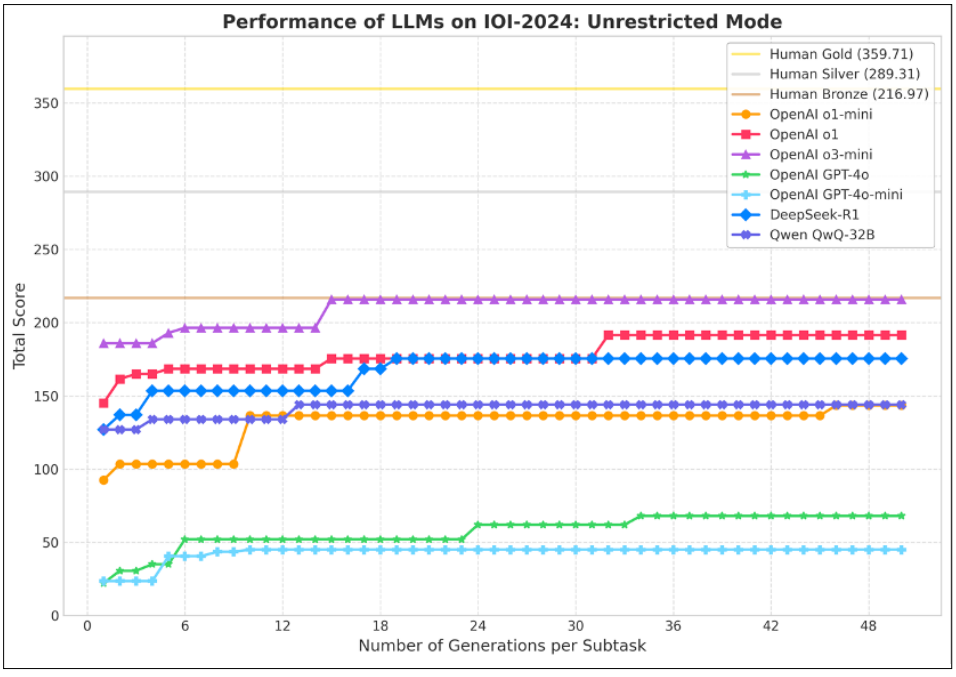

Fig. 4. OpenAI models performance on IOI 2024 with multiple generations.

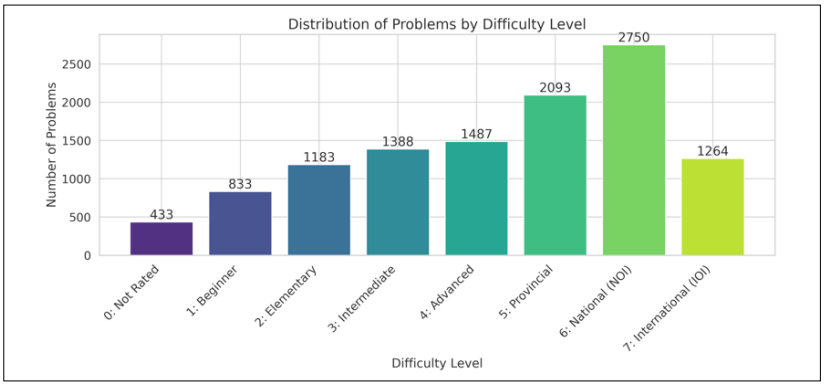

Fig. 5: Number of Problems for Difficulty Level.

