ingegneria sismica
International Journal of Earthquake Engineering
Anno XLIII - Num. 1 - 2026

Application of Program Code Semantic Understanding Model Based on Transformer Architecture in Automatic Grading of Data Structure Course

Mingxing Zhu[1,*] *and Xin Guo*[1]

[1] Zhixing College of Hubei University, Wuhan, Hubei, 430011, China

SUMMARY: *In order to solve the problem of "testable results but difficult to describe semantics" in automatic scoring of data structure course, this paper constructs a method for semantic understanding and automatic scoring of program code based on Transformer architecture. In this study, student submitted code, problem constraints, test results and structural features are integrated into the unified processing flow. Through code preprocessing, semantic coding, reference semantic alignment and multidimensional score aggregation, the joint evaluation of program correctness, implementation quality and structural rationality is realized. In the system design, a test set organization mechanism and a scoring feedback link for data structure questions are established, so that the automatic evaluation is no longer limited to input-output comparison, but can further identify the implementation differences in typical tasks such as linked lists, trees, and graphs. The experimental results based on 1184 valid program samples show that the Accuracy of the proposed method reaches 0.907, Macro-F1 is 0.889, QWK is 0.926, and RMSE is reduced to 3.84. The overall performance is better than that of rule scoring and multiple comparison models. The research shows that the introduction of Transformer code semantic modeling into the course evaluation process can effectively improve the accuracy, stability and teaching adaptability of automatic scoring.*

KEYWORDS: *Transformer; Program code semantic understanding; Automatic scoring; Data Structure Course*

1 Introduction

The teaching evaluation of programming courses has long been faced with the gap between "correctable" and "understandable". For the data structure course, the code submitted by students should not only output correct results, but also reflect the real grasp of knowledge such as linear tables, trees, graphs and sorting search. If the scoring process only stops at the level of example comparison or rule matching, it is often difficult to distinguish between two fundamentally different programs: "accidentally correct results" and "reasonable semantic implementation". Messer et al. pointed out in their systematic review of automatic scoring tools for programming education that the value of automatic evaluation is not only to reduce the burden of teachers, but also to build a sustainable, feed-back and extensible teaching evaluation mechanism [15]. This judgment shows that if the automatic scoring system is to truly serve the course teaching, it should not only make the judgment of right and wrong at the execution level, but also have the ability to analyze the structure and semantics of the

*mxingz@163.com
https://doi.org/10.65102/is2026091 Published: 30 April, 2026

code.

In recent years, the development of code pre-trained models has provided new technical paths for this problem. UniXcoder proposed by Guo et al. proves that cross-modal pre-training can absorb both syntactic and semantic information in code representation learning, thereby improving program understanding [1]. Wang et al. further demonstrated the good transfer ability of large-scale encoder-decoder model in code understanding and generation tasks in CodeT5+ [6]. Yang et al. pointed out through empirical research that structural feature modeling in the Transformer framework has a direct impact on code semantic summarization [11]. These studies show that source code is no longer just executable text, but can be transformed into computable, comparable, and modelable semantic objects.

Based on this, this paper introduces the program code semantic understanding model of Transformer architecture into the automatic scoring scene of data structure courses, and constructs a complete process around "topic requirements-student code-semantic expression-scoring output". This paper does not attempt to replace teacher judgment with models, but hopes to enhance the system's ability to identify the programming idea, structural rationality and implementation quality through code representation learning, feature alignment and multi-dimensional scoring mechanism, so as to promote the course evaluation from simple result inspection to semantic analysis level.

1.1 Typical application scenarios of automatic scoring system for Data structure courses

In the course of data structure, the typical application of automatic scoring system is not to simply send student code into the compiler and give a pass or fail binary conclusion, but to form a relatively complete teaching processing chain under the synergy of course platform, evaluation engine and code semantic model. The most common running mode is that the teacher establishes the experimental task according to the teaching week, sets the subject description, input and output specifications, time and space constraints, test data grouping, and the evaluation baseline of the reference implementation. The system then organizes this information into callable evaluation units and opens the entry point for submission to students. Mishra and Edwards' research on programming exercise markup language showed that the standardized expression of problem description, test rules and feedback structure could significantly reduce the deployment cost of automatic evaluation tools in courses [17]. This is particularly critical for data structure courses, because linked list manipulation, binary tree traversal, and graph search and sorting algorithms often involve boundary conditions, procedural constraints, and implementation styles at the same time, and a single output inspection is difficult to cover the real teaching needs.

After students submit programs, the system usually completes lexical cleaning, compilation execution and basic correctness verification first, and then further enters the stage of code representation and semantic analysis. The point here is not just to determine whether a program "will run", but to identify whether its implementation path has the same semantic direction as the problem requirement. For example, in tasks such as queue simulation, hash collision handling or shortest path solving, two programs with the same output may adopt completely different structural strategies, and their time complexity, data organization and key statement dependencies may not be consistent. Oli et al pointed out in their research on automatic code comprehension evaluation that large language model and code semantic modeling technology can capture students' mastery of program logic to a certain extent, so as to make automatic evaluation move from the result level to the understanding level [19].

Therefore, in the scenario discussed in this paper, the Transformer model assumes the function of an intermediate interpretation layer, which transforms the source code into a comparable semantic representation and combines the test results, structural features, and implementation quality to generate a composite score.

From the perspective of teaching use, the participants in the system also have a clear division of labor. Teachers are responsible for the problem configuration, scoring weight setting and result review, students are responsible for submitting, viewing feedback and iterative modification, and course managers can use the platform to calculate the class pass rate, common error types and problem discrimination. Mitra pointed out in his research on instant feedback automatic raters that the quickly returned evaluation results would directly affect students' debugging behavior and practice engagement [18]. Dong and Liang proposed that the interpretability of programming assignment scoring can be improved by program summarization and semantic compression [20]. In this application mode, automatic scoring system is no longer just a substitute for teacher's marking task, but gradually becomes a data-based teaching interface connecting program analysis, learning diagnosis and curriculum improvement.

1.2 Related Research

Although a rich accumulation of technologies has been formed around the research of automatic scoring of programming courses at the present stage, the work on the integration of code semantic understanding and scoring of data structure courses is still insufficient. Existing research results can be roughly classified into three categories: evaluation platform research, machine learning scoring research and code semantic modeling research, with different focuses and obvious differences in application boundaries, as shown in Table 1.

