ISSN 1814-4225 (print)
ISSN 2663-2012 (online)

UDC 004.89 doi: 10.32620/reks.2024.4.06

Denys SELIUTIN, Elena YASHYNA

National Aerospace University "Kharkiv Aviation Institute", Kharkiv, Ukraine

INTELLECTUAL CODE ANALYSIS IN AUTOMATION GRADING

*Grades for programming assignments continue to be difficult to assign despite the fact that students have a wide variety of strategies available to address challenges. The primary factor is the existence of several technological frameworks and a range of coding methodologies. The **subject matter** of this article is the process of intelligent evaluation of students' knowledge based on code written by students during regular practical work. The **goal** is to develop an approach for intellectual code analysis that can be easily implemented and integrated into the most widespread grading systems. The **tasks** to be solved include: formalization of code representation for intellectual analysis by applications; analysis of the current state of research and development in the field of automated analysis and evaluation of software codes; introduction of a technique that offers substantial feedback through the integration of intelligent code analysis via code decomposition and providing grading systems an "understanding" of program log. The research **subjects** are methods of the programming code evaluation during distance learning. The **methods** used are: tree classification code analysis and graph traversing methods adopted for the tree linearization goal. The following **results** were obtained: 1. An examination of the current state of automated software code analysis and evaluation reveals that this issue is intricate due to the challenges involved in manually assessing programming projects. These challenges are further exacerbated by the intricate nature of the code, subjective judgment, and the need to adapt to various technical structures. Consequently, there is an urgent demand for automated assessment methods in educational settings. 2. The technique of representing the code structure as syntactic trees was employed to create an automated tool for analyzing software code. This facilitated the decomposition of the code into interrelated logical modules, enabling the analysis of the structure of these modules and the relationships between them. 3. The used methodologies and techniques were used for the analysis of Java code. The syntactic analysis enabled the detection of problematic and erroneous code blocks and the identification of fraudulent attempts (manipulating the program's output instead of performing the algorithm). **Conclusions.** Most current automatic student work evaluation systems rely on testing, which involves comparing the program's inputs and outputs. Unlike the other methods, the approach presented in this study examines the syntactic structure of the program. This enables precise identification of the position and type of mistakes. An astute examination of the gathered data will enable the formulation of precise suggestions for students to enhance their coding skills. The suggested instruments can be incorporated into the Intelligent Tutoring System designed for IT majors.*

Keywords: *data processing; intelligent data analysis; intelligent assessment systems; software code analysis; dynamic analysis of software code; feedback generation.*

1. Introduction

1.1. Background

The incorporation of technology into the ever-changing environment of education has changed how we teach, learn, and evaluate data. The automation of exercise grading technology is a significant advance. As traditional manual grading techniques struggle to meet the needs of modern education, automation provides an attractive alternative that not only answers efficiency problems but also opens up new opportunities for individualized learning and pedagogical innovation.

The traditional method for grading exercises is a time-consuming task for educators. As class numbers grow and online learning becomes more popular, instructors' pressure to provide timely and frequent feedback has reached a new level.

Furthermore, the subjectivity inherent in certain kinds of assessments frequently leads to grading disparities and biases, thus affecting the overall fairness of evaluations. Technology has recently intervened to change this landscape. Automated exercise grading technologies harness the power of artificial intelligence, machine learning, and natural language processing to evaluate student responses with exceptional accuracy and speed. The transition from manual grading to automation has numerous advantages. It not only relieves educators' workloads but also improves students' learning experiences by providing rapid feedback and allowing them to follow their progress in real time.

In addition to the obvious benefits of efficiency, automation opens up new opportunities in the field of education. An educational experience that is both per-

Creative Commons Attribution NonCommercial 4.0 International

sonalized and effective. Educators may now devote more time to meaningful interactions with students, concentrating on improving their conceptual knowledge rather than navigating piles of homework. Furthermore, the ability to handle vast amounts of data enables the detection of learning trends and areas where students frequently struggle. As a result, educators can adjust their teaching approaches to address unique obstacles, resulting in a more effective learning environment.

1.2. Motivation

Automation and grading technology have seen immense growth in the educational landscape. The interaction of crucial factors that address long-standing issues while using new educational opportunities drives the rise and widespread adoption.

Subjectivity, consistency, and scalability issues have long been a problem with traditional grading methods that rely on manual assessment. Educators frequently struggle to maintain consistent grading standards, especially when faced with the onerous task of analyzing a huge volume of assignments. Given these inherent restrictions, there is an urgent need for a more efficient and consistent grading approach.

Automation grading technology is introduced to provide a streamlined and efficient alternative to the time-consuming manual grading procedure. It can quickly grade assignments, quizzes, and tests, thus freeing educators from the time-consuming task of hand-grading. Moreover, it is positioned to satisfy the increasing demands of assessing assignments across more students and more types of distance learning programs without sacrificing quality or rapid review. This frees educators' valuable time and skills for more meaningful educational activities, such as classroom instruction and providing focused, constructive feedback to students.

However, not all educational areas can be easily covered with an automation grading system, especially in practical programming exercises, essays on free topics, and many unstructured assessment tasks related to free-form text processing.

1.3. Problem statement

The manual grading process is not very strict in the matter of giving feedback, especially in learning programming languages, because of the variety of ways each task can be solved, different approaches that can produce the same result, and different technology stacks that can be used under the hood. Moreover, assessment relies on too many human factors that are present during the educational process, like manual verification by the teacher to give advice on the task and usage of different technology stacks that can lead to rejecting overall tasks. This is very crucial to learning a programming language, data analysis, or AI-related courses because students should not be tight to technology stacks or frames setup by the teacher. They should have the ability to reach the goal in their own way or at least get a good explanation of what they need to accept boundaries. These problems mostly relate to human factors and prevent the creation of automated, teacher-like grading technologies. Nevertheless, grading technology systems have become more popular and widespread around the world, which is forcing the community to investigate new and improve existing areas of automation grading methods and techniques.

The problem of grading programming tasks is not new; however, it is partially solved because of the nature of programming code [1] and the variety of ways programming code can solve tasks that prevent automation systems from giving appropriate feedback. This is the reason why the **purpose of this study** is to observe and validate an approach based on abstract syntax for automating the grading of practical programming tasks, along with providing substantive feedback and identifying the most valuable tasks.

1.4. State of the art

Programming exercises are always connected to the process of writing part of the code. It can be a complete program or a part of it; however, even simple iterations over an array of values can be implemented in different ways. Most grading systems assume that students already have some part of the written code and only need to add a missing part to complete the task [2]. This approach is sufficient for learning basic programming language concepts, terms, and syntax. However, what about advanced levels when students need to write some kind of code based on some predefined contract (contract is a set of features along with logic behind of it)? The answer to this question has not yet been determined because this type of verification is very complex and has various limitations.

Ala-Mutka [3] classified the automated evaluation of various qualities into two major groups: static analysis and dynamic analysis, in their 2005 publication, "A Survey of Automated Approaches for Programming Assignments".

Dynamic analysis determines which properties of a running program will remain for one or more executions, which allows it to evaluate these attributes [4]. It often uses a suite of unit tests that compare the printed output or return values of various methods to grade the correctness of a student’s assignment.

The capacity of students to generate efficient code or complete test suites can also be assessed using dynamic analysis [5]. The creation of a thorough test suite

ISSN 1814-4225 (print)
ISSN 2663-2012 (online)

usually requires considerable time and effort from instructors who are experts in designing unit tests using advanced language capabilities like reflection. Students typically have one of three options regarding access to test suites: full access before the final deadline [6], partial access, or no access at all. The assessment approach determines how students are presented with the test suites [7, 8].