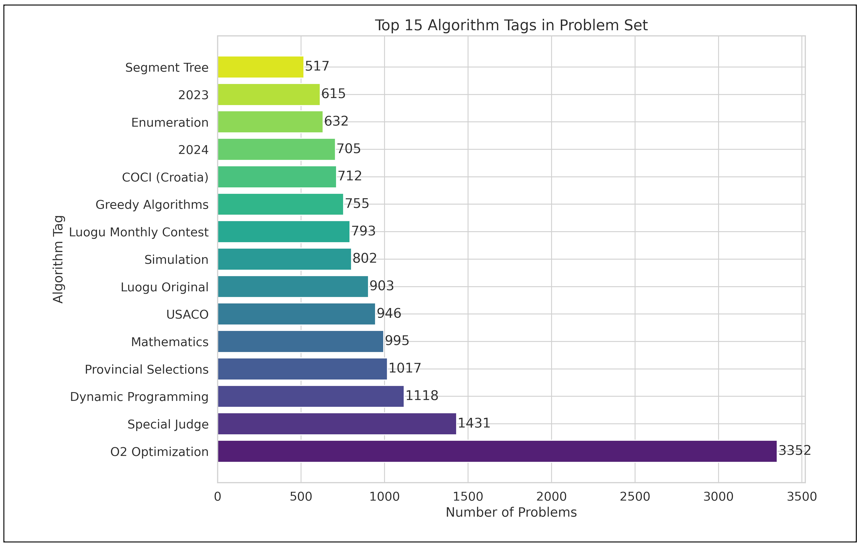

Fig. 6: Popular Tags in the dataset.

high quality solutions that have strong community engagement such as upvotes. We analyzed this dataset from several aspects to understand its educational value.

As shown in Fig. 5, we can see our dataset is balance for different difficulty level. Data difficulty definition can be found in section 3.1. Each difficulty level has about or more than 1000 questions. NOI questions are more than 2500. In Fig. 6, we can check the top 15 popular luogu tags for those questions in our datasets. The tags in Fig. 6 is from luogu to show it's real data features. Dynamic Programming and provincial selections are popular in our datasets. In Fig. 7, we can see the relationship between the acceptance and the difficulty of the luogu data. When the difficulty increases, the acceptance is dropping. And most of those questions are submitted by students for more than 10k times. This shows our dataset's meaningful status of helping those students to learn CP.

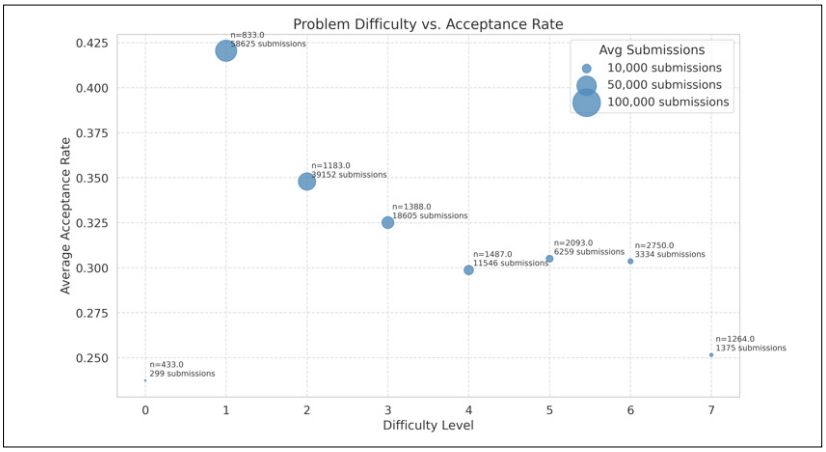

Fig. 7: Relationship between Difficulty and Acceptance Rate.

4.3. *Retrieval System Performance*

Building on our understanding of LLMs and dataset quality, we tested our multi-vector retrieval system. Traditional keyword searches often miss the deeper connections between problems, especially when similar challenges come wrapped in different contexts.

As shown in Fig. 8, we tested our system with three different settings to confirm our system can search and find similar problems given a new coding problem. Substring setting means the input question is a random substring of the existing problems in the datasets. Substring could be the problem definition or the solution to the existing problems. LLM Rewrite setting means we use GPT-4o to rewrite the existing questions with a different story to wrap the question with the same mathematic logics. Code Snippet setting means we use GPT-4o to generate an code answer for each existing problem. So with those three settings, we can comprehensively test our system's retrieval ability for both natural language and code input.

We created a dataset with 300 user input for each setting and 900 user input in total. We evaluated our 3 embedding retrievers and 1 llm ranker's recall performance. As shown in Fig. 8, the LLM ranker always gives the highest recall for three setting for all recall@K, where K = 10, 30, or 50. When we retrieve 50 candidates, the LLM ranker can achieve 80%+ recall for all settings. For the embedding retrieval, the Question representations gives the best score for about 69% at top 59% for the substring setting. For the code snippet settings, the recall drops dramatically to 30% which means it's hard for the embedding vectors to capture the semantic meaning of code. However, the Summary of the input can still keep decent recall at about 34% which shows the multiple vector embedding's advantages. After combining the results from three vector search, the LLM ranker can always get 80%+ recall at any settings. This shows the robustness of the LLM ranker. It also provide reliable similar problems output for our system.

