Medicon Engineering Themes
Volume 2 Issue 4 April 2022
Research Article

Application of Java Relationship Graphs to Academics for Detection of Plagiarism in Java Projects

Divya Tyagi*, Ritu Arora and Yashvardhan Sharma

Birla Institute of Technology and Science, Pilani, India

***Corresponding Author:** Divya Tyagi, Birla Institute of Technology and Science, Pilani, India.

Received: March 17, 2022; **Published:** March 31, 2022

Abstract

In today's online learning environment, plagiarism detection tools are increasingly used by teachers and instructors to restrain students from plagiarism. Moreover, need for plagiarism detection tools to detect plagiarism in programming assignments has also increased manifolds. In this paper, we present the application of Neo4j Graph Databases to detect similarity between Java program submissions by students, in academics. This is done by converting a Java program into a specialized dependency graph and then implementing various comparison techniques on this graph. The two graph comparison techniques proposed and implemented in this paper are based on structural comparison of graphs by node-type count comparison and elemental comparison of method nodes in graphs by body-element-count comparison. The results of these two techniques are combined with the call graph-based technique, proposed in an earlier work, to calculate overall similarity index between program codes. This study captures a large category of changes that may be introduced to the code for plagiarism.

Keywords: Plagiarism Detection; Computer Science; Object Oriented Programming; Java Programs, JavaRelationshipGraphs (JRG); Dependency Graphs; Call Graphs; Neo4j Graph Databases

Introduction

With the advancement of technology and the Internet, it has become easier for students to copy assignments from their peers. Plagiarism has various negative repercussions. Firstly, a student can take the credit for the work not originally produced by him/her. Moreover, by plagiarizing someone else's work, the student gets no knowledge of the subject matter. And this, in the long run, undermines the validity of academic degrees. According to the survey conducted by Donald McCabe, Rutgers University (2005) [1], 70% of total students surveyed (over 70, 000 US graduate and undergraduate students) were found to introduce plagiarism. Not only in the universities, but the plagiarism cases are common in companies also. These cases are graver and concerning, as they may even cause court disputes. However, our focus is primarily on detection of plagiarism in academics. Plagiarism detection has become even more important in today's online learning environment. Due to the ongoing pandemic situation, more and more schools and universities have been forced to shift to the use of various online platforms for teaching course content, and for conducting and evaluating laboratory assignments. As a result of this plagiarism cases have increased tremendously. So, detecting plagiarism is important and tedious job on the part of instructors.

Objective

The main aim of this research is to detect similarity between Java source codes. The objectives of this work are - converting Java programs into specialized dependency graphs (based on previous work), detecting plagiarism using structural and elemental comparison of graphs and devise a powerful similarity index to detect plagiarism.

For this study, we use JavaRelationshipGraphs (JRG) [2-3] stored using Neo4j Graph Databases. JRG is a specialized dependency graph and Java program code can be easily converted into its equivalent JRG using the JRG framework [3]. Moreover, this study is complementary to an already existing graph-based technique of detecting plagiarism [4]. In the previous study [4], call graphs were obtained from JRG and further used to calculate the similarity index between program codes. The motivation is to further enhance the results of the previously conducted research. In this paper, we pre- sent the novel approach by combining the results of the call graph-based technique with the structural and elemental comparison techniques, proposed in this work. Our goal is not to find true definite cases of plagiarism, but we are more focused on cases that are suspicious enough so that a teacher should assess them. The proposed technique of structural comparison of graphs by node-type count comparison, counts and matches the number of nodes of a certain type and uses this count to compare the structure of graphs obtained from their corresponding Java programs. Moreover, the elemental comparison of method nodes in graphs by body-element-count comparison technique, counts and matches various basic elements existing in a Java program body like number of lines, blocks and keywords (if, while, for, new, etc.).

The remaining paper is structured as follows: Section 2 discusses the relevant works in the field of plagiarism detection while section 3 presents the proposed techniques. In Section 4, the proposed method is demonstrated using an example and results are discussed. And finally, Section 5 concludes the paper with a brief note on future work.

Related Works

With the advancement of technology, different approaches have been used by researchers, including text and token-based comparison [5], graph-based [4, 6-9] and tree-based techniques [7]. In [10], a token sequence is generated using Java class files and then similarity is evaluated using adaptive local alignment. In CAPlag [11] attribute-based method is combined with structural based method and hidden structural similarities are analyzed in a program along with evaluating important attribute-based metrics.

