Visualizing Program Behavior: A Study of Enhanced Program Diagrams Using LLM

Ying Li[1,3] Runze Yang[1] ShiJie Gui Peng Shi[2] XuefeiHuang[2] Da Yang[2] Xiaozhou Zhang[2] Yiming Gai[2]
liying@buaa.edu.cn huangxuefei@buaa.edu.cn da.yang@buaa.edu.cn
[1]School of Computer Science and Engineering, Beihang University, Beijing, China
[2]Key Laboratory of Data Science and Intelligent Computing, International Innovation Institute,Beihang University, Hangzhou, China
[3]Corresponding author

***Abstract*—This paper aims to address the difficulties faced by novice programmers in grasping code structure and execution flow, improving programming thinking, and pinpointing code errors with accuracy. It proposes providing students with program behavior diagrams based on large language models (LLMs) and visualization techniques to achieve personalized guidance. Specifically, these program behavior diagrams include programming thinking visualization diagrams and code vulnerability visualization diagrams. A programming thinking visualization diagram employs static code analysis to gather code structure information, combined with the structured chain-of-thought method to collectively optimize the LLM. This enables the LLM to explain each interpretable part of the code from top to bottom, detailing the programming concepts, and displaying them on a modularized code structure diagram. The code vulnerability visualization diagram primarily utilizes the fine-tuned LLM, optimizing it based on program analysis and clustering analysis methods to accurately identify vulnerabilities in student code and display them on a code flow diagram. Its feature is to visually display to students the error location, error information, and the impact of errors on program flow, rather than providing the programming answers. Lastly, through experiments and statistical analysis of actual teaching data, this paper serves a demonstration that the enhanced models used in the visualization diagram generation process have a noticeable effect on mainstream LLMs, and that visualization diagrams hold significant value for students at different stages of learning.**

Keywords—Large Language Models, Code Analysis, Code Flowchart, Code Vulnerabilities, Structured Chain-of-thought, Computer Education

I. INTRODUCTION

In this current era of the digital age, smart education and artificial intelligence technologies are integrating into the field of education at an unprecedented pace. The use of artificial intelligence technology to achieve personalized learning has also appeared many times in academic reports [1],[2],[3]. In recent years, as LLM has demonstrated its extraordinary generative capabilities, some domestic and foreign educators have applied LLM to teaching practice [4],[5]. For example, CodeAid at the University of Toronto has applied LLM to teaching practice [6]. On the other hand, in the practice of programming teaching, especially in the teaching of programming for beginners, the process from zero foundation to entry poses a challenge to students' logical thinking. If we can directly provide students with visual help and guidance on programming thinking through visual means during the learning process, we can effectively help students get started.

In recent years, various applications based on large language models (LLMs) have achieved certain results in programming learning [7]. However, due to the complexity of the learning cognitive process and the actual teaching environment, there are still some problems, which are mainly reflected in the following two aspects.

1) In terms of code visualization

Currently, most of the related courses on widely used mainstream platforms, such as Coursera and edX, lack auxiliary functions for visual code analysis. On the other hand, current code-related visualizations often only reflect the control flow and data flow of the code, and do not provide students with guidance on programming thinking [8],[9],[10].

2) In terms of code-related applications of LLMs

Current research focuses on code generation. Direct use will destroy students' independent thinking and is not suitable for teaching scenarios. In addition, despite the high accuracy of LLMs, the occasional "hallucination" phenomenon can also have a serious negative impact on beginners [11],[12].

Due to the above two problems, this paper is mainly based on the LLM, comprehensively uses code analysis and other technologies, conducts a series of explorations in code visualization, and designs and optimizes programming thinking visualization diagrams and code vulnerability visualization diagrams. Then, experiments are designed to verify the effectiveness of the optimization scheme in the article. Finally, it was applied to teaching practice and achieved certain results.