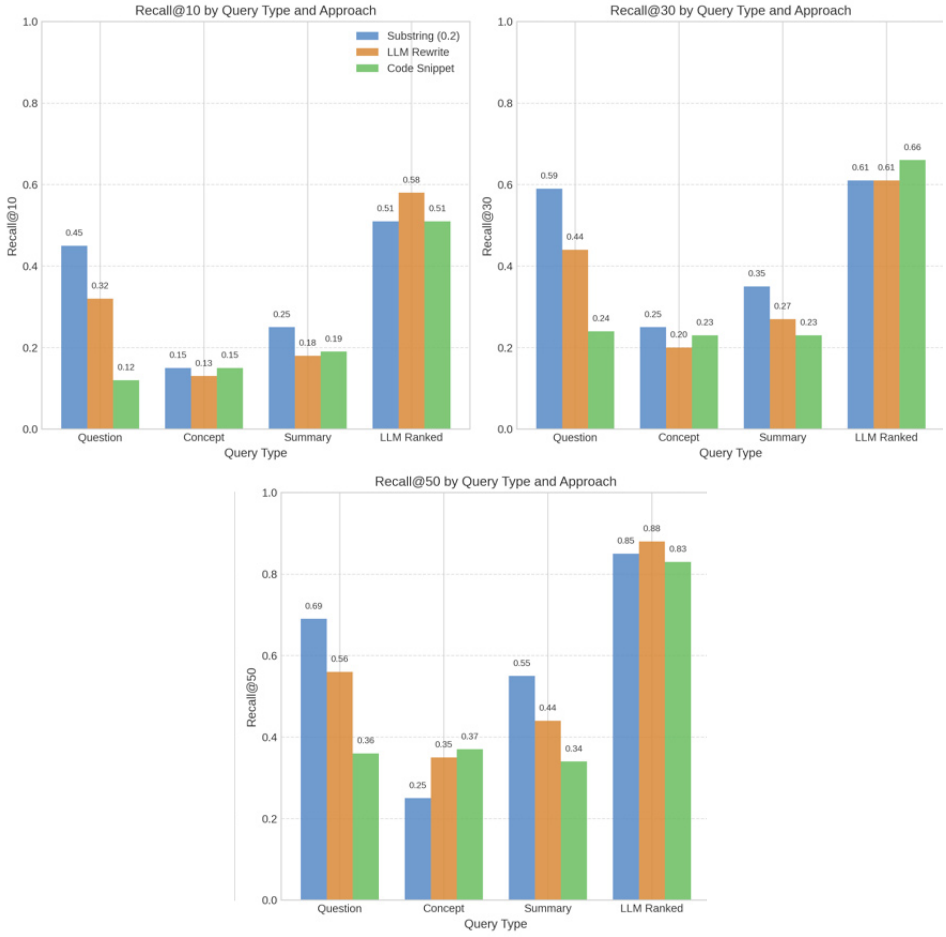

Fig. 8: Recall Performance of different modules on different settings.

4.4. *User Experience Evaluation*

After we confirm our system has good performance with multiple different recall testing. We conducted a user study with 36 competitive programming students who compared OI-Assistant with alternatives (ChatGPT and Luogu).

As shown in Table 2, our system outperformed both ChatGPT and luogu across all dimensions. Students especially give high ratings for our ability of finding similar problems (rated 4.8/5 compared to just 3.1/5 for ChatGPT and 2.8/5 for traditional platforms). This confirms that our focus on finding similar questions makes a real difference.