Table 1: Related research paths and their applicable characteristics

Research Category	Representative Studies	Main Functions	Applicable Advantages	Limitations
Automated Assessment Platforms	Messer et al. [15]; Cipriano and Alves [21]	Compilation, execution, judging, and feedback management	Suitable for large-scale course deployment, with a mature workflow	Focuses mainly on result verification, with insufficient semantic recognition
Teaching Feedback and Machine Learning-based Scoring	Mitra [18]; Messer et al. [16]	Instant feedback, learning behavior support, and data-driven scoring	Helps improve practice efficiency and evaluation flexibility	Limited cross-question generalization ability, and stability is affected by data quality
Code Semantic Modeling	Guo et al. [1]; Wang et al. [6]; Xiao et al. [14]; Mondal et al. [10]	Code representation learning, semantic understanding, and context modeling	Capable of capturing implementation logic and structural information	Mostly used in software engineering tasks and still needs deeper integration with course grading mechanisms

Messer et al. pointed out in their review that automatic scoring and feedback tools have been able to complete program compilation, test case execution, result comparison and basic feedback generation relatively stably, and such systems have obvious advantages in the expansion of teaching scale [15]. Cipriano and Alves combined with years of practical experience in automatic evaluation to further explain that platform-based evaluation can improve the efficiency of course operation, but its scoring basis still relies heavily on preset rules and test coverage, and the interpretability is still insufficient in the face of program tasks with diverse implementation paths [21]. This means that traditional automated scoring systems are better at answering the question "did the program pass?" but less at answering the question "why did the program score?"

In the level of teaching application, Mitra pays attention to the influence of instant feedback automatic raters on students 'practice behavior, and believes that rapid return of results helps to form a high-frequency debugging cycle [18]. Messer et al. 's meta-analysis of machine learning automatic scoring tools shows that data-driven methods are beginning to be used to identify program quality differences, but the generalization ability between different courses and different question types is not stable [16]. For the course of data structure, this problem is particularly prominent, because the same problem often allows multiple legal implementations, chain storage, sequential storage, recursive expansion and iterative rewrite may all lead to the same output, and it is difficult to fully describe the depth of understanding of students only by external behavior.

Compared with the above studies, the development of code pre-trained models provides a more fine-grained analysis basis for automatic scoring. Guo et al proposed UniXcoder, which incorporated code, comments and cross-modal information into a unified representation space, which significantly enhanced the program semantic representation ability [1]. Wang et al. verified the transfer potential of large-scale pre-trained models on code understanding tasks in CodeT5+ [6]. Xiao et al. empirical study on Transformer code technology shows that such models have shown strong adaptability in program analysis, summarization, and understanding tasks [14]. Furthermore, Mondal et al. specifically evaluated the performance of Transformer model in code semantic understanding, pointing out that it has better modeling ability for context dependencies and logical relationships [10]. However, these studies focus on software engineering tasks and are not directly equivalent to teaching grading systems themselves. It can be seen that how to embed the semantic understanding ability of Transformer code into the automatic scoring process of data structure course and jointly model it with test results, structural features, and teaching evaluation criteria is still worthy of research.

2 Task definition and system process of automatic scoring for data structure course

In the scenario of data structure course, automatic scoring is not a single-step task that students' programs are directly given to the compiler to run, and then the pass rate is returned according to a number of test points. In essence, it is a composite process consisting of problem definition, input-output modeling, submission management, program execution, semantic representation, result summarization and feedback generation. Without a clear task boundary and a unified data organization, the subsequent Transformer code semantic understanding model is difficult to be stably embedded into the teaching system even if it has strong representation ability. Therefore, this section first formally defines the automatic scoring task, and then explains the organization and operation process of each object in the

system.

From the perspective of course objectives, the scoring object of data structure topics is not simply "program text", but the executable implementation given by students around specific problems. A complete grading task contains at least five types of core information: problem description, test data, reference constraints, student submission, and grading output. Problem statements are used to limit problem boundaries, such as sequential table insertion, binary tree hierarchical traversal, shortest path in a graph, or hash table conflict handling. Test data is used to trigger boundary branches and complex situations. The reference constraints include time complexity expectation, space occupation range, function interface specification and whether recursion is allowed. Student submissions consist of source code, language type, submission time, and version number. The scoring output is the result object generated by the system for a submission, which contains not only the compilation status, running status and test passing, but also the semantic matching degree, structural rationality and abnormal risk tips. It can be seen that the automatic scoring task in this study can be understood as a mapping process from "question specification - code implementation - multi-dimensional evaluation results", rather than a single judging action.

In order to make this task run stably in the teaching platform, the system needs to define a unified project object first. Each project corresponds to an exercise or experiment problem, and the project manager is usually undertaken by the course teacher or the course platform configuration module. Its responsibility is not to manually view the code one by one, but to complete the task template configuration in advance, so that the subsequent scoring process can be automatically executed. Content such as question number, supported languages, input and output formats, test set location, resource limits, and scoring rules need to be clarified in the project object. A simplified project definition can be written as Code 1.

Code 1 Sample project configuration object

```
public class AssignmentSpec {
    public String assignmentId;
    public String title;
    public String language;
    public int timeLimitMs;
    public int memoryLimitMb;
    public String entryClass;
    public List<TestSetSpec> testSets;
    public ScorePolicy scorePolicy;
}
```

At the execution level, test case is the smallest unit of data generation, and test suite is the basic running unit of system scheduling. For the data structure course, the design of test cases should not only cover normal inputs, but also consciously include empty structures, repeated elements, extreme scales, illegal boundaries and degenerate structures. For example, a linked list problem needs to detect empty list insertion, head deletion, and tail concatenation. Tree structure problems need to detect single-node trees, completely unbalanced trees, and duplicate keywords. Graph algorithms problems need to detect outliers, heavy edges, unreachable nodes, and sparse-dense graph switching. Automatic scoring results are stable only if these states are coded in advance as test objects that can be called repeatedly. The abstraction of test input and expected output can be written as code 2.

Code 2 Sample test case object

```
public class DSTestCase {
    public DSInput input;
    public DSOutput expectedOutput;
```

```
        public String group;
        public int weight;
    }
    public class DSInput {
        public int n;
        public int[][] edges;
        public int[] values;
        public String mode;
    }
    public class DSOutput {
        public String textOutput;
        public int[] sequenceOutput;
        public boolean success;
    }
```

Based on this, the test set file needs to support batch definition. Different from the simple judgment system that only stores random examples, the test set of data structure course is more suitable to be organized in the way of "group + parameter", so that the system can distinguish different evaluation objectives such as basic correctness, boundary robustness and performance pressure. A test set file can describe the test number, scale, data distribution and question type pattern by line, and the system can regenerate the actual input after reading. An example is shown in code 3.