Generally, students have full access to all parts of a formative assignment, whereas they may only have access to part of the tests in a summative assignment. To prevent students from hardcoding return values and gaming the system, it is common practice to either use hidden tests or restrict student access to the entire test collection [9].

The static analysis type evaluates software without actually running the program or considering inputs. Software is focused on the detection or partial identification of faults, as well as the verification of maintainability, readability, and the presence of documentation [10]. There are many such tools for each of the programming languages, e.g. JaCoCo, Checkstyle, FindBugs, SonarQube etc.

A comparison of dynamic and static code analysis [11] demonstrates that for optimal software quality, a combination of both strategies is advisable. By employing a mix of these strategies, software engineers can guarantee that their programs possess superior quality and are devoid of flaws. However, this approach will work for experienced developers rather than students who are only learning languages and executing the “hello world” programs.

Meanwhile, from the feedback generation, dynamic code analysis is the most interesting and valuable part of the analysis of segregation because it should determine where students made mistakes. The only thing the grading system needs to do is discover the code block that is causing a problem and give sustainable feedback to a student that should contain the root cause of the problem and instructions on how to fix it.

The most questionable thigs are related to the method or methods of code block identification and further analysis that allows not just verification of “code smells” but also understanding and predicting possible code issues. One such identification method was explained and used by Anh-Tu Phuong Nguyen and Van-Dung Hoang [12]. The present study relies on an abstract syntax tree approach for analyzing a code written in Python to create an abstract layer based on abstract syntax trees that allows a program to walk through the code tree and perform analysis on its own logic on an isolated code base. This approach perfectly fits the goal of source code representation. However, testing code in isolation may not work with other languages due to the inability to isolate and run part of the code in languages like Java or C#.

The usage of machine learning techniques along with artificial intelligence is becoming increasingly popular in all kinds of learning systems: from the tutoring systems to educational robots and grading automation [13]. Based on this, the further use of machine learning for syntax trees may seem very logical. Such an example was provided by Francisco Ortin et al. [14] in his research on how to apply a machine learning technique to analyze Java language code. This study demonstrated the ability of machine learning techniques to identify and evaluate constructs with high precision. However, from a practical perspective, using this approach on different programming languages mean to perform a deep investigation of each language along with involving subject-matter experts to validate the language processing results.

There is no single silver bullet in machine learning that can be used for any programming language. The great analysis work by Peter Hazem et al. [15] proves this statement and forces us to think about the generalization of the approach from the very beginning instead of moving from a concrete language-based implementation to a more generic one.

However, not only syntax trees and machine learning techniques can also be used for code representation. Paiva et al. [16] described alternative approaches that can be used for code analysis, especially feedback generation. Based on this research, we found that syntax trees and code property graphs are the best candidates for the intermediate representation of the source code for further analysis. Moreover, the combination of any of these algorithms with others like data flow graphs or control flow graphs, can be an answer to the question of how a computer system can understand human-written code and compare it with an example one.

Another significant question is a practical way to construct the code representation. This question can be solved using two common approaches: reflection usage that is applicable to popular languages like Java or C#, and object-oriented code analysis. Each of them can be rather correct but slow or fast but not very accurate in a meter of processing inheritance, which takes a major part in learning programming languages. The experimental comparison shows that the object approach works better in the C# language [17].

Nevertheless, any type of tree-based algorithm is a possible performance issue due to the necessity to traverse over the structure more than once along with performing searching and filtering over the node values.

1.5. Objective and Approach

This paper investigates and evaluates an approach based on abstract syntax trees for automating the eval-

uation of code submissions of students. It considers not only the accuracy (that is a goal of the static code analysis) of the output but also the correctness and conformity in a form of evaluating the logic behind the code. This leads to formalization of the primary goal, namely, ensuring grading consistency by intellectual analysis in the form of a comparison of code terms and code branches with predefined by tutor solutions, along with finding the biases and explaining the mistakes in a logic flow and how to correct them.

The main objectives of this research are as follows:

- describing the problem and formalization of code representation for intellectual analysis by applications;
- adapting a method and developing an algorithm that offers invalid code block identification via code decomposition and provides grading systems an "understanding" of program execution;

exploring algorithm behavior in common educational cases;

- discussion of results and recommendations;
- summarizing the results and describing future research areas.

To achieve these aims, the article was divided into five sections to form the article structure:

- Section 1 – "Introduction" explains the background and motivation of this article, along with analyzing existing literature to better and accurately demonstrate the intention of the current work. In addition, this section presents current trends in the research field, algorithms, practical implementations, and alternative approaches that may solve the problem in the case of future research.
- The methodology of providing feedback during the practical task assessment along with the use of the syntax tree approach is described in Section 2. This provides an understanding of the algorithm developed in the scope of this paper.
- Section 3 – "Implementation and evaluation of code samples" explains, in an example, how the algorithm works in common use cases during programming language learning.
- The "Discussion and recommendations" section shows the benefits and concerns of this algorithm usage along with guiding key components that need to be taken into account during algorithm implementation and integration into existing distance learning or grading systems.
- The paper ends with a conclusion section that summarizes the content provided in the paper and highlights future research areas.

2. Methodology

The process of giving feedback is complicated and should consider already written code, be able to analyze errors, and make assumptions about the logical program component. This process breaks it down into several stages: determining which part of code is wrong (this stage supposed to include **splitting code** into logical blocks and further processing of the code structure), code analysis to find the exact cause (this is the stage where deviation or **invalid block identification** should be performed based on the tree code structure), and generating advice on what needs to be done to make it work (this is a stage of **advice preparation** based on invalid code found on the previous stage) (Fig. 1).

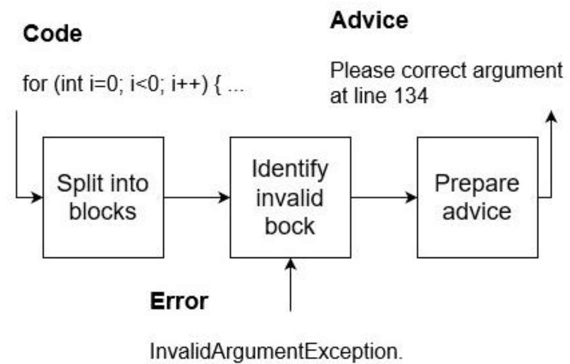

Fig. 1. General approach to advice generation

However, the advice generation algorithm is more complex and comprises different stages (Fig. 2).

The **code block identification stage** (Fig. 2) is almost easiest. This process comprises parsing, function extraction, and obfuscation stages. The goal is to walk through the code and split it by programming language expressions such as variables, simple entities, and control structures [18]. In addition, it assumes the creation of a dictionary of functions used in the code, e.g., the creation of pairs for method names and method bodies in programming languages like Java or C#. Another method of identification that will be helpful in cases of large amounts of code is the use of fingerprints [19] and text preprocessing like obfuscation, which replaces variables and makes code more recognizable for fingerprints. The results of such parsing steps are simply a plain code block structure and a map of pointers to code blocks.

Further code analysis, invalid block identification, and advice preparation are the most difficult steps (Fig. 2). By the line of code where an error occurs, we can identify the block statement it belongs to. However, this does not mean that blocks are identified correctly because there may be a chain of errors that lead to errors. This means that block detection is effective for simple pieces of code but may not be effective for tasks where students should implement contracts on their own. This is a case in which we must have some background information to identify the problem correctly.