As shown in Table 3, we also show the system's helpfulness for different modules. students liked the similar problem recommendations (4.8/5), Code validation and feedback (4.9/5). These features create a comprehensive CP learning experience. Students can first understand solution approaches through clear explanations and visuals. Then

Table 2
User Ratings for Different Platforms

System Aspect	OI-Assistant	ChatGPT	Luogu
Solution quality	4.5	3.8	2.6
Finding similar problems	4.8	3.1	2.8
Overall Helpfulness	4.5	3.5	2.1

Table 3
User Ratings for Different Modules

OI-Assistant Feature	Average Rating (1–5)	Standard Deviation
Algorithm analysis quality	4.7	0.4
Flowchart visualization	4.6	0.5
Code validation and feedback	4.9	0.3
Similar problem recommendations	4.8	0.5
Interactive follow-up capability	4.1	0.6
LLM-generated test cases	4.3	0.3

they can improve their learning by using with similar problems found by our retrieval system and the real-time testing system with automatically generated test cases. The Overall helpfulness is also high for our system. Those real user experience feedbacks and ratings confirm our system's educational value.

5. Limitations and Future Work

Our OI-Assistant shows promise, but it has some clear limitations. Let's look at what could be better.

First, the system needs lots of computing power. This makes widespread deployment challenging. The LLM-based reranker that gives us great results also adds delays that users notice. When we generate multiple solutions to find the best one, we need even more computing resources. Not all schools or learning centers can provide this kind of cost. These issues matter most when trying to use the system in places with limited resources or when scaling up to many users.

Our dataset has its own limitations. We only used problems from Luogu. Programming problems often contain cultural references that might confuse users from different backgrounds. Also, different programming communities create problems in their own unique ways.

In the future, we could try model distillation to create smaller, faster versions of our reranker without losing much performance. Better search techniques, like hybrid search and small model ranking. Smart caching for common problem patterns would speed up responses for frequently asked questions.

We also want to expand what our database Adding more competitive programming platforms data would provide a richer knowledge base with diverse problem-solving approaches. Supporting more programming languages would help more students use the system, especially in schools that teach specific languages.

6. Conclusion

We developed OI-Assistant to address a significant challenge faced by competitive programming students and coaches that finding similar problems with similar concepts not just by tags is super useful for student learning. By leveraging Retrieval Augmented Generation (RAG), our system achieves promising results. Experiments demonstrate over 80% recall for retrieving similar questions even for code snippets. Users confirmed that our integrated features including solution generation, algorithm explanations, flowchart, and real-time code validation, greatly help their learning process. Our work shows a great future for competitive programming education. OI-Assistant doesn't just help students find problems; it strengths their understanding by connecting similar code problems and providing real-time feedback for any coding problems with automatic test cases and validation. This approach builds stronger thinking, helping students recognize patterns across diverse problems and figure out the errors in the code by test cases driven way and educational feedback. As language models continue to evolve, combining RAG with them will become increasingly valuable for specific knowledge base, making it easier for students to learn complex concepts and do better in competitive programming.

References

Alyoshyna, Y. (2024). AI in Programming Education: Automated Feedback Systems for Personalized Learning. *University of Twente Student Theses*.

Anderson, J.R., Corbett, A.T., Koedinger, K.R., and Pelletier, R. (1995). Cognitive tutors: Lessons learned. *The Journal of the Learning Sciences*, 4(2), 167–207.

Asai, A., He, J., Shao, R., Shi, W., Singh, A., Chang, J.C. *et al.* (2024). OpenScholar: Synthesizing scientific literature with retrieval-augmented LMs. arXiv preprint arXiv:2411.14199.

C.R.A.C. Generation. (2024). *CODERAG-BENCH: Can Retrieval Augment Code Generation?*

Corbett, A.T. and Anderson, J.R. (1992). The LISP intelligent tutoring system: Research in skill acquisition. In: *Computer Assisted Instruction And Intelligent Tutoring Systems: Establishing Communication and Collaboration*. Lawrence Erlbaum Associates, Inc, 141–194.

Denny, P., Kumar, V., Giacaman, N. (2023). Conversing with Copilot: Exploring prompt engineering for solving CS1 problems using natural language. In: *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 1136–1142.

El-Kishky, A., Wei, A., Saraiva, A., Minaev, B., Selsam, D., Dohan, D., *et al.* (2025). *Competitive Programming with Large Reasoning Models*. *arXiv preprint arXiv:2502.06807*.

Fakhoury, S., Naik, A., Sakkas, G., Chakraborty, S., and Lahiri, S.K. (2024). LLM-Based Test-Driven Interactive Code Generation: User Study and Empirical Evaluation. *IEEE Transactions on Software Engineering*.

