Proceedings of CONF-MLA 2025 Symposium: Intelligent Systems and Automation: AI Models, IoT, and Robotic Algorithms
DOI: 10.54254/2755-2721/2025.LD28518

A Collaborative Framework of Knowledge Graphs and Large Language Models for the Purpose of Carrying Out the Process of Solving Algorithmic Problems

Yukai Wu

School of Computer and Information Technology, Beijing Jiaotong University, Beijing, China
23281023@bjtu.edu.cn

Abstract. The process of solving algorithm problems is a fundamental task within the realm of computer science education and competitive programming. However, in the situation where existing large language models (LLMs) are faced with the task of solving complex problems and generating correct as well as efficient solutions under the condition of strict time and space constraints, they often encounter difficulties. There is still a substantial amount of room that can be improved. In this piece of work, we put forward a new collaborative framework. This framework combines the knowledge graph and the LLM in order to further enhance the capability of dealing with the algorithm problem. In this work, we select 30 challenging algorithmic problems from the Luogu Online Judge platform. We ensure that these problems cover a wide variety of algorithmic categories so as to evaluate the effectiveness of the method that we have proposed. The experimental results show that the validity of the code generated by our approach is significantly higher than that of the baseline methods.

Keywords: Large Language Model (LLM), Knowledge Graph (KG), Retrieval-Augmented Generation (RAG), Competitive Programming, and Algorithm

1. Introduction

Algorithmic problem - solving plays a central role in the process of computer science education and competitive programming. That is to say, it requires one to carry out a precise reading of the specifications, make a correct selection of paradigms, and perform implementations that can meet the strict time and space limits. Even though there has been recent progress in the process of code synthesis and multi - turn reasoning, large language models (LLMs) still have problems such as misinterpreting constraints, selecting sub - optimal algorithms, and hallucinating or producing brittle code. These are the kinds of issues that impose limitations on both the automated solving and the pedagogical use [1-3].

We put forward a collaborative framework that combines knowledge graphs (KG) and large language models (LLM). This framework makes the generation process be grounded in structured and auditable knowledge. In the situation where test - time verification is carried out, it closes the loop. Specifically speaking, we take the action of constructing a Neo4j knowledge graph from OI - Wiki. This knowledge graph has the function of organizing each algorithm's principles, its complexity, the prerequisites that it needs, the pitfalls that may occur, as well as runnable examples. A hybrid

© 2026 The Authors. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (https://creativecommons.org/licenses/by/4.0/).

DOI: 10.54254/2755-2721/2025.LD28518

retriever, which includes Lucene full - text retrieval, 1024 - D vector similarity comparison, and keyword/alias matching methods, takes the step of shortlisting candidate algorithms. Then it expands the local neighborhoods so as to assemble targeted context. This context is woven into layered prompts. These prompts play the role of guiding the LLM to give justifications for the algorithm choice, to implement contest - grade C++ solutions, and to state the complexity. After that, a validator compiles and runs the code under circumstances similar to contest limits. If there are failures (such as wrong answer (WA), time limit exceeded (TLE), memory limit exceeded (MLE), runtime error (RE), or compilation error (CE)), they are fed back for the purpose of regeneration. This kind of approach is one that is free from the training process and it operates completely at the time when inference is taking place.

When our method is evaluated on 30 Luogu problems that cover the areas of graphs, dynamic programming, data structures, search, and geometry, it produces higher average scores, a greater number of perfect solutions, and significantly fewer zero - score failures compared to a vanilla large language model (LLM) that has the same decoding process. The gains are most clearly visible when the correctness of the solution depends on the appropriate paradigm and the careful handling of edge - cases. This situation aligns with the evidence that neuro - symbolic grounding and verifier loops can improve the reliability of the system.

Contributions.

A practical framework that combines a Knowledge Graph (KG) and a Large Language Model (LLM) for the process of dealing with algorithmic problems, which makes use of a typed Neo4j schema and a method of layered prompting.

Hybrid and graph - aware retrieval, which is a process that makes principles, examples, prerequisites as well as pitfalls that are most relevant to a query come to the surface.