Code 3 Sample test set profile

```
case1:LIST:10:RANDOM
case2:LIST:1:EMPTY
case3:TREE:31:COMPLETE
case4:TREE:31:SKEWED
case5:GRAPH:100:SPARSE
case6:GRAPH:100:DENSE
case7:GRAPH:1000:STRESS
```

When the system parses the test suite, it should not treat each line as just a string record, but it should be converted into structured parameters that can be passed to the test factory to generate standard input objects. This process determines the consistency of the subsequent execution. If the test generation logic lacks a unified interface, there will be some problems between different problems, such as the confusion of sample organization, the lack of boundary conditions and the unrepeatable evaluation. An implementation of the test factory is shown in Code 4.

Code 4 Sample test case generation method

```
    public class DSTestFactory {
        public DSTestCase generateTestCase(String line) {
            String[] parts = line.split(":");
            String id = parts[0];
            String type = parts[1];
            int size = Integer.parseInt(parts[2]);
            String mode = parts[3];

            DSInput input = new DSInput();
            input.n = size;
            input.mode = mode;
```

```
            if ("TREE".equals(type)) {
                input.values = buildTreeValues(size, mode);
            } else if ("GRAPH".equals(type)) {
                input.edges = buildGraphEdges(size, mode);
            } else if ("LIST".equals(type)) {
                input.values = buildListValues(size, mode);
            }

            DSTestCase testCase = new DSTestCase();
            testCase.input = input;
            testCase.expectedOutput = solveReference(input, type);
            testCase.group = mode;
            testCase.weight = getWeight(mode);
            return testCase;
        }
    }
```

When the project definition and the test set are ready, the system needs to process the student submission. A commit record should not only hold source code text, but should also be accompanied by information such as language type, compilation result, run log, abstract syntax tree summary, and semantic embedding index. The reason is that our automatic scoring does not place the Transformer model on the periphery of the system, but instead involves it in the formal scoring link. In other words, after the student code is compiled, it not only enters the traditional executor, but also enters the code preprocessing and presentation module. This module is responsible for word segmentation, identifier normalization, statement block segmentation, control flow fragment extraction and necessary syntax tree mapping, and then the results are sent to the semantic encoder. The goal is not to replace the test results, but to make up for the unexplained parts of the test results. For example, both programs work through the examples, but one implements a stack structure in a sequence table, and the other outputs a large number of hardcoded conditions. From the perspective of teaching evaluation, the two should not get the same judgment.

Therefore, the system flow should be designed as a two-channel structure. One channel is responsible for performing layer verification, compiling, running, comparing results and resource detection. The other channel is responsible for semantic layer analysis to complete code standardization, Transformer encoding, reference de-alignment, and structural anomaly identification. The intermediate results of the two channels are finally merged into the score aggregation module to generate a comprehensive score and feedback text. The execution object inside the system can be abstracted as the following interface.

Code 5 Sample evaluation execution interface

```
public abstract class AbsSubmissionEvaluator {
    public abstract CompileResult compile(SubmissionRecord submission);
    public abstract RunResult runOnTest(SubmissionRecord submission, DSTestCase
testCase);
    public abstract SemanticResult analyzeSemantics(SubmissionRecord submission);
    public abstract FinalScore aggregate(
        CompileResult compileResult,
        List<RunResult> runResults,
        SemanticResult semanticResult
    );
}
```

In the process order, the system usually performs compilation and security checks first. The commit with compilation failure directly enters the error feedback branch, but its code can still be retained for subsequent teaching statistics. Successfully compiled commits enter the test set scheduler, which executes the tests in groups. The basic group mainly examines functional correctness, the boundary group examines exception handling and detail stability, and the pressure group examines efficiency and resource occupancy. At the same time, the semantic analysis module generates code representations asynchronously and matches them with standard solutions, excellent samples, or problem target representations. If a program performs well in the execution result, but the gap between the implementation and the reference implementation is too large, the system can mark it as "the result is correct but the implementation quality is weak". On the contrary, if the program fails on some examples, but the semantic structure is close to the target idea, it can provide a basis for teachers to review. This design avoids a long-standing problem with automated scoring: looking only at the external behavior and not asking about the internal implementation.

In order to make the scoring results serve teaching, rather than just form a ranking, the system output should also adopt the form of object. The result object should contain at least compilation status, test pass rate, timeout times, memory exception times, semantic similarity score, structure specification score and feedback description. A simplified result structure is shown in Code 6.

Code 6 Sample score result object

```
public class FinalScore {
    public double correctnessScore;
    public double efficiencyScore;
    public double semanticScore;
    public double structureScore;
    public double totalScore;
    public String feedback;
}
```

In teaching practice, this result object can also continue to connect the course analysis module upward. Teachers can identify common errors based on the result distribution at the class level, such as a large number of students losing points on the recursive termination condition of the tree, or semantic shifts in the adjacency list construction stage of the graph. The platform can also track the difficulty of questions, the coverage quality of test sets, and whether the scoring rules are reasonable. In other words, the system doesn't end when a student gets a grade, but continues to flow back to course improvement. To sum up, the core of automatic scoring task of data structure course is not to simply transplant the existing online judgment logic to the teaching platform, but to establish a set of task organization methods for program semantic understanding. The project definition is responsible for fixing the scoring boundary, the test case and test set are responsible for providing stable input, the submission record is responsible for carrying the program object, the execution channel is responsible for verifying the external behavior, the Transformer semantic module is responsible for describing the internal implementation, and the aggregation module transforms these heterogeneous information into unified teaching evaluation results. Only after this process is clearly defined, the subsequent program code semantic understanding model and its scoring method based on the Transformer architecture have a realistic basis for stably landing in the automatic scoring scene of data structure courses.

3 Program code semantic understanding model and scoring method based on Transformer architecture

If the automatic grading in data structure course still remains at the level of input-output comparison, it can only judge whether the program produces the expected results under the given test, but it is difficult to identify whether students truly understand the dependencies between abstract data types, algorithm processes and key operations. Especially in linked list operation, tree traversal, graph search, hash table construction and other problems, different programs may have similar external results, but their internal implementation paths, control logic and data organization are different. To this end, the Transformer architecture is introduced into the scoring process, and the source code is not regarded as a simple text to be executed, but as a decompose, encodable, and comparable semantic object. Besides the test results, a program understanding mechanism is added, so that the scoring process covers two dimensions of "whether to do it right" and "how to do it right".