II. PHYSICAL IMPLEMENTATION

This paper primarily consists of two modules: first, it provides a detailed introduction to the generation process of programming thinking visualization diagrams. Its distinctive feature is the use of chain-of-thoughts to generate programming guidance for each modularized module, assisting students in quickly getting started. Next, it explains the process of the generation of code vulnerability visualization diagrams. Its unique feature is the generation of a specialized LLM for students in this discipline, using code analysis and fine-tuning of LLMs to achieve high accuracy. Additionally, this paper will also present the detailed implementation process of such tools, providing an application paradigm using LLMs tailored for educational purposes.

A. Generation of Programming Thinking Visualization Diagrams

The main approach in this section can be divided into four steps: modularization of code, enhancing the LLM's understanding of code overall, enhancing the LLM's understanding of relationships between code parts and the overall structure, and the generation of visualization diagram.

1) Modularization of Code

On the one hand, in order to generate the visualizations of code, it is necessary to modularize and break down the code to clearly display its hierarchical relationships. On the other hand, due to the existence of structural features in the code, it is necessary to incorporate structured features into prompts, design thinking chains to guide LLMs, and to make a reasonable modular division of the code itself. Based on these objectives, a rational way of division needs to meet the following two requirements: the process of code division should not lose structural information, and the smallest units obtained after code division should have logical independence for effective interpretation by LLMs. This paper will divide modularization into three distinct levels, each with their own specific definitions.

File level: Each file serves as a node. This is because in actual teaching aimed at beginners, automatically assessed assignments typically consist of only a single file, making it convenient to serve as a root node.

Function level: A single file that consists of multiple functions (including the main function). This is because functions often exhibit strong independence from each other, and variables and statements within two functions are only related through explicit function calls.

Structure level: Three consecutive statement segments distinguished by execution process, including sequential structure (which includes initialization statements), selection structure, and

looping structure. This is because using explicit keywords can effectively segment the code, and the segmented code segments generally exhibit distinct characteristics. Additionally, this segmentation method has the additional benefits of covering the entire code content.

In terms of the specific implementation, this section of the paper utilizes an open-source tool, tree-sitter, to assist in generating syntax trees, combined with relevant knowledge of compiler theory to modularize the code. This is because abstract syntax trees naturally contain structural information about the code, and by analyzing the structure of abstract syntax trees, the code can be modularized conveniently and comprehensively without any omissions.

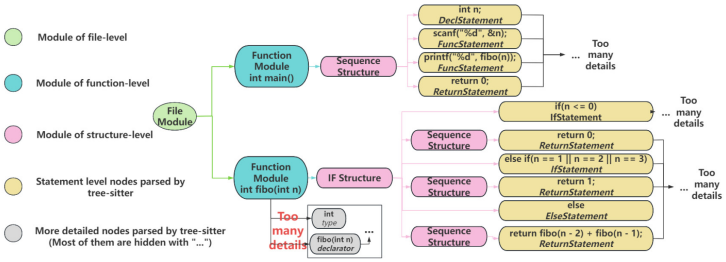

Figure 1 Preliminary modularization of a student code

Here, tree-sitter acts as an assistant in generating abstract syntax trees (AST) for C code, whose structure is roughly depicted in Figure 1. By analyzing the structural characteristics of the abstract syntax tree, it is revealed that nodes at the same depth correspond top-down to code blocks in the source code, while nodes at different depths correspond to syntax nesting in the source code. Leveraging the hierarchical information provided by the syntax tree, preliminary modularization of syntax structures can be achieved by traversing the syntax tree in a preorder manner, resulting in a control flow graph containing code structure information.