Georgia Department of Education. (2020). *Georgia's ReStart: Embrace, Engage, Expand, and Enhance Learning with Technology (GRE4T) Initiative*.

Kazemitabaar, M., Ye, R., Wang, X., Henley, A.Z., Denny, P., Craig, M., Grossman, T. (2024). CodeAid: Evaluating a classroom deployment of an LLM-based programming assistant that balances student and educator needs. In: *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*. 1–20.

Kira Learning. (n.d.). *The AI platform for schools*.

Kumar, S. (2025). Teaching LLMs to generate Unit Tests for Automated Debugging of Code. *Medium*.
Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., *et al.* (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. *Advances in Neural Information Processing Systems*, 33, 9459–9474.
Marouf, A., Al-Dahdooh, R., Abu Ghali, M.J., Mahdi, A.O., Abunasser, B.S., and Abu-Naser, S.S. (2024). Enhancing Education with Artificial Intelligence: The Role of Intelligent Tutoring Systems. *International Journal of Engineering and Information Systems (IJEAIS)*, 8(8), 10–16.
Nicol, D.J. and Macfarlane-Dick, D. (2006). Formative assessment and self-regulated learning: A model and seven principles of good feedback practice. *Studies in Higher Education*, 31(2), 199–218.
Price, T.W., Dong, T., and Barnes, T. (2016). Generating data-driven hints for open-ended programming. In: *International Conference on Educational Data Mining*. 446–451.
Price, T.W., Zhi, R., and Barnes, T. (2017). Evaluation of a data-driven feedback algorithm for open-ended programming. In: *International Conference on Educational Data Mining*. 530–535.
Psotka, J., Massey, L.D., and Mutter, S.A. (Eds.). (1988). *Intelligent Tutoring Systems: Lessons Learned*. Lawrence Erlbaum Associates, Inc.
Shao, R., He, J., Asai, A., Shi, W., Dettmers, T., Min, S. *et al.* (2025). Scaling Retrieval-Based Language Models with a Trillion-Token Datastore. *Advances in Neural Information Processing Systems**, 37, 91260–91299.
Shi, X., Tian, M., and Zhang, J. (2022). A summary of personalized learning research. In: *IET Conference Proceedings*. Vol. 2022, No. 9, 53–58.
Sollenberger, Z., Patel, J., Munley, C., Jarmusch, A., and Chandrasekaran, S. (2024). LLM4VV: Exploring LLM-as-a-Judge for Validation and Verification Testsuites.
Tang, H., Hu, K., Zhou, J.P., Zhong, S.C., Zheng, W.L., Si, X., and Ellis, K. (2024). *REx: An Exploration-Exploitation Framework for LLM-Based Code Refinement*. arXiv preprint arXiv:2411.14199.
VanLehn, K. (1988). Student modeling and mastery learning in a computer-based programming tutor. In: *Intelligent Tutoring Systems*. Springer, Berlin, Heidelberg, 479–506.
Zhang, Z., Cheng, L., and Chen, X. (2024). Students' Perceptions and Preferences of Generative Artificial Intelligence Feedback for Programming. *Journal of Educational Computing Research*, 71(4), 647–673.
Zhou, Y., Peng, X., Zeng, A., Xie, Q., and Luo, T. (2024). *LLMFix: Automatically Fixing Code Generation Errors in Large Language Models*. arXiv preprint arXiv:2409.00676.

Y. Su – a senior high student at the International School Altdorf in Switzerland. With four years of competitive programming experience, he won a gold medal in the Swiss Olympiad in Informatics. His research interest focuses on machine learning and human-computer interaction. His past projects include building a stuttering recognition system and enhancing a micro-expression spotting network.

P. Nie – a Senior Applied Scientist with a Master's degree from Peking University. His research interests lie in Code Large Language Models, Information Retrieval, and Natural Language Processing. He also serves as a program committee member for conferences such as ACL, SIGIR, and NeurIPS.

X. Meng – a Senior Deep Learning Engineer with a Master's degree from Peking University. He has 10 years of experience in AI development, like CUDA acceleration, Parallel Computing. His research interests lie in Computer Vision and Autonomous driving with LLM, Robots, Embodied Intelligence.