Verifier-in-the-loop regeneration which carries out the process of enforcing contest - style constraints and also undertakes the process of stabilizing the quality of the code.

Empirical gains that are obtained on Luogu, which serve to highlight the places where the processes of graph grounding and the processes of validation matter the most.

The remaining part of this paper is arranged in the following way. In the process of Section 2, it conducts a review of the development status as well as the practical applications of large language models and knowledge graphs. In Section 3, it puts forward the whole pipeline of the work. In the process of Section 4, it presents the experimental setup and the results in a detailed manner. In Section 5, it offers a comprehensive conclusion of the study. It summarizes the key contributions and the findings, and at the same time, it also identifies the avenues for future enhancement and progress.

2. Related works

2.1. Large language models

Large Language Models (LLMs) have witnessed a rapid advancement process. This advancement occurs in the situation where they are advanced through the means of scaling and instruction tuning. Such advancement enables them to achieve strong code synthesis and multi - turn reasoning. However, these models still encounter issues. They still have the problem of hallucination. They also struggle when they are faced with strict constraints. And frequently, they fail when dealing with competitive - programming tasks. These tasks require the correct choice of algorithms and have tight time and space bounds.

In recent times, the remedies for these problems include retrieval - augmented generation, tool use, and verifier - or test - driven regeneration. This situation then motivates the action of coupling LLMs with structured external knowledge. The purpose of this coupling is to let the faithfulness and reliability of LLMs be improved [4-6].

DOI: 10.54254/2755-2721/2025.LD28518

2.2. LLM agents

Agentic large language models (LLMs) carry out an extension of prompting through the utilization of tools and the establishment of control loops. These control loops involve a sequential process of planning, then acting, then observing, and then reflecting. This process is grounded in patterns such as ReAct, which combines reasoning and acting, Reflexion, which involves self - critique, and Tree/Graph - of - Thought search.

Modern agents engage in the integration of function calling, the execution of code, the use of web tools, and the implementation of verifiers. Moreover, they are increasingly adopting hierarchical planning and multi - agent collaboration, which includes activities like debate and role specialization. In the process of conducting benchmarks in software engineering, for example, fixing bugs at the repository level, web navigation, and scientific workflows, it has been highlighted that the combination of closed - loop execution and verification is of crucial importance for ensuring reliability. Meanwhile, the open challenges in this area include the control of looping and termination, the management of cost and latency, and the guarantee of safety.

As a result, state - of - the - art systems combine external memory, skill libraries, and domain - specific orchestrators, which include retrievers, checkers, and simulators, in order to place constraints on actions and improve the success rates.

In the situation where these activities are taking place, agent designs that base their decisions on structured knowledge, such as graphs or databases, and carry out test - time validation have proven to be especially effective when dealing with precision - sensitive tasks like solving algorithm problems [7-9].

2.3. Knowledge graphs

Knowledge graphs (KGs) carry out the encoding of entities and relations so as to achieve precise querying and a transparent structure, and they are extensively employed in search and question - answering. In the process of modern graph storage, it provides support for hybrid retrieval. This hybrid retrieval is a process that combines full - text similarity and vector similarity over embedded content. Knowledge graphs are increasingly playing a role in complementing large language models (LLMs). They do this by performing the actions of grounding facts, disambiguating concepts, and constraining generation. We follow this neuro - symbolic direction. We do so by organizing algorithmic concepts, prerequisites, pitfalls, as well as examples in the form of a graph in order to guide the model's reasoning process [10-12].

2.4. Retrieval-Augmented Generation (RAG)

Recently, in the situation where research and development on RAG systems are being carried out, the recent RAG systems have shifted from the “BM25 + concatenate context” approach to end - to - end, learning - assisted pipelines. These pipelines optimize multiple stages in the following ways:

(i) In the process of query understanding, there are operations such as query rewriting, query decomposition, multi - hop planning, and the use of HyDE - style hypothetical docs.

(ii) For the hybrid retrieval, it fuses sparse retrieval methods (such as BM25/Lucene), dense retrieval methods (like bi - encoder) and cross - encoder re - ranking.

(iii) Instead of only dealing with raw text, source selection is carried out over structured stores, including tables, code, and knowledge graphs.