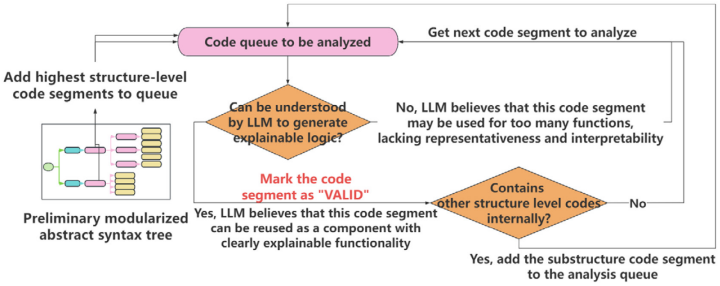

Figure 2 Filter structural-level modules with independent logic

It is worth noting that since the syntax tree does not inherently understand the meaning of the code, for more complex code, the internal structures within selection and loop constructs often contain smaller structural code segments. During the initial modularization process, we have to recursively split these segments; however, due to their short length after the split, they may lack logical independence. Therefore, for each structural entity, we conduct exploratory inquiries to ensure that the LLM comprehends the behavior corresponding to the structural entity's code segment. Invalid code segments are flagged, ensuring that in subsequent chain-of-thought guidance work, no inquiries are made about the invalid segments, as depicted in Figure 2.

Enhancing the LLM's Understanding of Whole Code

The main purpose of this section is to enable the LLM to have a more holistic perception of the code, avoiding scenarios where subsequent modularization and other operations lead the LLM to focus excessively on code particulars and overlook the overall architecture. This section will be divided into two parts. Initially, we designed prompts specifically to help the LLM understand the code holistically.

However, through experimentation, we found that the LLM's understanding of code control structures was inadequate. Therefore, we had to optimize the prompts by incorporating code structure information, enhancing the LLM's comprehension of the overall code structure.

Given the complexity of any educational settings, it is crucial to identify which useful information, aside from the student's code itself, is available to design foundational prompts. In our basic prompt, we focused on three parts: teaching topic description, reference answers and test points.

When testing different lengths of code using basic prompts, we found that for lengthier code, the LLM's performance to grasp of the overall code structure was not satisfactory. The cause for this issue is that the LLM interprets code as natural language. Therefore, the problem was mitigated by incorporating a code control flow into the prompts.

Specifically, the open-source tool tree-sitter was used to assist in generating abstract syntax trees (AST) for C code. Then, based on the AST, a code control flow was derived. Subsequently, this control flow information is then described in natural language within a designed framework. This approach essentially represents the meaningful part of code in natural language, helping the LLM to understand the overall code structure from a top-down perspective. Figure 3 illustrates an example process of representing code in natural language.

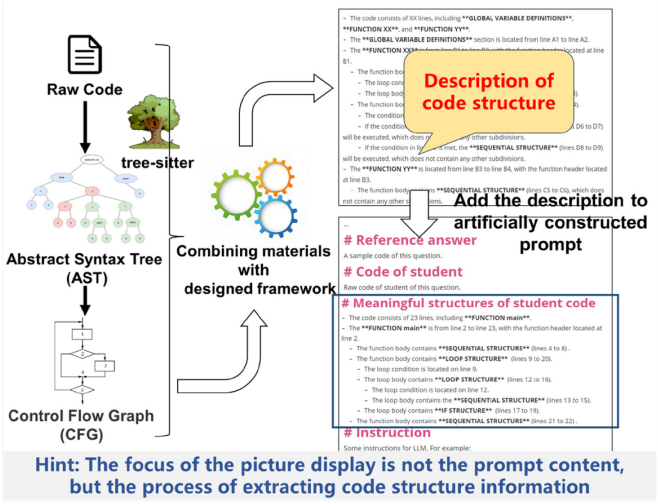

Figure 3 Represent the meaningful part of code in natural language

2) Enhancing the LLM's Understanding of Programming Thinking

This section introduces how to use CoT in this article. Its feature is to use a structured CoT template to guide LLM to deepen its understanding of the local and overall structure of the code, thereby improving the model's ability to understand the overall programming thinking. The CoT template is constructed by experts, and its intention is divided into two parts: part and whole. First, guide LLM to analyze the behavior of each part of the code structure, such as "Think about the function of the X part of the code segment" or "Think about the order of the X part of the code in the programming process thinking", and then guide LLM to think about the connection between the part and the whole, such as "think about the role of the X part of the code segment in the Y part of the code segment (Y contains X)".

Our CoT achieved good results for the following three reasons:

First, when implementing X and Y of the CoT, they will be given by modular code snippets. As mentioned in Part A 1) above, we removed a large number of meaningless small code snippets and

retained the code snippets that the model believes have independent logic.