ISSN 1814-4225 (print)
ISSN 2663-2012 (online)

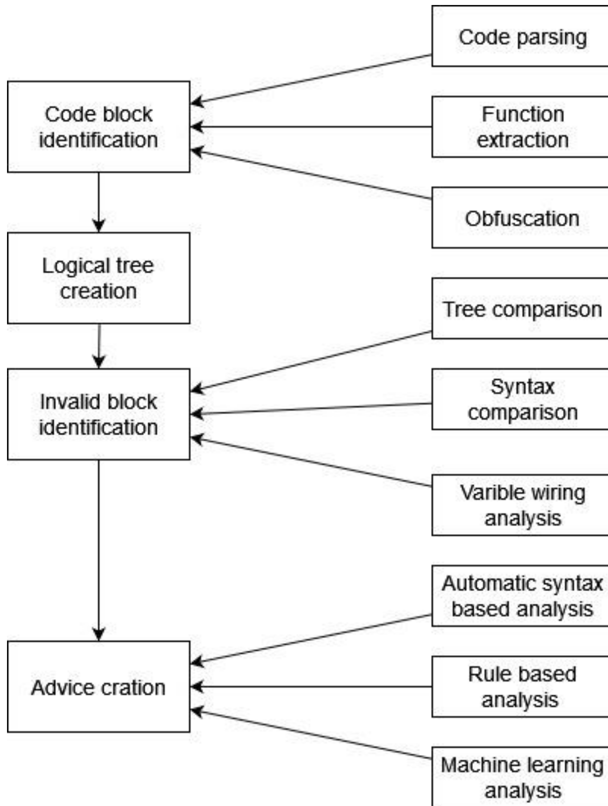

Fig. 2. General flow of a grading process in automated systems

Background in a code analysis is an understanding of the code logic that can be obtained by analyzing code blocks and their interconnections. Here, identified code blocks must be gathered into a **logical representation** (logical tree creation stage on Fig. 2 for further processing.

The best choice for gathering a logical code structure is to combine blocks into a tree because this allows us to create a full picture of the operations performed under the hood.

A tree (the program representation) in this case is a finite set of nodes with a specially designated parent node called the root (entry point to the program). The remaining nodes are based on blocks and logical conditions inside blocks when each block is a node in the tree and partitioned into *d* disjoint sets **{R1; R2;:::;Rd}** such that each of these sets is a tree. Each node may have other nodes inside unless its' value is a terminal operation - operation that does not have code blocks inside. E.g., the **if-else** statement will be presented as a node with two child nodes; while **i++** will be positioned as a leaf - node without children.

The main benefit of tree structure is the possibility to perform a comparison based on subtrees **{R1;R2;:::;Rd}** for each of the nodes and identify the inconsistency between two trees in a way of corrupted nodes identification (1). The comparison is a content equality operation on the values of two nodes (2).

$$T1(n) \subseteq T2(m), \tag{1}$$

where *n* and *m* denote the positions of nodes in trees T1 and T2, respectively.

Ideally, n and m should be the same number or at least have the same degree. Therefore, each node should be compared with an appropriate node from a tree of valid solutions. This comparison (P) should consider expressions in the node under comparison and expressions in the child nodes (2).

$$P_{nodei\ T1} = \begin{cases} 1, \text{ if } E_{nodei\ T1} = E_{nodei\ T2} \\ 1, \text{ if } E_{nodei\ T1} \subseteq \underset{i=0..k}{\forall} E_{nodei\ T2} \\ 0, \text{ otherwise} \end{cases}, \tag{2}$$

where $P_{nodeiT1}$– result of the comparison of code statement $E_{nodeiT1}$ in $node_i$ of the student code tree T1 and code statement $E_{nodeiT2}$ in $node_i$ of the expected code tree T2, i – represents a logical segment in a code tree rather than a simple line of code.

Thus, if an expression in a target node has at least one similar expression in any node to form a valid solution, we can identify such a node as a matched node and identify it as a mismatched node otherwise.

The summation of the equality operations of each (3) node will give some correct nodes and returns the expected results. This number can be used to obtain a general understanding of the logical validity of the overall code.

$$P_{T1} = \sum P_{nodeiT1}, \tag{3}$$

where P_{T1} – aggregation of the equality of each code statement $P_{nodeiT1}$ in the students' code tree T1.

The next step is to identify **the problem** or **invalid block identification** (Fig. 2). In this step, two types of misleads must be considered

The first is the simplest one, where an error occurs, and the error code and error line are already known. In this case, trees should be used to identify problematic code structures and advice generation. This situation does not require explanation because it is common in any integrated development environment and can be represented via hints or stack traces.

The second type of error occurs when there is no error; however, the expected result is inappropriate. In this case, the teacher or data gathered from prior successful code executions should provide an "ideal" implementation for each task. This concept and implementation are used to compare tree nodes and find deviations. The code tree comparison algorithms should consider the variations in the code structures and different approaches that can be used. It should find the differences in the logical code composition of the input data

for advice generation. This leads to the comparison of code lines, variables, statements, methods, and even classes in a way of performing syntax and variable wiring comparisons.

The idea of variable wiring comparison is to identify that all variables, method usage, and return values are wired together. This understanding is required to identify cases in which a student attempts to cheat by returning constant values or is using inappropriate methods. In contrast, syntax comparison targets invalid method usage identification rather than verification of the compilation to provide an understanding of whether correct libraries are used or not.

Advice generation is based on logical differences, syntax, and wiring analysis and can transform them into sustainable feedback. In this part of the process, machine learning techniques can be used along with rule-based mechanisms.

3. Implementation and evaluation of code samples

3.1. Code parsing and metadata pre-processing

Let's assume that we have simple Java code with several methods and simple calculations inside (Fig. 3).

The provided code is parsed into a code block structure with main operations that is easy for the machine to analyze.

line 2 - variable - **private int a;**
line 3 - variable - **private int b;**
lines 5-7 - method - **public int sum();**
line 6 - **return** from method;
line 6 - math operation - **a+b**;
line 9-11 - method - **multiplySum(int x);**
line 10 - **return** from method;
line 10 - **method execution;**
line 10 - math operation - ***x**;
lines 13-16 - **constructor**;
line 14 - variable set - **a = 3;**
line 15 - variable set - **b = 5;**
lines 18-26 - **entry point**;
line 21 - instance creation - **new Main();**
line 22 - method execution - **m.sum();**
line 22 - variable set - **c = m.sum());**
line 23 - if statement - **if(c > 3)**
line 24-26 - **output** - console;
line 25 - method execution - **m.multilySum(3);**
line 27 **else** statement
line 28-30 - **output** - console;
line 29- method execution - **m.multilySum(4);**

```
1  public class Main{
2      private int a;
3      private int b;
4
5      public int sum(){
6          return a + b;
7      }
8
9      public int multiplySum(int x){
10         return sum() * x;
11     }
12
13     public Main() {
14         this.a = 3;
15         this.b = 5;
16     }
17
18     public static void main(
19         String[] args
20     ){
21         Main m = new Main();
22         int c = m.sum();
23         if(c > 3) {
24             System.out.println(
25                 m.multiplySum(3)
26             );
27         } else {
28             System.out.println(
29                 m.multiplySum(4)
30             );
31         }
32     }
```

Fig. 3. Correct coding assignment execution

During the parsing process various kinds of metadata gathered: scope of variable visibility (relationship between variable declaration and places where they can be used), method to line relationship and method hierarchical usage (Fig. 4).

The metadata algorithm can map variables and methods together by finding places where these variables are used (Fig. 5). Such information is a source data for the variable wiring analysis, which is a simple checking of next statements:

- all objects have initialized variables (verification that class variables are present in the constructor or in appropriate set method and that all the set method are executed in the entry point or in other methods under the entry point execution);

- there are no missed or undeclared variables in the code;

- all the variables are of the defined, existed and valid types.

Fig. 5 gives a hierarchical representation of the variable usage at the entry point. In the case of several entry points (e.g., in case of several endpoints that are available over the network), we obtain several hierarchical representations for each entry point with its own variables inside.