(iv) Regarding generation - time control, it includes citation - aware decoding, self - consistency with verifier filters, and retrieval refresh during decoding (which is also known as “retrieve - while - generate”).

DOI: 10.54254/2755-2721/2025.LD28518

Long - context models have the effect of reducing retrieval pressure. However, they are increasingly combined with cache/memory RAG and preference - optimized retrievers. The purpose of this combination is to curb the occurrence of hallucinations and improve the faithfulness of the system.

There is also a parallel line of study that focuses on multi - hop and graph - aware RAG. In this type of RAG, entity/edge expansions provide explanatory neighborhoods.

Collectively, these trends cause RAG to shift from ad - hoc concatenation to neuro - symbolic, evidence - grounded toolchains. As a result, there are measurable gains in factuality and task success [13-16].

3. Method

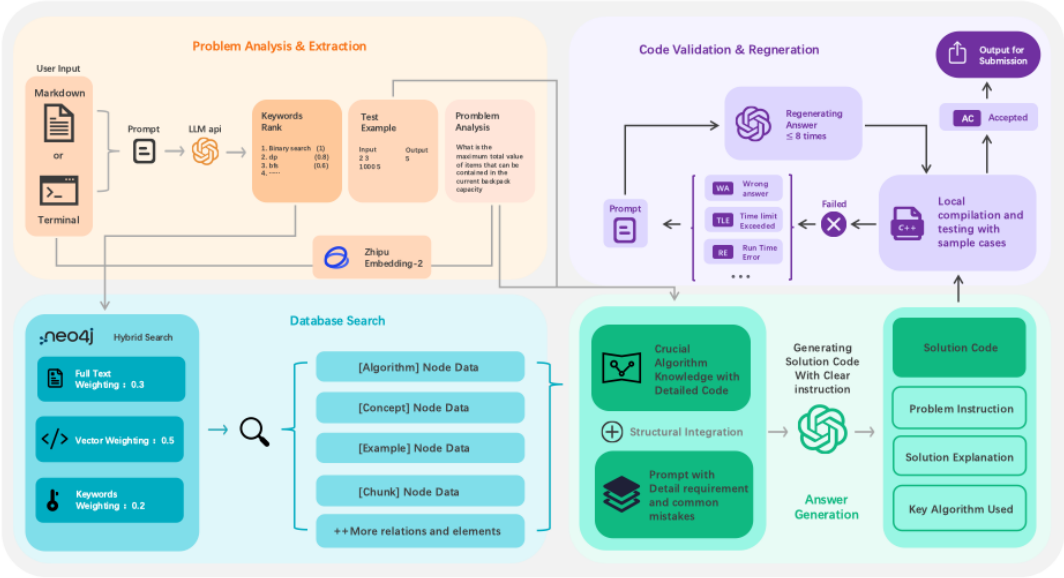


Figure 1. The overall workflow of the system is presented. The orange - colored part is used to demonstrate the process of Problem Analysis and the process of Problem Extraction. The blue - colored part is used to show the process of conducting a search in the database. The green - colored part is used to show the process of constructing a prompt and the process of generating an answer. Meanwhile, the purple - colored part is used to show the process of validating the code and the process of regenerating the code

The whole process of the entire pipeline is shown in Figure 1. In the situation where each module is considered, it corresponds to the methodological components that are described as follows.

3.1. Problem analysis & extraction

In this particular stage, the RAG system takes in inputs in two different forms. That is, it can either accept plain problem text that is entered through the terminal or Markdown files. This is because most Online Judge platforms usually provide full problem descriptions in the Markdown format. When the input has been received by the system, the system carries out the process of integrating the raw problem content together with explicit instructions for problem analysis. By doing so, it constructs the formalized prompt. Then, this formalized prompt is sent to the LLM API. After that, the LLM processes this prompt and gives back a structured response that conforms to the predefined standard format. Specifically speaking, the output is divided into three components. The first component is a

DOI: 10.54254/2755-2721/2025.LD28518

set of algorithm - related keywords. These keywords are assigned with different weights in order to highlight their relative importance. The second component is structured test cases. These test cases are used to ensure the systematic evaluation of solutions. The third component is a detailed analytical text. This text offers a comprehensive understanding of the problem.