In graph-based techniques, dependency graphs are very effective in representing the relationship between different program elements in the source code. Static call graphs technique [4] that is based on application of JRG [3] represent the method calls between different methods in static source code. In [6], the author extracted the call graph from Abstract Syntax Tree of program source code and then used n-grams technique. Hyun [7] used a combination of syntactic and dynamic information to compare programs. For syntactic information, parse trees are compared using specialized tree kernels, while for retrieving dynamic information, call graphs are studied for isomorphism. In Program Interaction Dependency Graph [8], the behavior of programs in terms of data used and how it interacts with the system is represented. BPlag [9] analyses the two programs based on their execution behavior and represent a program in a graph-based format.

Some of the most popular existing detection tools are:

- MOSS [11] - It is a freely available tool. Information of the algorithm used, and test results are not provided for MOSS. It detects plagiarism in programming languages – C, ML, Pascal, C++, Java, Scheme, Lisp, and Ada.
- JPLAG - It works for the programs written in C, Java, Scheme, C++, free text languages. Test results are not published for JPlag. However, it makes use of the Greedy String Tiling (GST) [12] technique to compare pairs of strings for evaluating similarity and it also provides a robust graphical interface.

Proposed Method

In this work, we extend the usage of JavaRelationshipGraph (JRG) framework [3] to academics for detection of plagiarism in Java projects. The framework converts a Java project into a dependency graph, comprising of the seven program elements that exist in any Java program, namely – project, package, class, interface, method, class-attribute, and method-attribute. Each program element is connected to its owner element using ownership relationship. Additionally, a Java project might consist of five different types of dependency relationships – extends, implements, imports, uses and calls. Moreover, a JRG is created using Neo4j Graph Databases. Graph databases are based on property-graph model and consist of entities, called nodes, which are connected to other nodes through

directed relationships, called edges. Both nodes and edges can have any number of properties (key-value pairs) associated with them. In a JRG, nodes represent the program elements and edges represent the ownership and dependency relationships. Further, we convert Java projects submitted by students, into corresponding JRG and apply various algorithms to compare these graphs, calculate similarity scores and hence detect plagiarism. One such technique was proposed in our previous work [4], in which we obtained the call graph from the JRG and then compared the call graphs of various submissions and obtained a measure of similarity score. In this work, we add two more techniques for comparing graphs obtained from Java projects and calculate corresponding similarity scores. Finally, a combined similarity score is calculated, which combines the similarity score obtained from all the three techniques. This helps us achieve a more effective similarity score, better accuracy, and less false positives. The sub-sections below discuss the two new proposed techniques.

Structural Comparison of Graphs by Node and Edge Count Comparison Technique

In the first technique, we carry out structural comparison of JRGs created from Java programs by counting the nodes and edges of the same type. In this study, we count only five type of nodes – class, method, interface, class-attribute and method-attribute nodes. Project and Package nodes are not counted because they mostly remain the same. Students usually do not make changes to these higher-level nodes while introducing plagiarism. Additionally, while counting edges, ownership edges are not counted because these edges connect nodes together and hence are indirectly counted while counting the nodes. Moreover, only three types of dependency edges are counted – calls, uses, and imports. Dependency edges of types - extends and implements are not counted for the same reason as discussed above for project and package nodes.

After calculating these nodes/edges for the two JRGs, a similarity % between them is calculated using the exponential weighted difference formula which is given by:

$$Structural\ Similarity\ \% = (e^{-\sum_{i=1}^{8}(weight)_i|difference|_i}) * 100 \qquad (1)$$

Where *difference* is the subtraction of corresponding count values of nodes and edges between the two graphs. If the difference in nodes/edges counts (for corresponding node/edge type) is zero, this implies there are high chances of code being plagiarized because the structure is similar. Absolute value of difference is taken to avoid the effect of nullification. And *i* moves from 1 to 8 since there are a total of eight elements (five node and three edge types). The points that were kept in mind while devising the formula are - the function used should be a decreasing function because if summation of weighted differences is small, then similarity % should be high. And the mathematical function range should be (0,1). Therefore, an exponential function is considered.