ISSN 1814-4225 (print)
ISSN 2663-2012 (online)

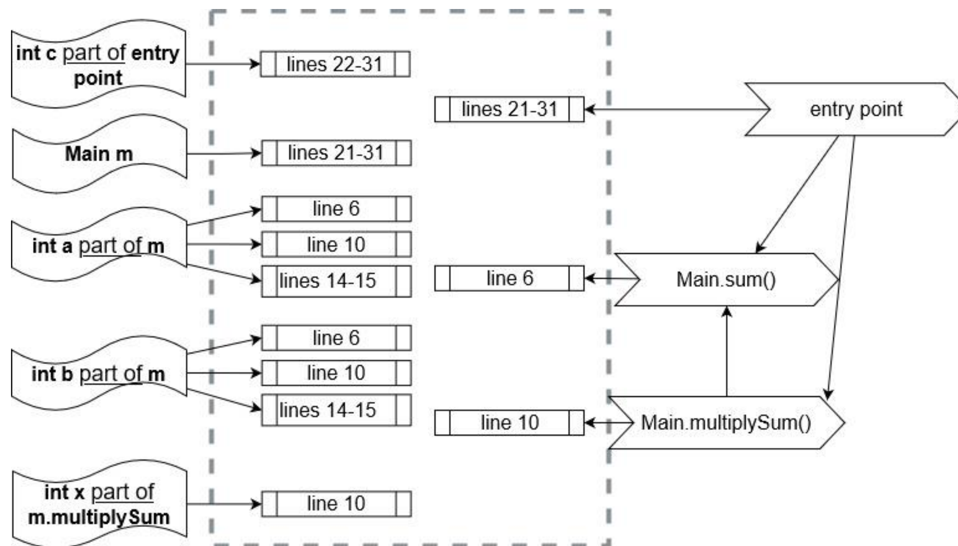

Fig. 4. Metadata for the entry point

This representation can be used for further fingerprint creation to simplify searches over the logical structure. A simple statement-based fingerprint can be used in this study [12] with minor modifications. For example, we consider fingerprints in the **miltiplySum** method (Fig. 6). This method calls **the sum** method inside and performs a mathematical operation. The **sum** method performs a mathematical operation on the two integer values inside.

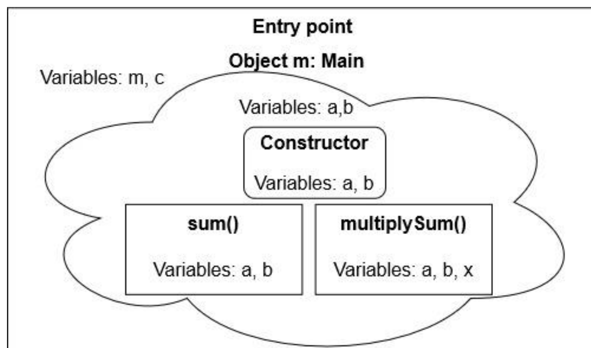

Fig. 5. Hierarchical representation of variables inside an entry point

Thus, if we remove variable names, replace concert types with more generic ones (like replace int with number), show the hierarchy as a part of fingerprint, and then cut the number of method execution (linearize the execution tree). We will obtain the fingerprint, which shows what operations are actually performed for this method. Such an approach is very interesting in terms of practical experience; however, it is not a goal of this article.

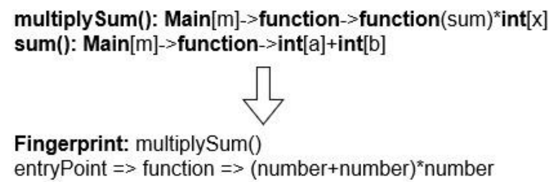

Fig. 6. Fingerprint for multiplySum method

3.2. Syntax tree composition

In the next step, after metadata preparation, the code block structure is transformed into a tree that shows the logical code structure from the very beginning of the program to the entry point to the very end and possible variants of the program end (Fig. 7).

The structure starts from the entry point that is represented via "*public static void main*" construct (construct - is the line of code that may contain method executions, variable declaration, mathematical operation and any syntax sugar that is available in the programming language). This construct does not have any subtrees inside.

The next node (**line 21**) in the tree is the instance creation of class **Main** along with variable **m** declaration. This is a complex construct because instance creation in our case should include a variable setup at the instance level, which is transformed into an inner tree for this node. The inner tree comprises two node value setups for instance or class variables **a** and **b.** The same statement applies to the next node at line **22** because it is also complex due to variable **c** assignment and method execution on previously created local variable **m.** Sub tree of this node represents the execution of the method