3.2. Database search

We carried out the design of a hybrid search strategy in the process of operating within the Neo4j database. This hybrid search strategy is one that integrates full - text search, vector similarity search, as well as keyword matching so as to let the retrieval performance be improved. Specifically speaking, the full - text search component makes use of the Lucene indexing to conduct a search on algorithm descriptions and content. It does this by using the keyword strings that are extracted by the LLM (with a weight of 0.3).

The vector similarity search takes the core summaries generated by the LLM and embeds them into 1024 - dimensional vectors. It uses the Zhipu embedding - 2 model to achieve this embedding. Then, it proceeds to compute the cosine similarity against the algorithm embeddings that are stored in the database (with a weight of 0.5).

The keyword matching search further combines the keywords that are extracted directly from the problem statement with those that are produced by the LLM. After that, it matches them against the algorithm names, aliases, and keywords that are stored in the graph.

After the results are retrieved from these three methods, we take the scores and fuse them. By doing so, we can rank the algorithms. Then we select the top eight candidates for the process of graph expansion in order to collect detailed information. This detailed information is then prepared for the final prompt.

In addition to this, a fallback mechanism is put into implementation. That is to say, in the situation where the combined search fails or returns empty results, the system will revert to the keyword - only search. And if all the methods fail, instead of terminating abruptly, it will return a warning with empty content.

3.3. Prompt construction

In the process of this stage, we carry out the construction of the final prompt for the large language model (LLM). This construction is carried out on the basis of the knowledge that has been retrieved from the Neo4j database. The process of constructing this prompt is made up of three complementary layers. These layers are the retrieved context, the system prompt, and the user prompt. These three layers work together in such a way that they ensure the model is able to generate solutions that are accurate, standardized, and at a competition - level.

3.3.1. Retrieved context

We engage in the process of constructing a detailed contextual background. This background undertakes the task of integrating multiple aspects of algorithms. The purpose of this integration is to guide the reasoning process of the large language model (LLM).

The context encompasses the following aspects:

(i) Algorithm Overview: It involves carrying out a concise summary. This summary includes the name of the algorithm, its category, and its features. These details are obtained from the top three retrieved results.

(ii) Core Principle: It pertains to the theoretical foundation and the working mechanism of the algorithm.

DOI: 10.54254/2755-2721/2025.LD28518

(iii) Prerequisites: These are the necessary background concepts. Such concepts involve data structures or mathematical foundations.

(iv) Application Scenarios: These refer to the types of problems where the algorithm can be effective.

(v) Common Pitfalls: These are the typical mistakes and misconceptions that one needs to avoid.

(vi) Detailed Code Examples: These are complete and runnable implementations. The implementations are presented in one or more languages.

(vii) Related Technical Snippets: These are the auxiliary techniques or supporting tricks.

(viii) Implementation Tips: These are the practical advice for coding, debugging, and optimization.

3.3.2. System prompt

The system prompt is used to define the role of the large language model (LLM) as a coach for algorithmic competitions. It enforces a series of global instructions. These instructions include the following aspects:

Firstly, there is the role instruction. That is, the LLM is required to act as an expert in the fields of OI (Olympiad in Informatics) or ACM (Association for Computing Machinery).

Secondly, there are the core tasks. The LLM needs to understand the problem. It should ground the answers in the retrieved knowledge. It also has to leverage code examples. Moreover, under the situation where strict complexity constraints are imposed on it, it must ensure the completeness of the answers.

Thirdly, there is a required answer structure. This structure involves problem analysis, then algorithm selection, followed by implementation details. It also includes providing a complete C++ code, conducting complexity analysis, carrying out testing, and explaining the code reuse.

Fourthly, there are the code example principles. The LLM should learn from the retrieved implementations in terms of style, boundary handling, and optimizations.

Fifthly, there are conditional extensions. The LLM should adjust its focus dynamically for code, explanation, or comparison.