Second, in the construction of prompts, we used natural language to describe the code structure to help LLM extract structural information from long codes, as mentioned in Part A 2) above.

Third, we also used examples constructed by experts for few-shot, which provided an effective reference for LLM's thinking ability.

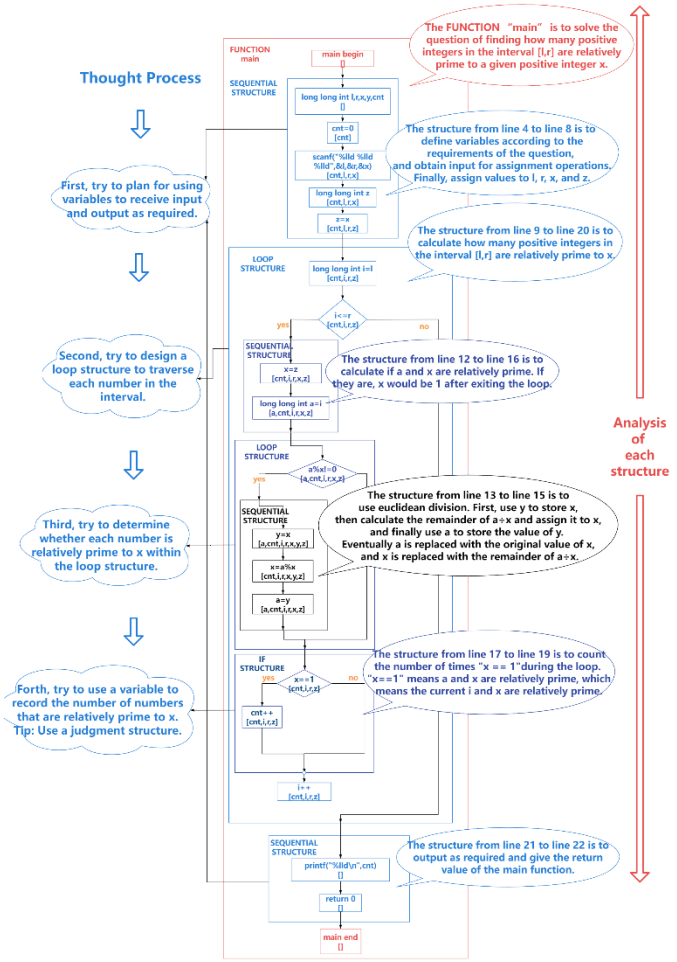

Figure 4 A programming thinking visualization diagram

3) Programming Thinking Visualization Diagram

The programming thinking visualization diagram, depicted in Figure 4, consists primarily of control flow, data flow, and programming thinking information, modularized and modeled based on syntax trees and the LLM. This visualization diagram exhibits two distinct advantages: unlike existing code flowcharts, the programming thinking visualization diagram features clear modularization characteristics. It models textual code into structured modules with explicit meanings, enabling beginners to swiftly grasp the code's structural information and enhance their understanding of the overall code structure. Additionally, distinct from code representation tools, the programming thinking visualization diagram incorporates analyses and guidance on coding thought processes. Beginners can gain clearer insights into programming thinking using this enhanced flowchart perspective.

B. Localization and Visualization of Code Vulnerabilities Using LLM

The process of localizing code to vulnerabilities can be broken down into three key steps. First, by integrating dynamic and static program analysis techniques, various programming errors and code defects are accurately identified from student-submitted code. These are then incorporated into prompt statements for the LLM, aiding in generating more precise responses. Second, using error-prone code from previous student submissions, targeted adjustments are made to the training data of LLM. This includes increasing the proportion of erroneous code in the training data, fine-tuning the model's abilities in error classification and localization, and designing prompt statements that align closely with real-world educational scenarios. Finally, in the visualization of code vulnerabilities, not only are error locations annotated and explained, but the impact of these errors on data flow and control flow is also showcased, helping students understand the reasons behind error occurrence.