The model input is composed of student code submission, problem description, function signature and reference constraints. Considering that the program code contains lexical information, statement order information and structure dependence information at the same time, this paper first completes tag segmentation, comment cleaning, identifier normalization and structure fragment extraction in the preprocessing stage. For variable names, function names and data structure names, not all of them are directly anonymized, but some naming features with teaching significance are retained, such as push, pop, enqueue, dfs, left, right, etc., to avoid losing students' explicit expression of data structure semantics. Subsequently, the code sequence is mapped to an embedding vector, which is added to the position encoding to obtain the initial model input:

$$X = [x_1, x_2, \ldots, x_n], \quad x_i = e(t_i) + p_i \tag{1}$$

Here, t_i denotes the ith code token, $e(\cdot)$ denotes the token embedding function, and p_i is the position encoding.

In the encoding stage, the model uses the multi-head self-attention mechanism to model the long-distance dependence. For data-structured programs, the meaning of a judgment statement is often not only determined by the local context, but also related to initialization, node access, recursive exit, and return value organization. Multi-head attention can map these cross-sentence relationships into a unified representation space. Its core calculation form is as follows.

$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^{\top}}{\sqrt{d_k}}\right)V \tag{2}$$

$$\mathrm{MultiHead}(Q, K, V) = \mathrm{Concat}(\mathrm{head}_1, \ldots, \mathrm{head}_h)W^{O} \tag{3}$$

where Q, K, V are the query, key, and value matrices, respectively, d_k is the scaling factor, and h is the number of attention heads. After stacking multiple layers, the model obtains the context-aware representation $H = [h_1, h_2, \ldots, h_n]$ of the whole program.

Unlike the general code comprehension task, the course grading scenario has a stronger focus on "locally critical statements". For example, the linked-list insertion problem is not about the print statement, but about relinking the Pointers. The tree traversal problem is not in the container declaration, but in the recursive access order. The key of graph shortest path problem is focused on priority queue update and distance relaxation process. Therefore, this

paper adds a semantic aggregation layer to the top of Transformer to reallocate the weights of key statements and obtain the program level vector representation z:

$$\alpha_i = \frac{\exp(w^\top h_i)}{\sum_{j=1}^{n} \exp(w^\top h_j)}, z = \sum_{i=1}^{n} \alpha_i h_i \tag{4}$$

Here, α_i represents the contribution weight of the ith code unit in the overall semantics. This mechanism makes the model not be diluted by a large number of template statements, and is more conducive to capturing the implementation core which is highly related to scoring.

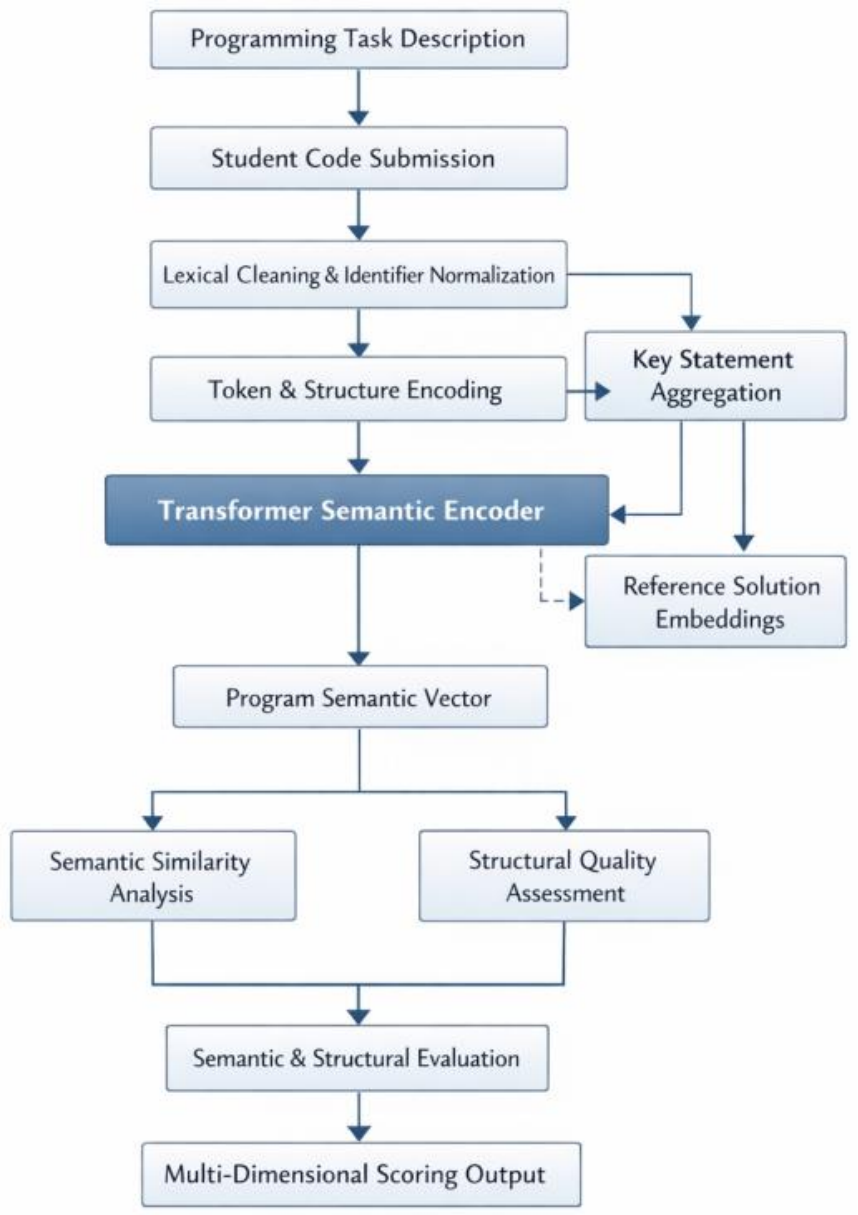

Figure 1: Flow chart of Transformer-based program code semantic understanding and scoring

As shown in Figure 1, code execution and code understanding are not two separate processes in the system, but converge at the semantic vector level. Once the program is coded,

it can be compared with the reference implementation or matched with high-quality student samples or subject target vectors, thus providing a finer discrimination basis for subsequent scoring.

The program representation alone is not enough to directly form a scoring conclusion. The system also needs to judge the correspondence between the student's code and the goal of the question. Therefore, this paper constructs a reference semantic library, which is composed of teacher reference solutions, selected excellent samples and typical error samples. The reference semantic library does not seek to have only one correct path, but retains semantic clusters of multiple legal realizations. For example, a binary tree hierarchical traversal problem can contain both the standard queue-based implementation and a small number of structurally different but semantically consistent variants. Topological sorting problems may allow in-degree queue method and DFS inverse post-order method to coexist. The purpose of this treatment is to avoid narrowing the course grading error to "the more similar to a single standard answer, the higher grade".