3.3.3. User prompt

The user prompt serves to offer the task - specific instruction layer. In the situation where the system prompt sets the global rules, the user prompt takes on the process of integrating the actual user query together with the retrieved knowledge and structured specifications. It encompasses the following aspects: (i) the process where the LLM analyzes the problem description; (ii) the structured input format and then the structured output format; (iii) multiple test examples which are used to enforce standardized outputs; (iv) the context information that has been retrieved; and (v) the explicit answer requirements that play the role of ensuring accuracy and clarity.

3.4. Code validation & regeneration

To make sure that the reliability of the solutions that are generated can be guaranteed, we carried out a design for a validation module. This validation module imitates the workflow of human contestants. Human contestants usually take the action of submitting their code only after they have managed to pass local test cases. Specifically speaking, after the LLM has produced a solution, the system takes the action of extracting both the code and the sample cases that are embedded in the response. Then, the extracted code is subjected to the processes of being compiled and being executed against the test cases under the situation where there are strict runtime constraints. These constraints include a time limit of 5 seconds (this time limit is slightly relaxed in comparison to Online Judges because the local execution is slower) and a memory limit of 256MB. In the event that the execution violates these

DOI: 10.54254/2755-2721/2025.LD28518

limits or generates incorrect outputs, the error type, the code that has failed, and its context are sent back to the LLM so that the LLM can carry out the process of regeneration. This iterative process is repeated up to eight times, and this repetition significantly contributes to the improvement of the final accuracy of the solutions.

The validation module provides support for a range of error types. There is the Wrong Answer (WA) situation when the output does not match the expected result. There is the Time Limit Exceeded (TLE) situation when the runtime goes beyond the limit. There is the Memory Limit Exceeded (MLE) situation when the memory usage exceeds the limit. There is the Runtime Error (RE) situation when issues such as array out - of - bounds, null pointer dereferencing, or division by zero take place. And there is the Compilation Error (CE) situation when syntax mistakes, missing headers, or undeclared variables are detected. This automatic validation - regeneration loop functions as a safeguard. It ensures that only the code that is executable and correct can pass through the pipeline.

3.5. Database

We engage in the process of designing a structured knowledge base by making use of the Neo4j graph database so as to provide support for algorithmic retrieval within the RAG pipeline. This database is used to store structured algorithmic knowledge in the form of interconnected nodes and relationships. In this context, algorithms, concepts, examples, as well as code snippets are represented in an explicit manner. This design that is based on the graph enables not only flexible traversal but also explicit reasoning regarding prerequisites, application scenarios, and common pitfalls.

To bring about an enhancement in retrieval accuracy, the database makes the adoption of a hybrid retrieval paradigm, which involves the combination of symbolic and semantic methods. When it comes to the symbolic aspect, the full - text indexing that is based on Lucene offers support for the retrieval that is based on keywords over algorithm names, aliases, as well as descriptions. In the case of the semantic side, textual chunks and code fragments are taken and embedded into 1024 - dimensional vectors through the utilization of the zhipu embedding - 2 model. These embeddings are then put through the process of being indexed via Neo4j’s vector search engine, which offers support for the retrieval that is based on cosine - similarity. This kind of integration of keyword search and semantic vector similarity makes it possible to provide both precision and recall when it comes to the process of matching queries to relevant algorithms.

As the data source, we make a selection of OI Wiki. This is a source that offers a comprehensive and structured way of covering mainstream algorithms. Its content, which has been well - curated, makes sure that the database contains not only definitions and theoretical explanations but also detailed code implementations, practical applications, and potential pitfalls. In the situation where this knowledge is embedded and structured in Neo4j, the RAG system can gain advantages from a rich, multi - relational knowledge graph. This knowledge graph is capable of powering hybrid search and the construction of downstream prompts.

4. Experiment

4.1. Setup

4.1.1. Details

We make use of Zhipu GLM - 4.5 as our LLM api in the situation where the framework has a configuration in which the sampling temperature is set to 0.6 and the nucleus sampling parameter (top - p) is set to 0.95. And we use Zhipu embedding - 2 for all of the work related to embedding. Since the model often generates overly long and looping reasoning traces when dealing with harder

DOI: 10.54254/2755-2721/2025.LD28518

tasks, we take the action of turning off the built - in reasoning capability both in our API setting and in the baseline so as to conduct a fair comparison.