Through the integration of these three key steps, the localization and visualization of code vulnerabilities have been designed and implemented. The following sections will detail the specific methods and practical outcomes of each step.

1) Enhancing the LLM Based on Error Information

This section will elaborate on the detailed process of enhancing the LLM based on error information. The core focus is on automatically generating effective prompts for the LLM. To address the issue where the LLM often overly emphasizes code style and overlooks critical code vulnerabilities, this paper emphasizes in the prompt the specific code segments that should be prioritized by the LLM. Additionally, due to the likelihood of coding compilation errors in a typical beginner learner's programming code, directly converting code with the compilation errors into a syntax tree can lead to a series of errors. Therefore, this paper mitigates the problem by applying different processing methods to codes that do not pass compilation and codes that do, as depicted in Figure 5.

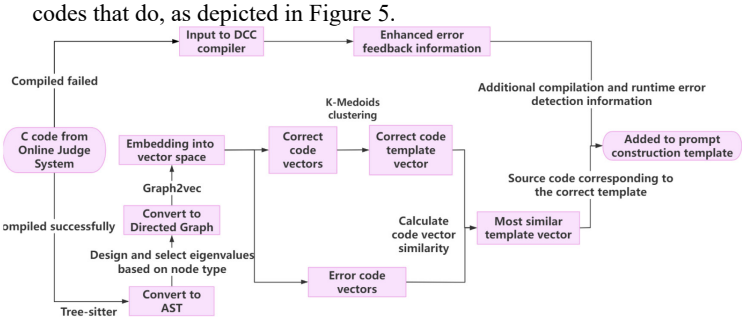

Figure 5 Flowchart for enhancing LLM based on error information

For codes that do not pass compilation, DCC (Debugging C Compiler) is used to compile and run the code submitted by students to obtain more detailed feedback results. Specifically, DCC embeds runtime error detection tools such as Valgrind, AddressSanitizer and GDB into the generated executable file to provide additional information to the DCC error interpretation system. This includes printouts of call stacks and results of memory leak detection. The integration of these tools enables comprehensive monitoring and analysis of potential errors during program execution, aiding in more effective diagnosis and problem resolution.

For codes that pass compilation, a template-based code vulnerability analysis method was deployed which focuses on identifying the most similar correct code to the current code. Firstly, this paper used a tree-sitter parser to convert source code into a structured syntax tree. Then, the syntax tree is mapped to a graph structure representing nodes and edges. Specifically, node types are mapped to integers as node feature values in the graph representation, ensuring that nodes of the same type are assigned the same feature value regardless of the size of the syntax tree. Subsequently, Graph2Vec is used as the core method for graph embedding to convert nodes and edges in the graph into low-dimensional vectors while preserving the topological structure and semantic relationships between nodes. Finally, cosine similarity is used as a measure of code

similarity, and the K-Medoids algorithm is applied to perform clustering analysis on vectors derived from all correct codes for the same programming problem, yielding clusters of correct code where cluster centers represent correct template vectors. By calculating the cosine similarity between each error code vector and all correct template vectors, we can find one or more correct template vectors most similar to each of the error code vectors, thereby identifying corresponding source codes. This information is also used to guide subsequent analysis and correction work.

Finally, we incorporated the information obtained from the different scenarios into prompts. For codes that do not pass compilation, we emphasized syntax errors, whereas for codes that do pass compilation, we highlighted the differences from accessible template codes, especially logical errors. Additionally, evaluation results from the assessment engine were included in the prompts to provide references.