The similarity between the student program vector z_s and the reference semantic cluster center $z_r^{(k)}$ is calculated by cosine distance:

$$Sim(z_s, z_r^{(k)}) = \frac{z_s^\top z_r^{(k)}}{\|z_s\| \cdot \|z_r^{(k)}\|} \tag{5}$$

The system selects the maximum similarity as the semantic matching score:

$$S_{sem} = \max_k Sim\,(z_s, z_r^{(k)}) \tag{6}$$

If the performance of the program is normal at the output level and the S_{sem} is high, it can be determined that the realization idea has strong consistency with the course objectives. If the test passes but the semantic match value is low, it means that the code may have problems with hard coding, example fitting, or non-target structure substitution. This is not uncommon in automated grading, especially in the context of basic questions and online submission.

In order to further constrain the program structure, this paper designs a structure quality identification module. This module does not directly participate in semantic vector encoding, but reads abstract syntax trees, control flow fragments, and complexity approximation metrics to judge the structural soundness in the student's implementation. For the data structure course, we should pay attention to at least four kinds of phenomena: one is too many invalid branches, indicating that the logical organization is loose; Second, the key update statement is missing, indicating that although the implementation works, the core data relationships are not properly maintained. The third is abnormal use of brute force traversal instead of the target structure, such as frequent linear array scanning when the stack should be used; Fourth, there is potential instability in recursive exits, boundary conditions, or pointer updates. The structural quality score can be expressed as follows.

$$S_{str} = 1 - \lambda_1 r_{red} - \lambda_2 r_{mis} - \lambda_3 r_{irr} - \lambda_4 r_{risk} \tag{7}$$

where r_{red} is the redundant structure rate, r_{mis} is the critical statement missing rate, r_{irr} is the realization offset rate that does not match the topic goal, r_{risk} is the potential risk ratio, and λ_i is the corresponding penalty weight. This formula does not require accurate simulation of program complexity, but for the purpose of curriculum evaluation, it approximately describes the stability and rationality of the implementation structure.

In the scoring link, this paper does not use a single index to directly map the total score, but jointly models the result correctness, execution efficiency, semantic consistency and structural quality. The reason is that "high-quality solutions" in data structures courses don't just mean passing tests; If a student implements a simple queue operation with a large number of redundant statements, or struggles to complete a task in an inefficient way in a graph algorithm problem, such programs should not be evaluated the same as code that implements a clear and structured matching.

Suppose the test correctness score is S_{cor}, the efficiency score is S_{eff}, the semantic matching score is S_{sem}, and the structure quality score is S_{str}, then the comprehensive score is defined as follows.

$$S = \omega_1 S_{cor} + \omega_2 S_{eff} + \omega_3 S_{sem} + \omega_4 S_{str} - P \tag{8}$$

Here, $\omega_1 + \omega_2 + \omega_3 + \omega_4 = 1$ and P is the anomaly penalty term. According to the evaluation characteristics of data structure course, this paper puts the correctness of results in the dominant position, but does not give the absolute only position. The introduction of semantic and structural items enables the system to make a finer distinction between "passed but not standardized" and "not completely passed but basically correct". The penalty term P is mainly used to deal with cases such as timeouts, memory out-of-bounds, illegal calls, too strong copy traces, or obvious hard-coded features.

The efficiency score itself is not directly represented by the original running time value, but is processed by piecewise normalization to reduce the interference caused by scale differences between different question types. If the time consumption of the MTH test group is T_m and the corresponding reference threshold is τ_m, then:

$$S_{eff} = \frac{1}{M} \sum_{m=1}^{M} min\left(1, \frac{\tau_m}{T_m + \epsilon}\right) \tag{9}$$

Here, M is the number of test groups and ϵ is the smoothing term. This avoids a few extreme test points having a disproportionate impact on the overall score.

In order to enhance the teaching interpretability of the scoring results, the system does not only output a numerical value, but generates structured feedback synchronously. The feedback template is triggered by the result of model analysis, such as "the basic logic is correct, but the update of the linked list head is unstable", "the output result passes, and the stack operation loses its structural characteristics due to multiple loop scans", "the recursive termination condition is reasonable, and the tree traversal order is consistent with the title", etc. Compared with the traditional automatic judgment, this output method is closer to the teacher's marking language, and it is more helpful for students to locate the source of the problem.

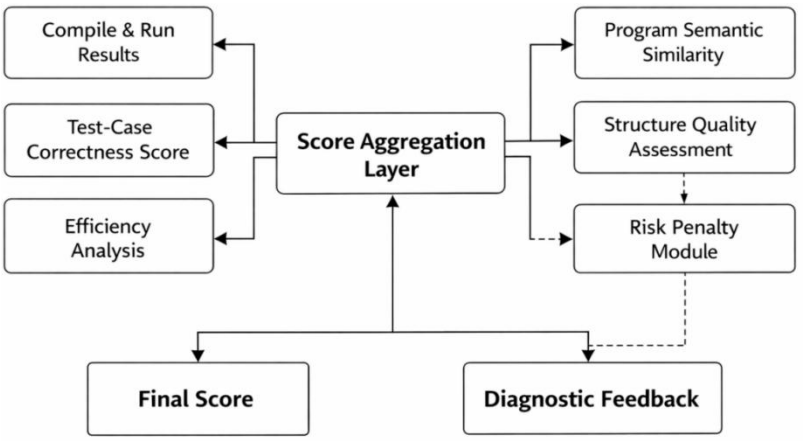

Figure 2: Structure diagram of multidimensional rating aggregation and feedback generation

As shown in Figure 2, the indicators in the scoring system are not substitutes for each other, but are pooled together into the aggregation layer. The results of the execution layer ensure the bottom line of the score, the semantic layer and the structure layer improve the resolution of the score, and the risk punishment module is responsible for correcting abnormal situations. This design not only retains the efficiency advantage of automatic grading in large-scale teaching, but also prevents it from becoming a mechanized output comparison tool.

At the same time, the key reason why this method is suitable for the data structure course is that it is not dealing with the abstract code text classification problem, but the program understanding problem driven by the course goal. The teaching of data structure emphasizes not only programming proficiency, but also the grasp of the internal laws of linear structure, tree structure and graph structure. How nodes are connected, how the traversal order is maintained, how auxiliary containers are used, and how boundary states are handled are the most obvious parts of the student code. The Transformer model has strong representation ability for long-distance dependencies and context relations, which enables it to cross the local sentence level and capture the semantic patterns that are really related to the intention of the algorithm in the program. Multidimensional grading method transforms this pattern into quantitative results that can be embedded in course evaluation.