4.1.2. Datasets

We made a selection of 30 challenging problems from Luogu Online Judge. These problems cover a situation where they span across a wide range of algorithms. The full text of the problems was taken and copied to the input.md. This input.md is the file that will be read by the system.

4.1.3. Baseline

We take the step of adopting the vanilla large language model (Zhipu GLM - 4.5) to serve as the baseline. This is combined with a simple prompt for the purpose of solving problems. In the situation where the sampling temperature is set to 0.6 and the nucleus sampling parameter (top - p) is set to 0.95, the process of providing the complete problem description in Markdown format to the model is carried out through a dialog interface. Then, the C++ code that is generated by the model is taken and compiled, and after that, it is executed to conduct a verification of its correctness in comparison with the given test cases.

4.1.4. Knowledge graph

We take the action of organizing algorithmic knowledge in the form of a typed property graph within Neo4j. This is done in order to offer both an explicit structure and semantic retrievability for the work pipeline. The graph creates models for four node classes. These node classes are Algorithm (which includes fields such as title, category, intro, principle, complexity fields, keywords, and aliases), Chunk (which are retrieval - oriented text segments that come with 1024 - dim embeddings), Example (which are code snippets having language/solution metadata), and Concept (which consists of prerequisites, applications, and pitfalls). These node classes are connected through semantically meaningful relations. These relations are HAS_CHUNK, HAS_EXAMPLE, PREREQUISITE, APPLICATION, and PITFALL. This connection enables traversals that can bring to the surface background knowledge, typical use cases, and failure modes around each algorithm.

To provide support for hybrid search, we carry out the combination of Lucene full - text indexing (which is applied on Algorithm titles, intros, and keywords) with vector similarity search (which is based on the cosine over Chunk embeddings that are produced by zhipu embedding - 2). Also, we preserve symbolic matching by means of names and aliases. The content is obtained from OI - Wiki and is normalized into consistent properties and unique identifiers. Constraints and indexes are defined with the purpose of reproducibility.

This design allows the retriever to first create a shortlist of relevant algorithms. Then, it can expand locally in the graph to assemble rich, task - conditioned context (including principles, examples, and concepts) for prompt construction. This serves to bridge structured reasoning with semantic evidence at the time when inference takes place.

The following Table 1 and Table 2 show the categories of the nodes as well as the relationship between them.

Table 1. Nodes categories

Node label	Key properties	Semantics	Constraint	Indexing
Algorithm	uid, title, category, intro, principle, cpx_time, cpx_space, keywords, aliases	Canonical record of an algorithm, including metadata, textual description, and stated time/space complexity.	UNIQUE (uid)	Full-text index on title, intro, keywords (Lucene).

DOI: 10.54254/2755-2721/2025.LD28518

Table 1. (continued)

Chunk	id, content, embedding	Short textual segment associated with an algorithm; embedding stores a dense vector for semantic retrieval.	UNIQUE (id)	Vector index on embedding (1024-D, cosine similarity).
Example	id, title, description, code, solution, lang, has_code	Worked examples and code artifacts illustrating how to implement/use the algorithm.	UNIQUE (id)	N/A
Concept	name	Concept node capturing prerequisites, application scenarios, and common pitfalls.	UNIQUE (name)	N/A

Table 2. Relation categories

Relationship Type	Source Node	Target Node	Semantics (Definition)	Usage Scenario
HAS_CHUNK	Algorithm	Chunk	Indicates that an algorithm contains associated text chunks	Vector similarity search; content retrieval
HAS_EXAMPLE	Algorithm	Example	Indicates that an algorithm is associated with illustrative examples	Code example lookup; implementation reference
PREREQUISITE	Algorithm	Concept	Represents prerequisite concepts required to understand the algorithm	Learning path planning; knowledge dependency modeling
APPLICATION	Algorithm	Concept	Represents application scenarios where the algorithm can be used	Problem categorization; algorithm recommendation
PITFALL	Algorithm	Concept	Represents common pitfalls or mistakes when applying the algorithm	Error prevention; debugging assistance

A table that offers more detailed information about the node is provided in Appendix A. Specifically, this table includes Table A.1, Table A.2, Table A.3, as well as Table A.4.