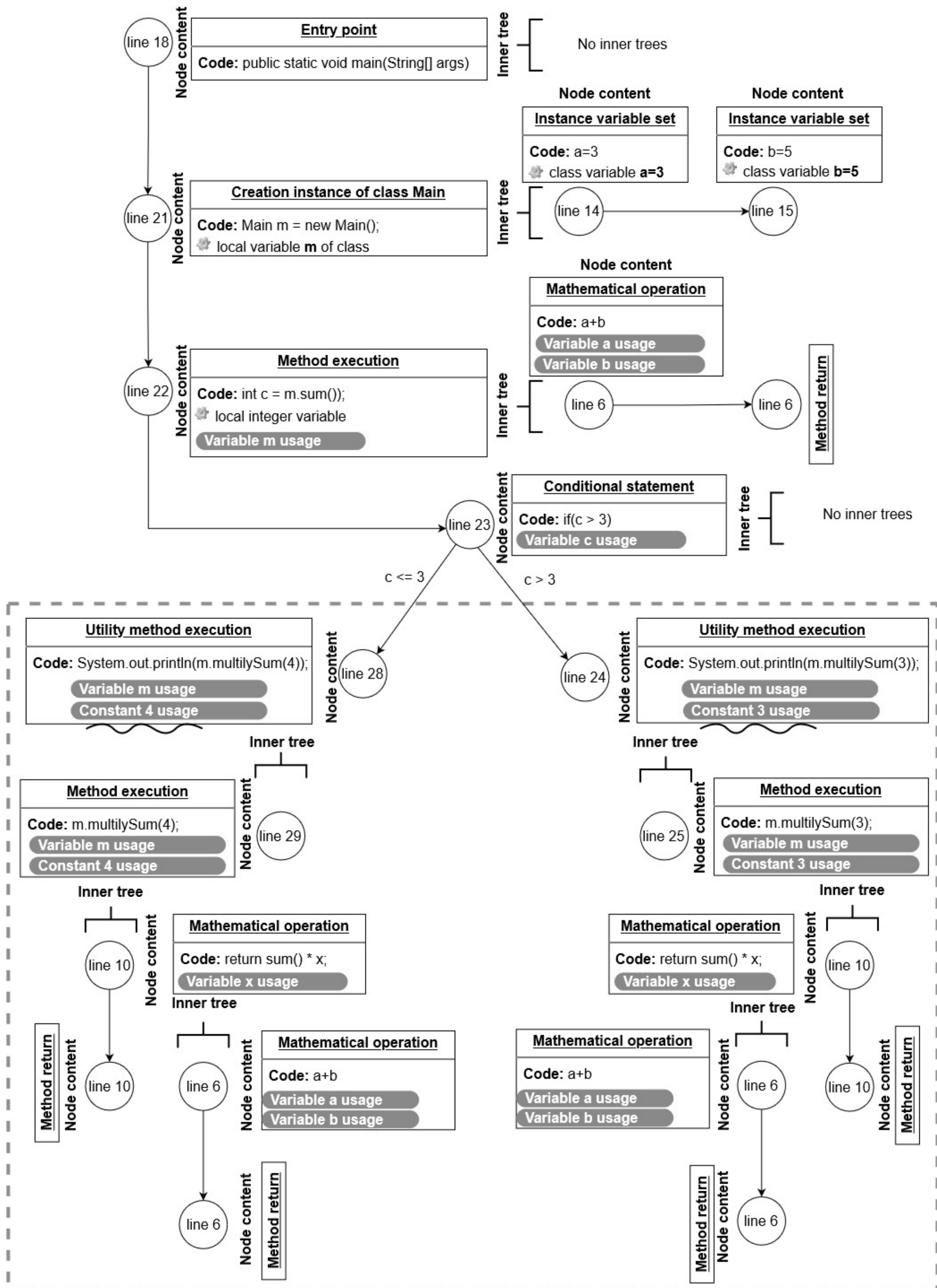

Fig. 7. Logical tree structure for correct coding assignment execution

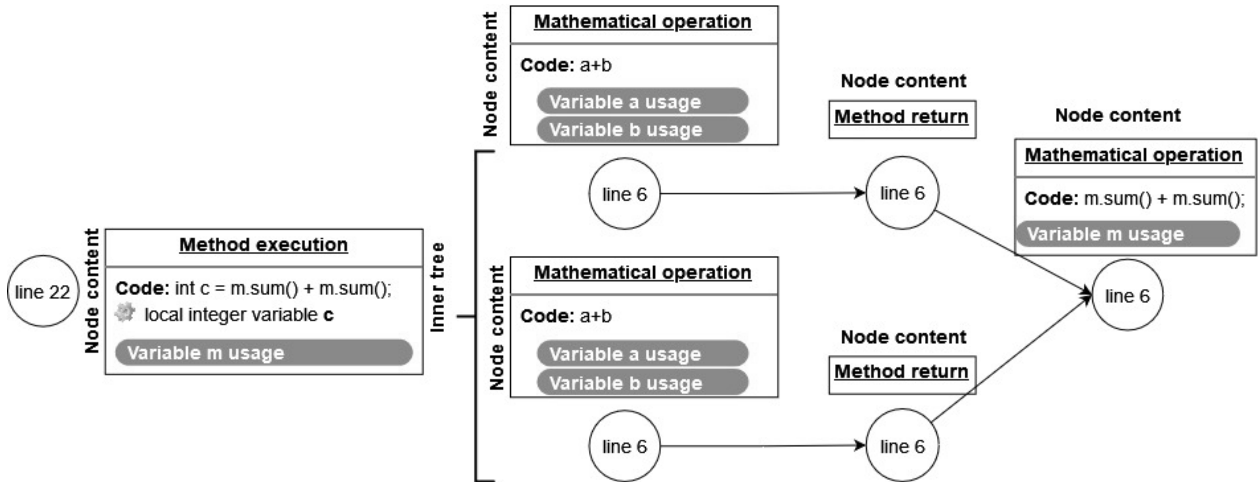

Fig. 8. Several method executions in one construct

sum on variable **m**, which is an instance of class **Main**. In the case of several method execution subtrees, all method execution trees are combined (Fig. 8).

Line 23 represents the conditional statement. Such nodes may split logical tree into more than two branches, due to possibility of programming languages to combine several **if-else** statements together (Fig. 9) or even having special statements with more than two possible solutions like **switch-case** (Fig. 10). In our example, the logical tree was split into two branches: a branch starting from node on **line 24** and a branch starting from **line 28**. They are almost identical in our example because we use the same utility method and code. The only difference – is the constant value that is used inside statements (block marked in dashes and values highlighted via wave lines on Fig. 7).

To minimize and simplify the overall representation of the logical tree structure in Fig. 7 and for better visualization, we can hide the additional data like value usages, remove return statement nodes, and replace complex construct nodes with simplified representation (Fig. 11).