4 Analysis of experimental results

In order to test the actual performance of the model built in this paper in the automatic marking of data structure course, the experimental system will generate a structured marking record for each submission after the completion of student program compilation, test execution, semantic coding and result aggregation. The records include question number, submission language, test group pass rate, average running time, semantic match score, structure quality score, composite score, and deviation value from teacher rating. This output is more suitable for statistical analysis than traditional pass/fail states because it captures the external behavior of the program while preserving fine-grained differences at the implementation level. Based on these records, this paper analyzes the overall scoring effect, the performance of different types of questions, the ablation results and the operation efficiency.

The experimental data are from the stage assignments and experimental submissions of two teaching classes of data structure course in a university, and a total of 1268 program samples are collected. After removing the invalid samples that could not be compiled and had

abnormal code length, 1184 valid submissions were retained, covering seven typical tasks such as sequence list, linked list, stack and queue, binary tree traversal, graph search, sorting and lookup. The manual scoring was completed independently by three teachers with teaching experience, and then the weighted average was taken as the reference label. Accuracy, Macro-F1, Quadratic Weighted Kappa (QWK) and RMSE were used to measure the difference between the automatic scoring results and the manual scoring. Macro-F1 is used to observe the equilibrium recognition ability at different score levels. QWK is used to evaluate the overall consistency between automatic scoring and teacher scoring. RMSE is used to describe the discrete level of score error.

Table 2 presents the overall comparison results of the different scoring methods. It can be seen that although the scoring method only based on the passing rate of test cases has high execution efficiency, it performs generally on the teacher consistency index. After adding the code statistical features, the scoring error decreases, but it is still difficult to stably describe the internal implementation quality of the program. In contrast, the proposed method achieves better results on all four indicators, especially on QWK and RMSE, indicating that the Transformer semantic representation has a practical role in narrowing the judgment gap between automatic scoring and human scoring.

Table 2: Overall performance comparison of different automatic scoring methods

Method	Accuracy	Macro-F1	QWK	RMSE
Test Case-based Rule Scoring	0.781	0.746	0.804	7.42
Test Results + Code Statistical Features	0.823	0.791	0.846	6.18
BiLSTM-based Code Representation Scoring	0.851	0.826	0.873	5.47
AST + XGBoost Scoring	0.864	0.838	0.881	5.12
Proposed Method	0.907	0.889	0.926	3.84

From the specific results, the Accuracy of the proposed model reaches 0.907, which is 0.126 higher than that of the rule score based on test cases. Macro-F1 reached 0.889, indicating that the recognition of the model on high score, middle score and low score samples was more balanced. QWK was increased to 0.926, indicating a high consistency between the automatic scoring results and the teacher's scoring. The RMSE decreased to 3.84, indicating that the deviation between the automatic grading output and the true teacher score was compressed to the lower range. This result does not mean that test cases are no longer important, but indicates that when the external behavior of the program has been basically covered, the semantic understanding module can continue to distinguish between the samples with correct results but rough implementation and the samples with correct results and complete ideas, thus improving the resolution of the scoring.

To further observe the performance of the model in different task types, seven types of questions are divided according to data structure characteristics, and the automatic scoring consistency of various types of questions is counted, and the results are shown in Table 3. It can be found that the scoring consistency of sequence table, stack queue and sorted search questions is relatively high, which is related to the relatively regular input and output forms and the concentration of reference implementation paths. Linked lists, binary trees, and graph structures are more difficult to grade because these programs involve more complex processes of pointer updates, recursive calls, and state propagation, and student code often differs significantly in how it is written locally. Even so, the QWK of the proposed method on these problems still remains around 0.90, indicating that the understanding ability of the model for structural programs is relatively stable.

Table 3: Scoring performance on different question types

Question Type	Number of Samples	Accuracy	QWK	Mean Absolute Error
Sequential List Operations	162	0.926	0.941	2.97
Linked List Operations	171	0.891	0.912	3.88
Stack and Queue	154	0.918	0.934	3.15
Binary Tree Traversal	188	0.902	0.921	3.64
Graph Search	176	0.884	0.907	4.11
Sorting Algorithms	171	0.914	0.929	3.36
Search Algorithms	162	0.913	0.925	3.28

From Table 3, we can see that the average absolute error of the graph search questions is 4.11, which is slightly higher than the other categories. This phenomenon is directly related to the diversity of graph algorithm implementations. For example, in the breadth-first search and depth-first search questions, although the test results of some student programs are completely correct, the organization of auxiliary containers is confusing, or the access mark and node expansion process are intermixed in the same loop, which leads to the moderate deduction of structure points in the grading. Test-driven scoring methods usually fail to identify such differences, while the proposed model can be corrected by semantic representation and structural quality score, so its overall error is still controlled within an acceptable range.

FIG. 3 illustrates the comparison results of different methods on the agreement between automatic and teacher ratings. It can be seen from the figure that the rule scoring method is relatively accurate in the judgment of high-scoring samples, but once it enters the middle interval of "partially correct" or "basically valid logic but defective implementation", the misjudgment increases significantly. The distribution of the proposed method is closer to the manual scoring curve, especially in the 70-89 partition. This suggests that the value of Transformer semantic models is not mainly reflected in extremely correct or extremely wrong samples, but in those edge samples that are most prone to distortion by traditional automatic evaluation.