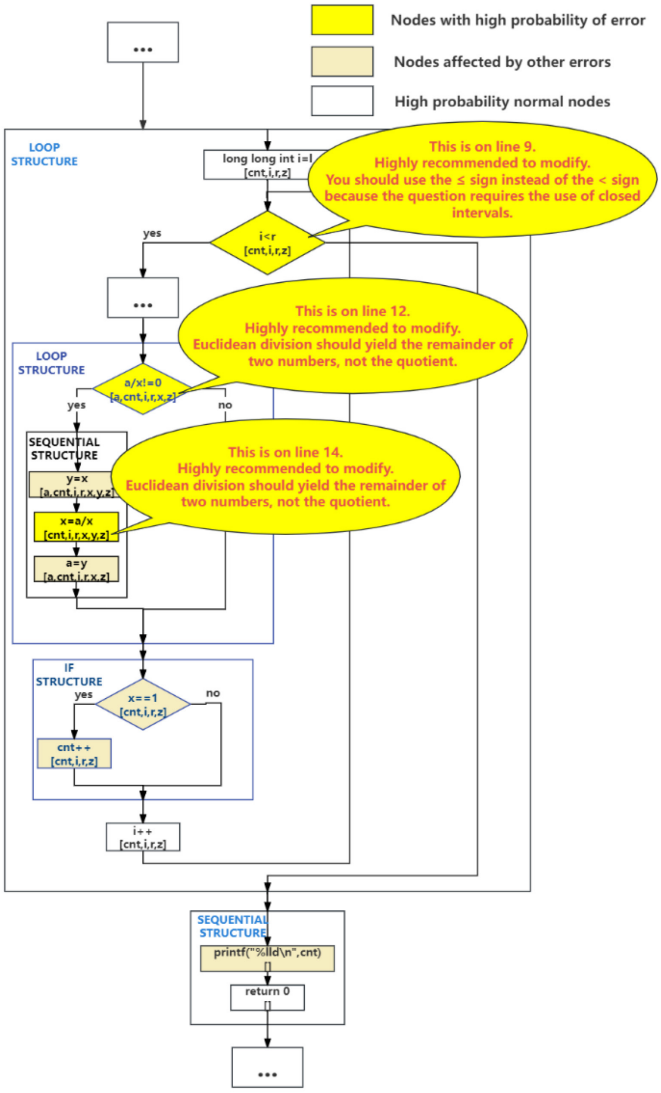

Figure 6 A code vulnerability visualization diagram

2) Fine-Tuning the LLM Based on Existing Student Code

Existing LLMs are generic models that sometimes provide suggestions beyond the understanding of beginners. This is because their training data contains a significant amount of more advanced programming content, leading to suboptimal feature extraction for beginner-level code. To better adapt the LLM to the characteristics of student-submitted erroneous code, this paper conducts fine-tuning based on existing student code.

We chose the CodeLlama LLM that suits the task requirements and dataset scale, then we expanded its vocabulary and pre-trained it on Chinese corpora. Additionally, we manually annotated a dataset of erroneous past student code (approximately 5000 pairs of questions and answers) to fine-tune the LLM. In addition, we also used GPT4 to expand our dataset to about 30,000 question-answer pairs. Due to resource constraints, we did not perform SFT fine-tuning, but instead performed LoRA fine-tuning. We conducted evaluations after each fine-tuning and manually checked the training data corresponding to the part where the model performed poorly, adjusted this part of the data, and then fine-tuned again or continued fine-tuning. The whole process took about 4 months. Through multiple rounds of fine-tuning and performance evaluation, we obtained a fine-tuned LLM for beginner-level code. Compared to directly using ChatGPT4, this fine-tuned model showed a significant improvement in error localization accuracy on our constructed beginner-level code dataset, which better serves subsequent error analysis and flowchart display tasks.

3) Code Vulnerability Visualization Diagram

The error information provided by the LLM includes error positions (line numbers) and error messages. The code vulnerability visualization generated in this paper hides programming thinking information due to different usage scenarios, retaining its modular structure, control flow information and data flow information to help students quickly identify code errors and understand the impact of code errors on control flow. The located control flow nodes or modules are highlighted in the control flow diagram, and error information is displayed in a tooltip format, as shown in Figure 6.

III. EXPERIMENTS

A. Model Capability Evaluation

Experiment 1 evaluated the model's ability to provide effective code repair suggestions and programming mindset advice, while also verifying the effectiveness of extracting code structure information and using the chain-of-thought approach. The dataset for this experiment was collected from 2400 answer codes obtained from a self-built Online Judge system, covering 120 questions across 6 different topics with increasing difficulty. After preprocessing the code, commonly used LLMs from domestic and international sources were employed to provide programming mindset suggestions, which were then evaluated by human assessors (scored on a scale of 0-10).