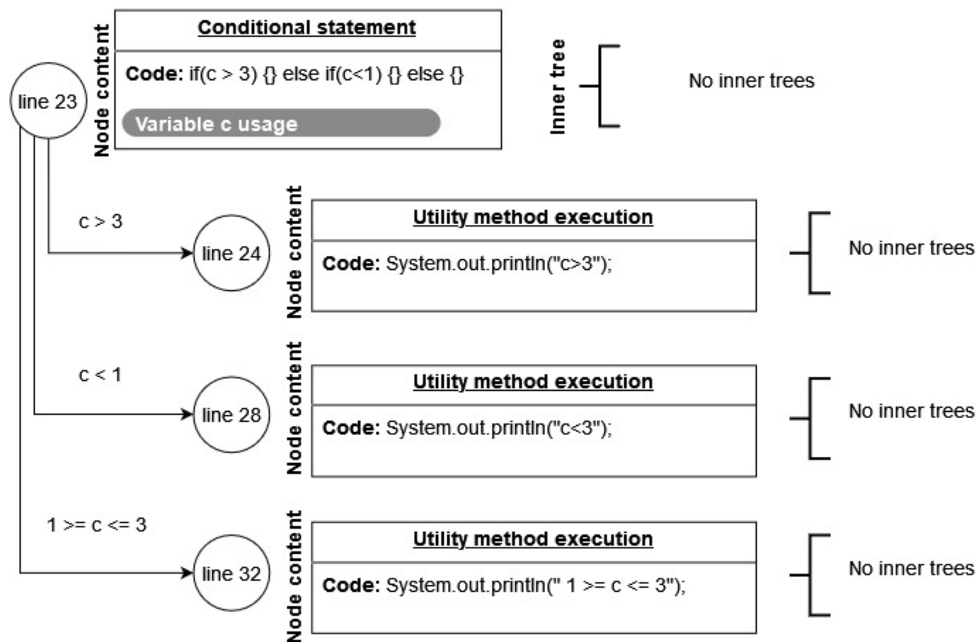

Fig. 9. Multiple outputs for several if-else statements

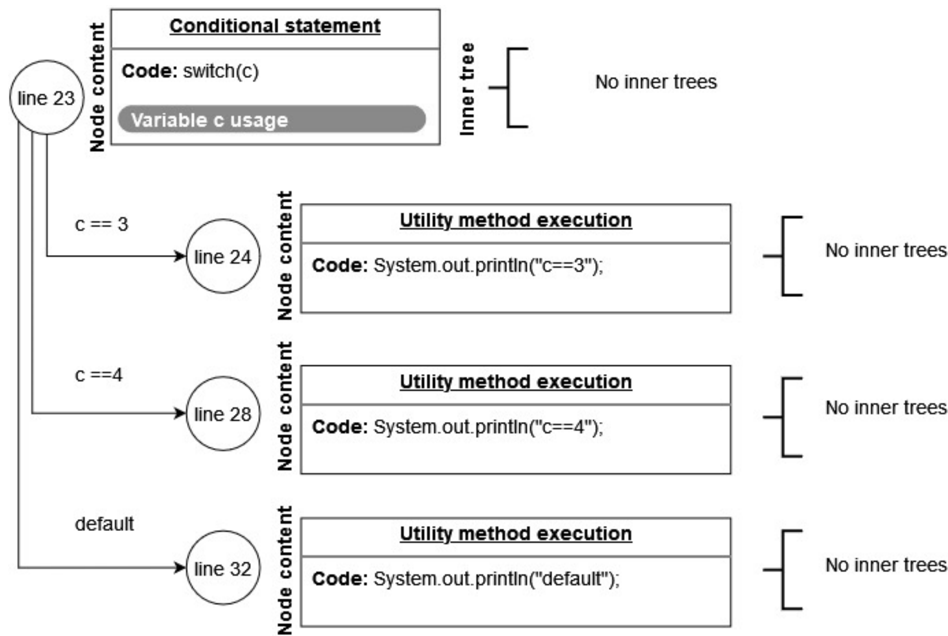

Fig. 10. Multiple outputs for several if-else statements

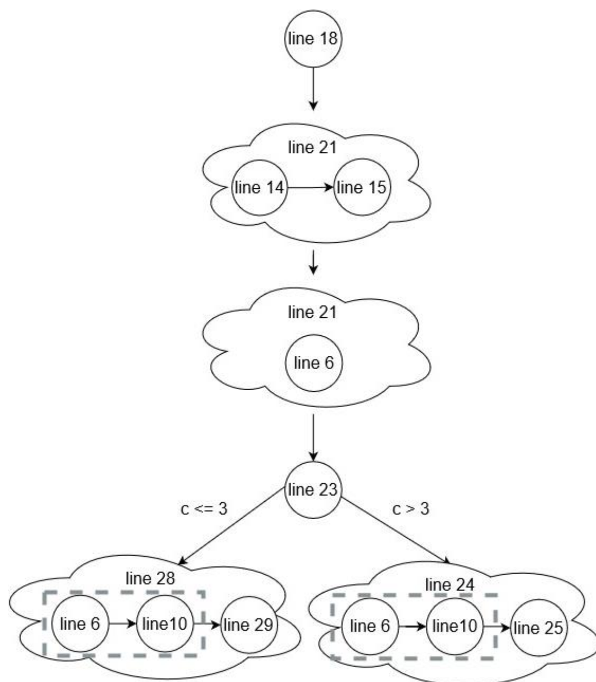

Fig. 11. Simplified representation of the logical tree structure

In Fig. 11, we have only the nodes that need to be considered during comparison (for the best understanding only line numbers are present on the simplified view of Fig. 7). Moreover, additional filtering of the tree allows us to remove codes that do not affect the overall result (marked in dash squares). Nevertheless, such removal is not always possible because: from the one hand, it allows to minimize the number of nodes during further comparison in case of removing code in leaves (like in our example); from the other hand, such removal may lead to skipping a large part of the code that will not be assessed at all but should be. From this standpoint, duplicate removal is logical for program code understanding but not applicable to our goal-assessing programming language leaning tasks.

3.3. Syntax tree analysis and advice generation

The next algorithm step is tree analysis. For our example, we combine this step with the next one–advice generation, because this approach provides a full understanding of the advice creation mechanism based on a logical code tree.

To demonstrate how it should work, we assume that we have four different cases with this structure – one per different student. In three cases, we simulate use cases when the code executes correctly but data in a console are completely different from the expected value – 24 and in the fourth one - with an exception during execution.

3.3.1. Invalid constant value usage

The problem occurs at **line 25**, where an invalid number is used by the student for the constant (used value is value 5 instead of 3). As a result, the console output will be 40 instead of 24 (Fig. 12).

Per the algorithm, this change will be detected via tree comparison, and advice will point at the wrong constant value at **line 25** (Fig. 13) and explain how this val-

ISSN 1814-4225 (print)
ISSN 2663-2012 (online)

ue will break the expected result (such an explanation should be given based on tree with a help of traversing on it to the very end and taking into account all direct and indirect usages of constant value that is different).

Thus, advice itself will be a line of code where the problem occurs and used value along with showing how value 5 affects the result (just inlining the further tree into text with variables, similar to "(3 + 5) * 5"). It will show students the place and explains how it affects the overall program execution.

```
18      public static void main(
19          String[] args
20      ){
21          Main m = new Main();
22          int c = m.sum();
23          if(c > 3) {
24              System.out.println(
25                  m.multiplySum(5)
26              );
27          } else {
28              System.out.println(
29                  m.multiplySum(4)
30              );
31          }
32      }
```

Fig. 12. Mistake in value usage

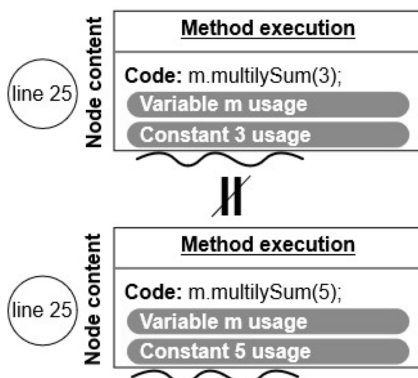

Fig. 13. Comparison of two nodes at the line where mistake occurs

3.3.2. Invalid comparison operator usage

In this case, the comparison operation is invalid in the **if-else** statement (Fig. 14).

In the algorithm, the tree comparison will handle such deviations by comparing child nodes and determining that nodes are not in the correct order (Fig. 15). This leads to comparing the constructs under the node at **line 23**. This comparison gives an understanding that the equality sign in the construct is invalid, or variable **c** and constant **3** are in the wrong places. As a result, there are two pieces of advice: correcting the sign and swapping constant **3** and variable **c** in the code.

```
18      public static void main(
19          String[] args
20      ){
21          Main m = new Main();
22          int c = m.sum();
23          if(c < 3) {
24              System.out.println(
25                  m.multiplySum(3)
26              );
27          } else {
28              System.out.println(
29                  m.multiplySum(4)
30              );
31          }
32      }
```

Fig. 14. Mistake in equality sign

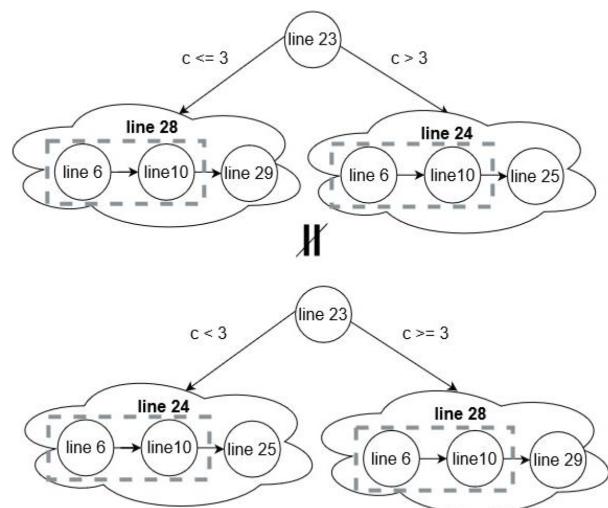

Fig. 15. Invalid child node order

3.3.3. Hardcoding of expected result value

However, what if a student just hardcodes the output value, like in **line 25** (Fig. 16)?

```
18      public static void main(
19          String[] args
20      ){
21          Main m = new Main();
22          int c = m.sum();
23          if(c > 3) {
24              System.out.println(
25                  24
26              );
27          } else {
28              System.out.println(
29                  m.multiplySum(4)
30              );
31          }
32      }
```

Fig. 16. Cheating attempt

The tree comparison also will be able to handle such a case because there will be missing nodes (Fig. 17). However, syntax and wiring analysis are also able to identify the same problem even faster due to comparing only a code syntax without working with a tree structure. For example, constant usage at **line 24** can be simply identified by the static analysis, whereas wiring analysis shows no problems. Such a case with wiring analysis is possible due to **line 29**, where all the methods used and variables are under the hood. However, if at **line 29**, students put the constant too, wiring analysis will complain about unused methods and variables. This is why wiring and static analysis should be used as complementary methods and should only be used to clarify the problem.

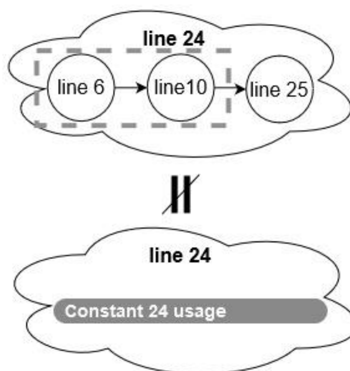

Fig. 17. Node difference in case of cheating attempt

Nevertheless, as a result, advice not to hardcode expected results will be given to the student along with a warning that such an attempt is a use-case of inappropriate solution. This information can also help tutors better understand how many students are trying to cheat instead of learning the topics.