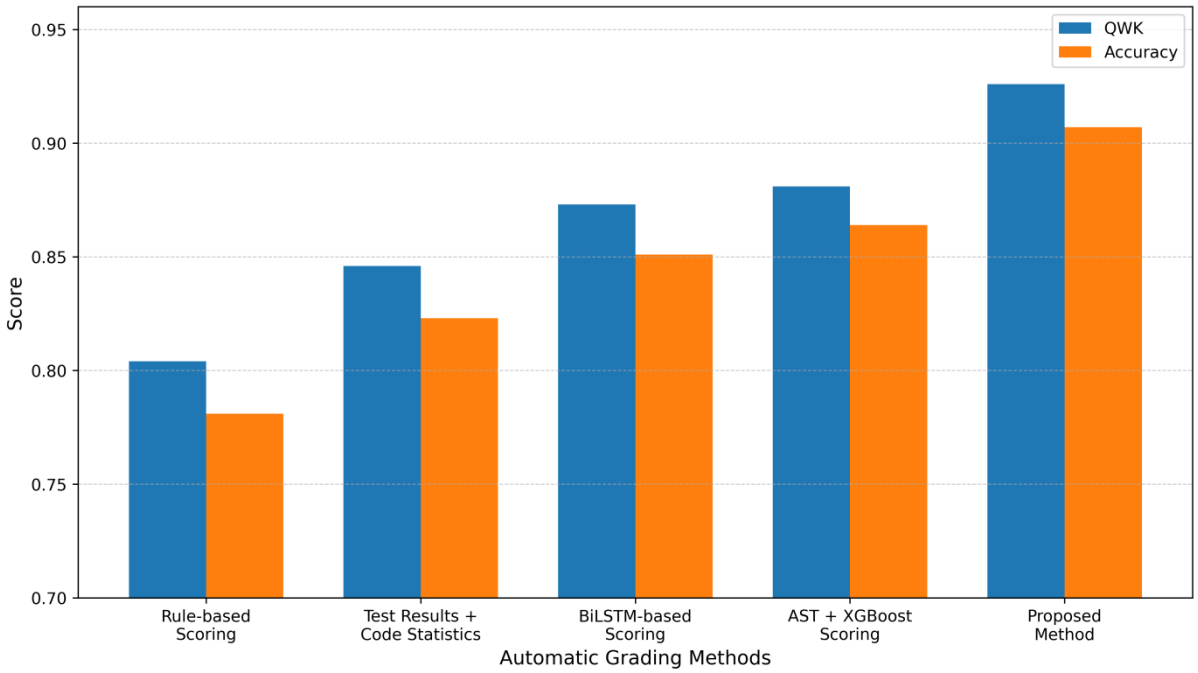

Figure 3: Comparison of consistency between different scoring methods and teacher ratings

In addition to the overall performance, ablation experiments were performed to examine the role of each component in the model. Ablation results show that QWK is 0.901 when only using Transformer encoder and test result aggregation. After removing the key sentence attention aggregation layer, QWK decreased to 0.889. When the structural quality identification module is further removed, the RMSE increases from 4.06 to 4.71. It can be seen that the semantic encoder provides basic program understanding ability, but it is the two links of key sentence weighting and structure constraint that really make the scoring closer to the teacher's judgment. The former improves the sensitivity of the model to the core implementation path, and the latter suppresses the overestimation phenomenon of "correct result but distorted structure". For the data structure course, this combination is necessary, because the course evaluation is not only oriented to the functional implementation, but also involves whether the data organization is consistent with the problem training objectives.

Figure 4 shows the distribution of prediction errors over different score intervals. The low partition sample has the smallest error, with an average absolute deviation of 2.31 points. The error of the middle interval sample is relatively higher, and the score of the 70-79 segment is 4.26. The error for samples above 90 points fell back. This trend has a strong teaching interpretation. Low score samples are usually accompanied by compilation failure, critical logic error or large area test failure, and the judgment boundary is clear. The realization of high-scoring samples is often more complete, and it is easier to form consensus among teachers. The real complexity is in the middle interval, where the program is often "the main idea is correct, but the local implementation is not stable". The automatic scoring model must refer to three types of information at the same time, such as test, semantic and structural information, to avoid excessive drift of the score. The results of this paper show that the model still maintains a relatively stable error control in this high difficulty interval.

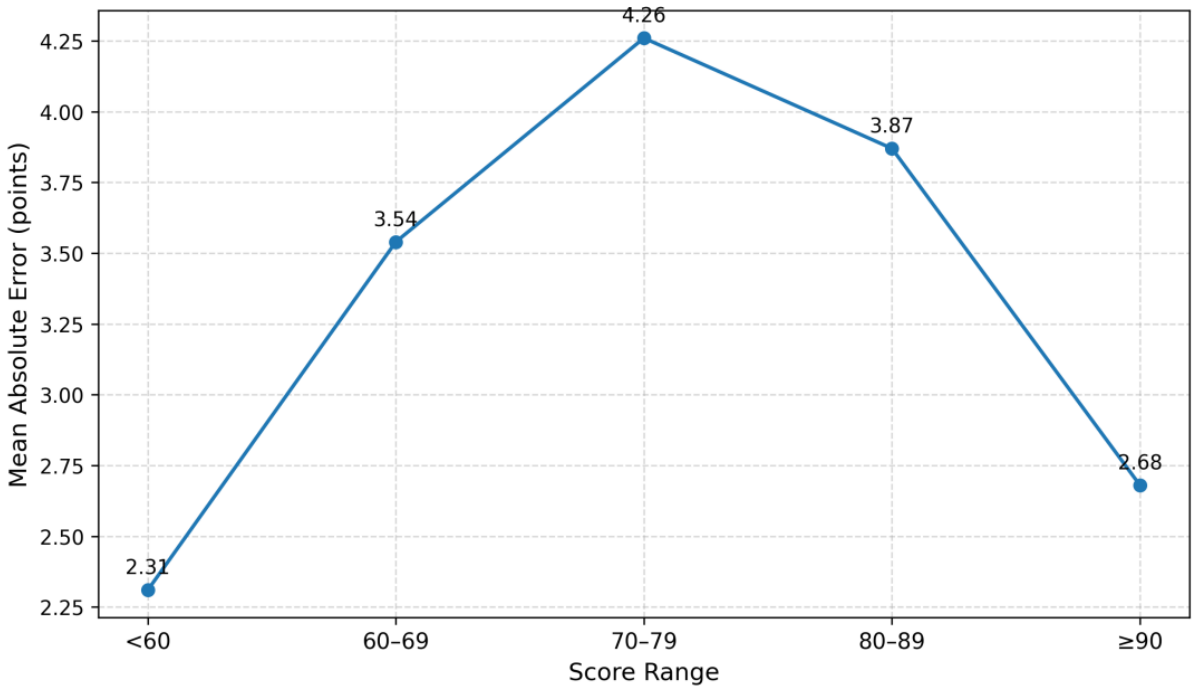

Figure 4: Distribution of mean absolute error over different score intervals

In addition to the consistency of scoring, the efficiency of system operation is also related to the feasibility of course deployment. The average processing time of a single commit is calculated under the same server environment. The average time of code compilation and test

execution is 0.84 s, the average time of semantic coding is 0.27 s, and the average time of structural quality analysis is 0.11 s. Compared with 0.79 s that only does rule judgment, the extra cost of the proposed method is 0.43 s, and the increase is controllable. In the batch grading scenario, the system can complete the complete processing of about 176 submissions per minute after using task queue concurrent scheduling, which can meet the use requirements of ordinary classroom experiments and stage assignments. In other words, the introduction of the semantic understanding module does not destroy the practicality of automatic scoring, but significantly improves the quality of scoring at an acceptable time cost.

Combined with the above results, it can be seen that the advantages of the model in this paper are not only reflected in the single numerical improvement, but also reflected in the degree of fit between the scoring logic and the needs of teaching evaluation. It can use Transformer's ability to model the context of the program to further advance the student code from "whether the output is correct" to "whether it really reflects the idea of data structure". The experimental data show that this method shows strong stability in different question types, different score ranges and different scoring indexes, which indicates that it is not suitable for a certain type of questions by chance, but has good overall adaptability for the task of automatic scoring of data structure courses.