The experimental results are presented in Table 1. The Code structure information column indicates whether to add code structure information to the prompt, and the CoT column indicates whether to add the CoT guidance generated by the code structure information and the CoT template to the prompt. The programming thinking effectiveness score indicates the manual evaluation of the effectiveness of the answer provided by LLM on the programming thinking suggestions.

Table 1 Model capability assessment test.

Model	Code structure information	CoT	Program Thinking Effect
Qwen1.5 (72B)	×	×	3.24
	√	×	5.61
	√	√	8.58
CodeLLaMA (34B)	×	×	3.75
	√	×	4.36
	√	√	8.43
GPT-4 (Turbo)	×	×	6.58
	√	×	7.26
	√	√	9.23

It can be observed that the inclusion of code structure information and chain-of-thought has a certain effect on error localization and programming mindset suggestions, with the addition of chain-of-

thought showing a particularly significant improvement in programming mindset advice.

B. Real-World Scenario Evaluation

To comprehensively understand the effectiveness of the program visualization graphs presented in this paper in a real-world scenario, a trial was conducted in a computer fundamentals course aimed at freshmen, involving approximately 100 students across two classes. The trial included the use of three functionalities: Basic Program Graphs (only containing preliminary modularized control flow graph), Programming Thinking Graphs, and Code Vulnerability Graphs. The primary model used during the trial was OpenAI GPT-4(Turbo). Subsequently, surveys were designed on a chapter-by-chapter basis, allowing students to evaluate the correctness and effectiveness of each functionality, and to obtain comparative evaluations against ChatGPT4 and teaching assistants, as shown in Figure 7.

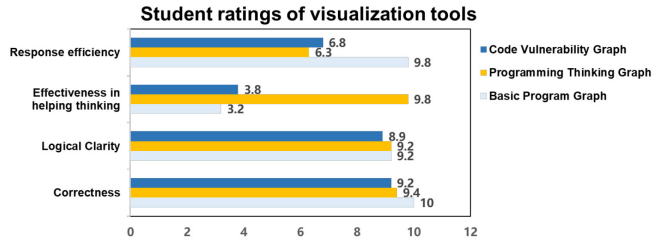

Figure 7 Student ratings of visualization tools

In addition, we calculated the average scores of students in the tests and exams for the semester this year and last year. Each test and exam consists of 10 programming questions, as shown in Figure 8. Our test arrangement is as follows: the first four tests are basic questions, the purpose is to help students get started quickly, the last four tests are 7 basic questions, the purpose is to use classroom knowledge to solve problems, and the other 3 questions are more complex and require some algorithmic thinking. The ratio of easy questions to difficult questions in the midterm and final exams is 9:1, but the time limit is stricter.

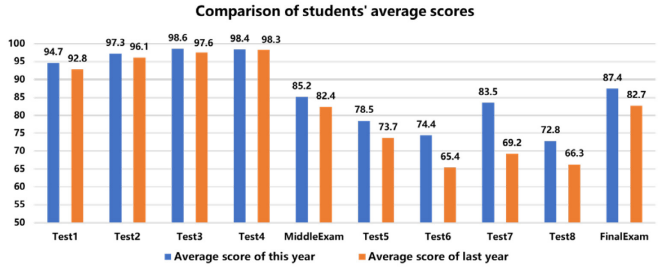

Figure 8 Comparision of students' average scores

Additionally, it was observed that students primarily used basic program graphs and code correction graphs when initially introduced to programming, as beginners primarily needed to focus on correcting syntactic errors. After several sessions, students gradually increased their usage of programming mindset graphs. Furthermore, assessments from the Online Judge system indicate a significant increase in students' attempts at solving more difficult problems compared to previous years, demonstrating the effectiveness of the tool in stimulating student interest and overcoming difficulty aversion.