3.3.2. Missing variable initialization

In the case of an error, the algorithm walks through the tree and determines whether all variable assignments are in place. In the example below (Fig. 18), variables **a** and **b** are not set, which leads to the error.

```
1  public class Main{
2      private Integer a;
3      private Integer b;
4
5      public int sum(){
6          return a + b;
7      }
8
9      public int multiplySum(int x){
10         return sum() * x;
11     }
12
13     public Main() {}
```

Fig. 18. No variable initialization

During the node comparison, we observed a difference in node structures (Fig. 19), where the algorithm at line 21 did not find any subtrees. However, his case can be simply covered by static code analysis or variable wiring, where the instance variable is identified as unknown due to the absence of variable set up code. Such an approach will work for simple cases but not for complex ones, where assignments are dynamic and may need to be investigated in-depth, which can take a significant amount of time and computing resources. This leads to proving the statement from the previous case that wiring and static analysis should be complimentary methods and applied to clarify cases with finite numbers of nodes.

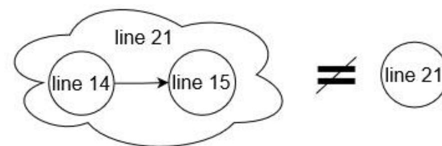

Fig. 19. Node comparison when no variable initialization is present

The algorithm provides a list of variables that were not initialized for the instance of class Main. Such advice points students on common issues during learning object-oriented programing – variable initialization and the usage of classes instead of primitive types.

The above cases involve common issues while studying programming languages. Such issues can be connected to coding, understanding coding paradigms and principles, or even attempts at cheating. The goal of the proposed algorithm is to identify them and prepare data for advice generation. On the other hand, advice generation principles and mechanics are complicated things that are not fully a part of this article due to the necessity to showcase how they work in the educational process. Nevertheless, building the logical code tree and searching it along with supplementary code analysis methods provide full information of the code written by the student to identify the issue and find a way of fixing it in a code written by the student rather than advising them to rewrite the whole code from scratch.

4. Discussion and recommendations

According to the algorithm examples, the process of invalid block identification as a key point of the grading process is based on comparing the code terms and branches in the abstract syntax tree of the source code, and handles common educational cases like:

- mistypes and cheating attempts - by validating a code constructs and variable values during term analysis;

ISSN 1814-4225 (print)
ISSN 2663-2012 (online)

- runtime errors are identified during the variable validation;

- common mistakes in operation usage are also identified by validating code constructs and variable values.

The proposed method can be used to identify not only syntactical errors but also semantic and unintentional errors. This capability enhances the grading process because it allows the system to understand the intended logic and flow of the code, rather than just its surface-level correctness.

Nevertheless, there are obstacles linked to the use of the proposed algorithm due to the use of abstract syntax trees for code evaluation. Obstacles arise from the computational difficulty associated with producing and analyzing abstract syntax trees, which may require significant processing resources. In addition, an algorithm is capable of processing several programming languages, where individual language features may necessitate customized parsing and analysis techniques, thus increasing complexity in the implementation process.

Considering the advantages and obstacles described, it is advisable to incorporate analysis into code grading systems after performing further experiments in which quantitative metrics can be used to evaluate effectiveness. Further implementation should be carried out considering the following factors:

- enhance the efficiency of tree generation and analysis to maintain the grading system's performance, especially when dealing with extensive codebases;

- create language-specific extensions to handle distinct constructions and idioms, guaranteeing the accuracy of analysis across various programming languages;

- regularly updating the system to incorporate new language features and paradigms, ensuring that the system remains up-to-date and accurate;

- provide a unified analysis approach that can be easily adopted to different programming languages and does not depend on them.

5. Conclusion

Manual evaluation of programming projects challenging due to the inherent intricacy of coding and the vast array of possible solutions. In addition to the complexity of the grading process, subjective evaluation and the incorporation of diverse technical frameworks are also considered using intermediate representations of the programming code. Despite the limitations, there is a growing need to develop and enhance automated grading systems in educational settings.

This study proposes the use of dynamic code analysis in a way of syntax trees usage to improve an automated grading system that can not only assess coding tasks but provides advice on how to fix existing students' code instead of rewriting it based on predefined examples. As we can see from the examples in the article, the most common codding mistakes are perfectly covered by the algorithm that can potentially reduce the amount of tutor involvement in the education process when the less trivial ones should be discovered during the experiment. This technique attempts to solve common issues in programming training by providing students with valuable assistance in identifying flaws, analyzing code structure, and comparing syntax. In addition, it attempts to identify the logic behind the code. However, although this technique demonstrates potential, it also acknowledges the complexity of coding locations, particularly at more advanced levels. To automate the grading and feedback generation process successfully, this technique may require additional customization and adaptation.

Future research. To fully realize its potential, it may be necessary to make additional adjustments and adaptations for the effective automation of grading operations and feedback production along with performing experiments to understand the time consumptions that instructors dedicate to grading with a help by using an automated solution instead of manual grading work. By integrating this technique into educational systems like Moodle, it is feasible to significantly speed up evaluation. The further research phases in this domain encompass the implementation and testing of algorithms, as well as their subsequent incorporation into educational frameworks.

Furthermore, the application of artificial intelligence techniques, such as recurrent neural networks and massive language models, for code analysis appears to be highly promising. Although parsing is more reliable and less prone to incorrect responses, these tools can be significantly more efficient than conventional methods. These techniques can facilitate the detection of more complex defect categories than traditional code analysis methods.

Contribution of authors: conceptualization, methodology – **Denys Seliutin**, **Olena Yashyna;** writing and original draft preparation – **Denys Seliutin**; review – **Olena Yashyna**.

Conflict of Interest

The authors declare that they have no conflict of interest in relation to this research, whether financial, personal, authorship or otherwise, that could affect the research and its results presented in this paper.

Financing

This study was conducted without financial support.

Data Availability

The work has no associated data.

Use of Artificial Intelligence

The authors confirm that they did not use artificial intelligence methods in their work.

All the authors have read and agreed to the publication of the finale version of this manuscript.

References

1. Conejo, R., Barros, B. & Bertoa, M. F. Automated assessment of complex programming tasks using SIETTE. *IEEE Transactions on Learning Technologies*, 2019, vol. 12, no. 4, pp. 470–484. DOI: 10.1109/tlt.2018.2876249.

2. Bertagnon, A., & Gavanelli, M. MAESTRO: a semi-autoMAted Evaluation SysTem for pROgramming assignments. *Proceeding of the 2020 international conference on computational science and computational intelligence (CSCI),* Las Vegas, NV, USA, IEEE, 2020, pp. 953-958. DOI: 10.1109/csci51800.2020.00177.

3. Ala-Mutka, K. M. A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 2005, vol. 15, iss. 2, pp. 83–102. DOI: 10.1080/08993400500150747.

4. Ball, T. The concept of dynamic analysis. *ACM SIGSOFT Software Engineering Notes*, 1999, vol. 24, iss. 6, pp. 216–234. DOI: 10.1145/318774.318944.

5. Coore, D., & Fokum, D. Facilitating course assessment with a competitive programming platform. *Proceeding of the SIGCSE '19: the 50th ACM technical symposium on computer science education*, New York, NY, USA, Association for Computing Machinery, 2019, pp. 449-455. DOI: 10.1145/3287324.3287511.

6. Ayewah, N., Pugh, W., Hovemeyer, D., Morgenthaler, J. D., & Penix, J. Using static analysis to find bugs. *IEEE Software*, vol. 25, no. 5, pp. 22–29. DOI: 10.1109/ms.2008.130.

7. Restrepo-Calle, F., Ramirez-Echeverry, J. & González, F. Using an interactive software tool for the formative and summative evaluation in a computer programming course: an experience report. *Global Journal of Engineering Education*, 2020, vol. 22, no. 3, pp. 174–185. Available at: https://www.researchgate.net/publication/346004432_Using_an_interactive_software_tool_for_the_formative_and_summative_evaluation_in_a_computer_programming_course_an_experience_report (accessed 09 June 2024).