4.1.5. Evaluation metrics

We adopt the approach of using the score of the problem on Luogu Online Judge as the primary evaluation. In the process of determining the score for each individual problem, it is decided by the number of passed test cases. And this score is directly calculated by the Luogu Online Judge system through a specific computational process.

4.2. Results & analysis

Table 3 below presents the outcomes of the experiment where scores are obtained from Our work along with those from Vanilla LLM. In this experiment, the situation is that it involves the assessment of scores coming from Our work and also from Vanilla LLM, and the results of this whole experimental process are shown in the following Table 3.

DOI: 10.54254/2755-2721/2025.LD28518

4.2.1. Results

Table 3. Quantitative results

Method	Average Score	Perfect Scores	Problems with ≥ 50-pt Gain	Complete Failures
Our work	73.67	17	8	1
Vanilla LLM	46.77	11	-	11

- Average score: RAG 73.67 vs. Vanilla LLM 46.77
- Perfect (100-point) problems: RAG 17 vs. Vanilla LLM 11
- Problems in the situation where Retrieval - Augmented Generation (RAG) makes an improvement of 50 points or more compared to Vanilla Large Language Model (LLM): 8
- Complete failures (0 points): RAG 1 vs. Vanilla LLM 11

A complete table regarding the process of problem selection and the process of filtering criteria is provided within the content of Appendix B (that is to say, Table B.1).

4.2.2. Analysis

Across a total of 30 algorithmic problems, the work that we have carried out substantially outperforms a vanilla Large Language Model (LLM). Specifically, when we look at the average score, it is 72.0 for our work as opposed to 43.43 for the vanilla LLM. In terms of the number of problems with a perfect score, our work has 18 such problems while the vanilla LLM has 11. There are large improvements (that is to say, improvements of ≥50 points) on 12 problems. And regarding the number of complete failures (0 points), our work has 1 and the vanilla LLM has 12 respectively.

The gains are most pronounced on topics that have strict complexity and engineering requirements. For example, in the area of graph algorithms such as the Hopcroft then Karp algorithm and the Bellman then Ford algorithm, in range data structures like segment trees with lazy tags, in binary search on answers, and in tree divide - and - conquer. In these cases, the retrieved and structured knowledge as well as code exemplars help to reduce implementation errors and ensure compliance with the complexity requirements.

Performance parity appears on foundational tasks that have clear paradigms. For instance, in tasks like the Dijkstra algorithm, the DAG longest path problem, and the Breadth - First Search (BFS). In these foundational tasks, both approaches often achieve full marks.

We make an observation of a small number of counter - cases (for example, convex hull or geometry, Longest Increasing Subsequence (LIS), block decomposition). In these cases, the situation is that the vanilla model sometimes matches or even goes beyond our work. These categories are either of a nature where they are highly template - heavy (such as LIS and block decomposition) or they are numerically fragile (like in the field of computational geometry). This characteristic makes the results sensitive to the coverage of exemplars (for instance, collinearity, precision, and the handling of degeneracy). It is worth noting that our work shows significantly better stability. There are far fewer outcomes with a zero - score. This suggests that the approach of hybrid retrieval (which includes full - text retrieval, vector retrieval, and keyword retrieval) along with example - aligned prompting helps to mitigate the problem of wrong - algorithm selection and brittle implementations.

5. Discussion

The task of solving algorithm problems is a fundamental one within the process of computer science education and competitive programming. Nevertheless, in the situation where there is a serious illusion problem, the existing large language models (LLMs) frequently face difficulties when it comes to dealing with complex problems and generating correct as well as efficient solutions under

DOI: 10.54254/2755-2721/2025.LD28518

the condition that strict time and space constraints are imposed. There is still a substantial amount of room that can be improved. We put forward a collaborative framework that combines a knowledge graph and a large language model, with the aim of solving the algorithm problems for computer science education and new algorithm learners. It automatically goes through the process of solving the algorithm problems and outputs the solution code along with detailed explanations. On the basis of the Neo4j platform, we constructed a comprehensive knowledge graph, extracting typical and practical algorithm knowledge from OI WIKI. The results show that our work has great performance in the process of algorithm problem - solving when compared to the vanilla LLM.