IV. CONCLUSION

This paper, based on large language models (LLMs), optimized various code analysis methods and LLM fine-tuning approaches, combined with visualization techniques to produce programming mindset visualization graphs and code vulnerability visualization graphs. The main contributions of this paper are:

Using static code analysis methods to extract code structure information and optimizing LLMs in conjunction with the chain-of-thought methods, allowing LLMs to explain each interpretable part of the code from top to bottom and displaying programming mindset visualization graphs on modularized code graphs. Experiments were designed to validate the effectiveness of these optimization methods on commonly used LLMs both domestically and internationally, with experimental results indicating that LLM optimization methods, especially the use of CoT, are particularly effective in analyzing programming mindset from code.

Using program analysis, clustering analysis, and other methods to optimize LLMs, accurately identifying vulnerabilities in student code, and displaying them for localization on code flow diagrams to obtain code vulnerability visualization graphs. The achievements of (1) and (2) were applied and evaluated in practical teaching scenarios. Evaluation results show that both types of program behavior graphs were highly popular among students and played an important role for novice programmers in their different learning stages.

In summary, this paper has achieved results but also shortcomings. Future research will focus on further improving the accuracy ofLLMs, avoiding model illusions as much as possible, and applying the results to other learning stages in programming learning.

ACKNOWLEDGMENTS

This work was supported by the National Key Research and Development Project of China (2021ZD0110700) and National Natural Science Foundation of China (No.62107002) and the Engineering Research Center of Integration and Application of Digital Learning Technology, Ministry of Education (1321008).

REFERENCES

[1] Zhang, Ling, James D. Basham, and Sohyun Yang. "Understanding the implementation of personalized learning: A research synpaper." Educational research review 31 (2020): 100339.
[2] Alam, Ashraf. "The Secret Sauce of Student Success: Cracking the Code by Navigating the Path to Personalized Learning with Educational Data Mining." 2023 2nd International Conference on Smart Technologies and Systems for Next Generation Computing (ICSTSN). IEEE, 2023.
[3] Alamri, Hamdan, et al. "Using personalized learning as an instructional approach to motivate learners in online higher education: Learner self-determination and intrinsic motivation." Journal of Research on Technology in Education 52.3 (2020): 322-352.
[4] Waisberg, Ethan, et al. "Large language model (LLM)-driven chatbots for neuro-ophthalmic medical education." Eye (2023): 1-3.
[5] Li, Qingyao, et al. "Adapting Large Language Models for Education: Foundational Capabilities, Potentials, and Challenges." arXiv preprint arXiv:2401.08664 (2023).
[6] Kazemitabaar, Majeed, et al. "CodeAid: Evaluating a Classroom Deployment of an LLM-based Programming Assistant that Balances Student and Educator Needs." arXiv preprint arXiv:2401.11314 (2024).
[7] Kazemitabaar, Majeed, et al. "How novices use LLM-based code generators to solve CS1 coding tasks in a self-paced learning environment." Proceedings of the 23rd Koli Calling International Conference on Computing Education Research. 2023.
[8] Hundhausen, Christopher D., and Jonathan L. Brown. "What You See Is What You Code: A "live" algorithm development and visualization environment for novice learners." Journal of Visual Languages & Computing 18.1 (2007): 22-47.
[9] Zhang, Yan, Sheela Surisetty, and Christopher Scaffidi. "Assisting comprehension of animation programs through interactive code visualization." Journal of Visual Languages & Computing 24.5 (2013): 313-326.
[10] Oktavia, Tanty, Harjanto Prabowo, and Suhono Harso Supangkat. "The comparison of MOOC (massive open online course) platforms of edx and coursera (study case: Student of programming courses)." 2018 International Conference on Information Management and Technology (ICIMTech). IEEE, 2018.
[11] Zhang, Yue, et al. "Siren's song in the AI ocean: a survey on hallucination in large language models." arXiv preprint arXiv:2309.01219 (2023).
[12] Rawte, Vipula, Amit Sheth, and Amitava Das. "A survey of hallucination in large foundation models." arXiv preprint arXiv:2309.05922 (2023).