8. Le, D. M. Model- based automatic grading of object- oriented programming assignments. *Computer Applications in Engineering Education*, 2021, vol. 30, iss. 2, pp. 435–457. DOI: 10.1002/cae.22464.

9. Liénardy, S., Leduc, L., Verpoorten, D., & Donnet, B. Café': Automatic Correction and Feedback of Programming Challenges for a CS1 Course. *Proceeding of the ACE'20: twenty-second australasian computing education conference*, New York, NY, USA, Association for Computing Machinery, 2020, pp. 95–104. DOI: 10.1145/3373165.3373176.

10. Ahire, P., & Abraham, J. Perceive core logical blocks of a C program automatically for source code transformations. *Proceeding of the 18-th Intelligent Systems Design and Applications conference,* Springer, Cham, 2019, pp. 386–400. DOI: 10.1007/978-3-030-16657-1_36.

11. De Silva, D., Samarasekara, P., & Hettiarachchi, R. *TechRxiv*. A comparative analysis of static and dynamic code analysis techniques. 2023. DOI: 10.36227/techrxiv.22810664.v1. (unpublished).

12. Narayanan, S., & Simi, S. Source code plagiarism detection and performance analysis using fingerprint based distance measure method. *Proceeding of the 2012 7th international conference on computer science & education (ICCSE 2012)*, Melbourne, VIC, Australia, 2012, pp. 1065–1068. DOI: 10.1109/iccse.2012.6295247.

13. Xu, W., & Ouyang, F. The application of AI technologies in STEM education: a systematic review from 2011 to 2021. *International Journal of STEM Education*, 2022, vol. 9, article no. 59. DOI: 10.1186/s40594-022-00377-5.

14. Barros, J. P. Assessment for computer programming courses: a short guide for the undecided teacher. *Proceeding of the 14th international conference on computer supported education*, Online Streaming, SciTePress, 2022, pp. 549–554. DOI: 10.5220/0011095800003182.

15. Samoaa, H. P., Bayram, F., Salza, P., & Leitner, P. A systematic mapping study of source code representation for deep learning in software engineering. *IET Software*, 2022, vol. 16, iss. 4, pp. 351–385. DOI: 10.1049/sfw2.12064.

16. Paiva, J., Leal, J., & Figueira, Á. Comparing semantic graph representations of source code: the case of automatic feedback on programming assignments. *Computer Science and Information Systems*, 2024, vol. 21, no. 1, pp. 117–142. DOI: 10.2298/csis230615004p.

17. Wojszczyk, R., Hapka, A., & Królikowski, T. Performance analysis of extracting object structure from source code. *Procedia Computer Science*, 2023, vol. 225, pp. 4065–4073. DOI: 10.1016/j.procs.2023.10.402.

18. Nguyen, A. T., & Hoang, V. D. Development of code evaluation system based on abstract syntax tree. *Journal of Technical Education Science*, 2024, vol. 19, no. 1, pp. 15–24. DOI: 10.54644/jte.2024.1514.

19. Ortin, F., Facundo, G., & Garcia, M. Analyzing syntactic constructs of Java programs with machine learning. *Expert Systems With Applications*, 2023, vol. 215, iss. C. DOI: 10.1016/j.eswa.2022.119398.

ISSN 1814-4225 (print)
ISSN 2663-2012 (online)

Received 17.07.2023, Accepted 18.11.2024

ІНТЕЛЕКТУАЛЬНИЙ АНАЛІЗ КОДУ В СИСТЕМАХ АВТОМАТИЗОВАНОГО ОЦІНЮВАННЯ

Д. А. Селютін, О. С. Яшина

Оцінювання завдань із програмування залишається проблемою, навіть незважаючи на різноманітність підходів, які студенти можуть використовувати для вирішення труднощів. Основною причиною є наявність численних технологічних стеків, що реалізуються, і різноманітність підходів до написання коду, які можна використовувати. **Предметом** вивчення даної статті є процес оцінювання знань студентів на основі коду, який був написаний студентом під час звичайної практичної роботи. **Мета** полягає в розробці підходу до інтелектуального аналізу коду, який можна легко реалізувати та інтегрувати в найпоширеніші системи автоматизованого оцінювання. **Завданнями**, які потрібно вирішити, є: формалізація подання коду для інтелектуального аналізу програмними засобами; аналіз сучасного стану досліджень та розробок в галузі автоматизованого аналізу та оцінювання програмного коду; розробка методу та алгоритму, які пропонують суттєвий зворотній зв'язок через інтеграцію інтелектуального аналізу методом декомпозиції та надання системам оцінювання «розуміння» журналу виконання програми у вигляді аналізу помилкових блоків. **Предметом** цього дослідження є методи оцінки програмного коду під час дистанційного навчання. Використовувані **методи**: методи аналізу коду на базі алгоритмів класифікації та представлення коду у вигляді дерева разом із його вирівнюванням. Були отримані наступні **результати**: 1. Проведено аналіз сучасного стану в галузі автоматизованого аналізу та оцінювання програмного коду показав, що ця проблема є складною бо труднощі, пов'язані з оцінюванням проектів програмування вручну, ще більше ускладнюються складною природою коду, суб'єктивним судженням і вимогою адаптації до різних технічних структур, що лише підкреслює нагальну потребу в автоматизованих методах оцінювання в освітніх середовищах. 2. Для розробки методу автоматизованого аналізу програмного коду було застосовано моделювання структури коду у вигляді синтаксичних дерев. Це дозволяє розбити код на взаємопов'язані логічні блоки, аналізувати структуру блоків та зв'язки між ними. 3. Розроблені методи та алгоритми застосовані для аналізу коду Java. Проведений синтаксичний аналіз дозволив виявити проблемні та помилкові блоки в коді, а також ідентифікувати спроби шахрайства (підроблення виводу програми замість реалізації алгоритму). **Висновки**. Більшість існуючих систем автоматичного оцінювання робіт студентів основані на тестуванні, тобто співставленні входів і виходів програми. На відміну від них запропонований в роботі метод передбачає аналіз синтаксичної структури програми, що дозволяє точно визначити місце та характер допущених помилок. Інтелектуальний аналіз зібраних при цьому даних дозволить розробити точні рекомендації для студентів щодо покращення коду. Запропоновані засоби можуть бути частиною Intelligent Tutoring System для IT спеціальностей.

Ключові слова*:* обробка даних; інтелектуальний аналіз даних; інтелектуальні системи оцінки; аналіз програмного коду; динамічний аналіз програмного коду; генерація зворотного зв'язку.

Селютін Денис Анатолійович – аспірант каф. комп'ютерних наук та інформаційних технологій, Національний аерокосмічний університет ім. М. Є. Жуковського «Харківський авіаційний інститут», Харків, Україна.

Яшина Олена Сергіївна – канд. техн. наук, доц., доц. каф. комп'ютерних наук та інформаційних технологій, Національний аерокосмічний університет ім. М. Є. Жуковського «Харківський авіаційний інститут», Харків, Україна.

Denys Seliutin – PhD Student of the Computer Sciences and Information Technologies Department, National Aerospace University "Kharkiv Aviation Institute", Kharkiv, Ukraine,
e-mail: denis.selutin.ds@gmail.com, ORCID: 0009-0000-2843-9689.

Elena Yashyna – PhD in Information Technologies, Associate Professor at the Computer Sciences and Information Technologies Department, National Aerospace University "Kharkiv Aviation Institute", Kharkiv, Ukraine,
e-mail: o.yashina@khai.edu, ORCID: 0000-0003-2459-1151.