For deciding the weights, the criteria considered is if the element is easier to modify or more prone to plagiarism modifications, then its corresponding weight is kept low, because it's significance in the similarity percentage should be less. We can understand this as it is less likely that students will introduce dummy classes/interfaces to make plagiarism changes, so class and interface nodes are given more importance. While a student tries to add dummy attributes/methods or break methods into smaller, therefore method, attribute and method-attribute nodes are given less weight. Weights for dependency edges are also decided in a similar manner. The weights are adjusted manually by analyzing similarity % after introducing small plagiarism changes in programs to decide final weights. The weights thus obtained are given in Table 1.

S No	*Node/Edge type*	*Weight*
1.	Interface Nodes	0.05
2.	Class Nodes	0.04
3.	Method Nodes	0.02
4.	Class-attribute Nodes	0.01
5.	Method-attribute nodes	0.01
6.	Calls Edges	0.02
7.	Uses Edges	0.03
8.	Import Edges	0.01

Table 1: Weights corresponding to different nodes and edges.

The plagiarism changes detected by the proposed structural comparison technique are:

- Plagiarism changes introduced to comments, white-spaces, and code formatting - since these elements are not considered in the counting process, so any changes made to these do not cause changes to the similarity %.

- Plagiarism changes introduced at Class/Interface level
  - Renaming or changing the signature of class(es) / interface(s)
  - Rearranging attributes and methods within a class/interface
  - Addition or deletion of classes / interfaces
- Plagiarism changes introduced at Method level
  - Renaming or changing the signature of methods
  - Reordering code blocks within a method
  - Addition or deletion of method(s)
  - Breaking a method into smaller methods
  - Replacing program statements with equivalent semantics in a method
  - Modifying expressions and changing decision logic within a method
- Plagiarism changes introduced at Attribute level for class and methods
  - Renaming or changing the data types of attribute(s)
  - Addition or deletion of attribute(s)

If the student has just changed the names or signatures of classes, interfaces, methods or/and attributes, then the weighted difference of nodes/edges obtained will be zero. Similarly, rearrangement of methods and attributes within a class and reordering code blocks within a method results in a weighted difference which is also zero. Changing method statements like replacing if statements with semantically equivalent switch cases or modifying expressions does not change the number of methods in the code. Therefore, any such changes deliberately introduced in the code will also be detected by our technique. The reasoning for Interface level changes will be the same as that of class level changes. If the student tries to break existing methods into smaller ones, then most likely he/she will break it into two. This will increase both method nodes and calling edges by one. But as weights corresponding to both are low, therefore similarity % does not get affected drastically. And if dummy methods are created, then also this will increase method count but method calls remain the same, hence it does not also change similarity % significantly. Similar is the case for introduction of dummy attrib- utes and classes. Deletion of class, interface, method, and attribute has the same effect as that of addition, because the absolute difference of count will remain the same.

Elemental Comparison of Method Nodes in Graphs by Body-Element-Count Comparison Technique

In this technique, the JRGs obtained from the program submissions are compared for elemental similarities of the basic elements that exist within the method-body for corresponding method nodes. Importantly, while creation of a JRG, the method-body is stored as

a property of the method node, in the form of key-value pair and be easily retrieved. Moreover, different elements in the method-body are counted and then the overall similarity % is calculated. The basic elements used are:

- Semicolons – to count the total number of lines in the method-body
- Curly brackets – to count the total number of blocks in the method-body
- Full stop – to count the total number of function calls
- Keywords – to count various elements in the method-body; keywords considered here are: - break, continue, try, catch, for, while, if, else, switch, case and new.

Moreover, as generally students rename the methods to introduce plagiarism, so to find out which method node in one graph is to be compared with which method node in another graph, three things are matched in the two methods - parameter list, return type and modifier. And the exponential weighted difference formula used to calculate similarity percentage between two method nodes is given by:

$$Elemental\ Similarity\ \% = (e^{-\sum_{i=1}^{14}(weight)_i|difference|_i}) * 100 \qquad (2)$$

Here, *weight, difference* and *i* have the same meaning as in Eq. (1). We are considering total 14 elements so *i* moves from 1 to 14. For deciding the weights, the same criteria are adopted as in Eq. (1). For instance, since it is less likely that a student changes break, continue, try and catch expressions, therefore they have more weights. While it is quite easy to alter the number of lines by simply introducing dummy print statements; hence given less weight. Table 2 lists the weights assigned to various elements, in increasing order of difficulty in modifying the element – which also leads to decreasing order of weights. The easier it is to change an element, the lower the weight assigned to it.