6. Conclusions

Our work demonstrates a high level of effectiveness when it comes to the process of dealing with algorithm - related problems. However, there are still a large number of problems that remain to be handled. In the process of dealing with many difficult problems that involve complex algorithms, the score of the solution can still reach 100. The framework shows strong performance in the situation where it is under the test - point–based OI scoring. But in ICPC settings that have binary verdicts (AC/non - AC), it often fails to achieve consistent full acceptance. We are looking forward to the process of improving and optimizing this system in the future so as to assist more people in the process of learning and mastering algorithms.

References

[1] Coignion, T., Quinton, C., & Rouvoy, R. (2024, June). A performance study of llm-generated code on leetcode. In Proceedings of the 28th international conference on evaluation and assessment in software engineering (pp. 79-89).
[2] Wei, M., Li, Z., Chen, X., Zheng, M., Qu, Z., Yu, C., ... & Ju, X. (2025). Evaluating and Improving Large Language Models for Competitive Program Generation. arXiv preprint arXiv:2506.22954.
[3] Huang, L., Yu, W., Ma, W., Zhong, W., Feng, Z., Wang, H., ... & Liu, T. (2025). A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. ACM Transactions on Information Systems, 43(2), 1-55.
[4] OpenAI, R. (2023). Gpt-4 technical report. arxiv 2303.08774. View in Article, 2(5), 1.
[5] Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., & Narasimhan, K. (2023). Swe-bench: Can language models resolve real-world github issues?. arXiv preprint arXiv:2310.06770.
[6] Ji, Z., Lee, N., Frieske, R., Yu, T., Su, D., Xu, Y., ... & Fung, P. (2023). Survey of hallucination in natural language generation. ACM computing surveys, 55(12), 1-38.
[7] Liu, X., Yu, H., Zhang, H., Xu, Y., Lei, X., Lai, H., ... & Tang, J. (2023). Agentbench: Evaluating llms as agents. arXiv preprint arXiv:2308.03688.
[8] Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., & Yao, S. (2023). Reflexion: Language agents with verbal reinforcement learning. Advances in Neural Information Processing Systems, 36, 8634-8652.
[9] Wu, Q., Bansal, G., Zhang, J., Wu, Y., Li, B., Zhu, E., ... & Wang, C. (2024, August). Autogen: Enabling next-gen LLM applications via multi-agent conversations. In First Conference on Language Modeling.
[10] Peng, C., Xia, F., Naseriparsa, M., & Osborne, F. (2023). Knowledge graphs: Opportunities and challenges. Artificial intelligence review, 56(11), 13071-13102.
[11] Pote, M. (2024). Survey on embedding models for knowledge graph and its applications. arXiv preprint arXiv:2404.09167.
[12] Pan, S., Luo, L., Wang, Y., Chen, C., Wang, J., & Wu, X. (2024). Unifying large language models and knowledge graphs: A roadmap. IEEE Transactions on Knowledge and Data Engineering, 36(7), 3580-3599.
[13] Asai, A., Wu, Z., Wang, Y., Sil, A., & Hajishirzi, H. (2024). Self-rag: Learning to retrieve, generate, and critique through self-reflection.
[14] Gao, L., Dai, Z., Pasupat, P., Chen, A., Chaganty, A. T., Fan, Y., ... & Guu, K. (2022). Rarr: Researching and revising what language models say, using language models. arXiv preprint arXiv:2210.08726.
[15] Sarthi, P., Abdullah, S., Tuli, A., Khanna, S., Goldie, A., & Manning, C. D. (2024, May). Raptor: Recursive abstractive processing for tree-organized retrieval. In The Twelfth International Conference on Learning Representations.

DOI: 10.54254/2755-2721/2025.LD28518

[16] Wang, J., He, Y., & Chen, H. (2024). Repogenreflex: Enhancing repository-level code completion with verbal reinforcement and retrieval-augmented generation. arXiv preprint arXiv:2409.13122.

Appendix

The table of the full content can be found in the situation where one accesses the website https://github.com/Wyk - formal/Appendix1.git.