5 Conclusion

Focusing on the automatic scoring problem of "decidable results but difficult semantics" in the data structure course, this paper constructs a scoring framework combining test execution and code semantic understanding, and introduces the program code representation model based on Transformer architecture into the course evaluation process. Compared with the methods only based on sample pass rate or rule matching, the proposed method treats student programs as both execution objects and semantic objects, and combines semantic representation, test results, structural quality information and implementation constraints into the scoring process, so as to form multi-dimensional evaluation results that are more suitable for teaching needs. The experimental results show that on 1184 valid program samples, the Accuracy of the proposed method reaches 0.907, the Macro-F1 is 0.889, the QWK is 0.926, and the RMSE is reduced to 3.84. The overall consistency is significantly better than that of rule score, statistical feature score, BiLSTM, AST+XGBoost and other comparison methods. This shows that Transformer's ability to model program context dependencies and key statement relationships can indeed improve the resolution of automatic grading of data structure courses.

Judging from the performance of different question types, the scoring results on order table, stack queue and sorted search tasks are relatively stable. Although structural problems such as linked list, binary tree and graph search have stronger implementation differences, the model still maintains high consistency, and the QWK of linked list operation, binary tree traversal and graph search reaches 0.912, 0.921 and 0.907, respectively. This means that the proposed method is not only suitable for basic problems with a single implementation path, but also has good adaptation ability for complex programs involving pointer update, recursive expansion and state propagation. At the same time, the system does not bring excessive deployment burden while maintaining the improvement of scoring accuracy. The average processing time of a single submission is 1.22 s, and the total additional overhead of semantic coding and structural analysis is only 0.38 s. Under the condition of batch concurrency, about 176 submissions can be completed every minute, which can meet the running needs of regular course assignments and experimental teaching.

The significance of the work in this paper is not only to improve the consistency between

automatic grading and teacher grading, but also to advance the grading process from "whether to do it right" to "how to do it right". The introduction of unified test set and unified semantic reference space also enables different student submissions to complete the comparison within the same evaluation boundary, which helps to enhance the fairness and reusability of scoring. Of course, our method is still affected by the type of questions, the coverage of the reference semantic base and the size of the training sample, and there is still room for further optimization when facing open programming tasks or cross-language submissions. Future research can continue to expand the multilingual code representation, error type diagnosis and interpretable feedback generation mechanism, so that the automatic scoring system can serve the teaching improvement of programming courses more deeply while maintaining computational efficiency.

Funding

This work was supported by Zhixing College of Hubei University, the 2024 University-level Teaching Reform Research Project —"The Transformation Practice of Computer Education Boosted by Generative AI Technology: A Perspective Based on Large Language Models" (Project Number: XJY202415)

References

[1] Guo D, Lu S, Duan N, et al. Unixcoder: Unified cross-modal pre-training for code representation[C]//Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). 2022: 7212-7225.

[2] Le H, Wang Y, Gotmare A D, et al. Coderl: Mastering code generation through pretrained models and deep reinforcement learning[J]. Advances in Neural Information Processing Systems, 2022, 35: 21314-21328.

[3] Fried D, Aghajanyan A, Lin J, et al. Incoder: A generative model for code infilling and synthesis[J]. arXiv preprint arXiv:2204.05999, 2022.

[4] Li Y, Choi D, Chung J, et al. Competition-level code generation with alphacode[J]. Science, 2022, 378(6624): 1092-1097.

[5] Neelakantan A, Xu T, Puri R, et al. Text and code embeddings by contrastive pre-training[J]. arXiv preprint arXiv:2201.10005, 2022.

[6] Wang Y, Le H, Gotmare A, et al. Codet5+: Open code large language models for code understanding and generation[C]//Proceedings of the 2023 conference on empirical methods in natural language processing. 2023: 1069-1088.

[7] Zhang F, Chen B, Zhang Y, et al. Repocoder: Repository-level code completion through iterative retrieval and generation[C]//Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing. 2023: 2471-2484.

[8] Guo D, Xu C, Duan N, et al. Longcoder: A long-range pre-trained language model for code completion[C]//International Conference on Machine Learning. PMLR, 2023: 12098-12107.

[9] Ma W, Liu S, Lin Z, et al. Lms: Understanding code syntax and semantics for code analysis[J]. arXiv preprint arXiv:2305.12138, 2023.

[10] Mondal D, Lodha A, Sahoo A, et al. Understanding code semantics: An evaluation of transformer models in summarization[C]//GenBench: The first workshop on generalisation (benchmarking) in NLP. 2023: 65.

[11] Yang K, Mao X, Wang S, et al. An extensive study of the structure features in transformer-based code semantic summarization[C]//2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC). IEEE, 2023: 89-100.

[12] Shi C, Cai B, Zhao Y, et al. Coss: Leveraging statement semantics for code summarization[J]. IEEE Transactions on Software Engineering, 2023, 49(6): 3472-3486.

[13] Haque S, Bansal A, McMillan C. Label smoothing improves neural source code summarization[C]//2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC). IEEE, 2023: 101-112.

[14] Xiao Y, Zuo X, Xue L, et al. Empirical study on transformer-based techniques for software engineering[J]. arXiv preprint arXiv:2310.00399, 2023.

[15] Messer M, Brown N C C, Kölling M, et al. Automated grading and feedback tools for programming education: A systematic review[J]. ACM Transactions on Computing Education, 2024, 24(1): 1-43.

[16] Messer M, Brown N C C, Kölling M, et al. Machine learning-based automated grading and feedback tools for programming: A meta-analysis[C]//Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1. 2023: 491-497.

[17] Mishra D S, Edwards S H. The programming exercise markup language: Towards reducing the effort needed to use automated grading tools[C]//Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1. 2023: 395-401.

[18] Mitra J. Studying the impact of auto-graders giving immediate feedback in programming assignments[C]//Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1. 2023: 388-394.

[19] Oli P, Banjade R, Chapagain J, et al. Automated assessment of students' code comprehension using llms[J]. arXiv preprint arXiv:2401.05399, 2023.

[20] Dong D, Liang Y. Grading programming assignments by summarization[C]// Proceedings of the ACM Turing Award Celebration Conference-China 2024. 2024: 53-58.

[21] Cipriano B P, Alves P. Seven Years Later: Lessons Learned in Automated Assessment[C]//5th International Computer Programming Education Conference (ICPEC 2024). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2024: 3: 1-3: 14.