The steps involved are detailed in Algorithm 1. If a student copies the entire code or changes the comments and indentations or renames the methods or rearranges the attributes, then the similarity % calculated will be 100 %. Moreover, if a student changes statements like replacing for-loop with semantically equivalent while-loop, this will not change the similarity % significantly as weights associated with these keywords are very low. Moreover, if a student tries to break a method into more methods, it will increase the number of blocks and parenthesis, which again have less weight age, while other element count will remain same; therefore, similarity % will not change drastically. Additionally, if there is introduction of dummy method(s), then these method node(s) in one graph will not correspond to any method node(s) in another graph, so will be dropped without affecting the similarity %.

S No	*Description*	*Weight*
1.	break keyword	0.03
2.	continue keyword	0.03
3.	try keyword	0.03
4.	catch keyword	0.03
5.	for keyword	0.02
6.	while keyword	0.02
7.	if keyword	0.02
8.	else keyword	0.02
9.	switch keyword	0.02
10.	case keyword	0.02
11.	new keyword	0.02
12.	blocks	0.02
13.	function calls	0.01
14.	lines	0.01

Table 2: Weights corresponding to various elements of the method-body.

Input	JRG1 and JRG2 - Java Relationship graphs of two Java Projects
Output	Similarity Percentage
Step 1	Run the CQL query on the JRG to obtain method name, return type, parame- ter list, modifier, and metod-body of all the method nodes for both JRGs
Step 2	Create an array list of methods and store the details obtained from Step 1 as the attributes of Method object.
Step 3	Calculate the counts of above-mentioned elements with respect to each method and store in an array list.
Step 4	To find pairs of most similar methods in two graphs, compare parameter list, return type and modifier of every combination of method node pairs.
Step 5	Calculate the similarity between pairs of most similar method nodes obtained in Step 4, using the exponential formula designed for this technique.
Step 6	To obtain overall similarity score, find the average of similarity score of different pairs of method nodes calculated in Step 5.

Algorithm 1: To calculate Elemental level-based Similarity Percentage.

Combining Results with Call Graph Based Technique

In the call graph technique [4], the author converts each program into its JRG and then extracts its call graph from it. Then in the next step, compares call graphs of two program for isomorphism and calculates similarity index. The formula used to calculate the similarity percentage is given by:

$$Call\ Graph\ Similarity\ \% = \left(\frac{\text{Number of common elements}}{\text{Total number of elements in the arraylist}}\right) * 100 \qquad (3)$$

This technique has higher chances of false positives, if considered standalone. Changes like addition of new method(s) by breaking a method into small ones, or introduction of additional dummy or useful attributes, are few types of changes that cannot be captured by the call graph technique. Therefore, in this work, we improvise on our results by extracting more structural and elemental features from the JRG of a program. After calculating additional similarity indices, we combine our results with the results of call graph technique. The average of all three is computed for overall similarity index.

For a large submission set of Java projects, each project is compared with all the other projects and an overall similarity % is calculated. Finally, the similarity % for a single project is calculated as the maximum of all the similarity scores obtained by comparison with all the projects. The threshold similarity percentage beyond which we will consider that code is plagiarized is 90%. This number is calculated experimentally.

The source code for the proposed technique is open sourced at: https://github.com/DivyaTyagi1111/Plagiarism_Detection/blob/main/Communi-tyNeo4jMaven/src/org/jrg/parser/Structural_Elemental_Similarity.java

Experiment and Result

Let us consider an example Java program (Fig. 1). File EvenAndOdd.java is the original file which has three classes. Class *Even* contains a method *is_even()*. Class *Odd* has one class attribute *value* and a method *is_odd()*. Class *EvenAndOdd* provides the usage of the entire program. It contains the *main()* method which takes an input number and tells if it is even or odd. TestNumber.java is created by introducing changes: Some comments are added. All the classes and methods *is_even()* and *is_odd()* are renamed. A dummy class attribute named dummy is added in class *CheckEven*. The *if* expression is replaced with else expression in the main() method.

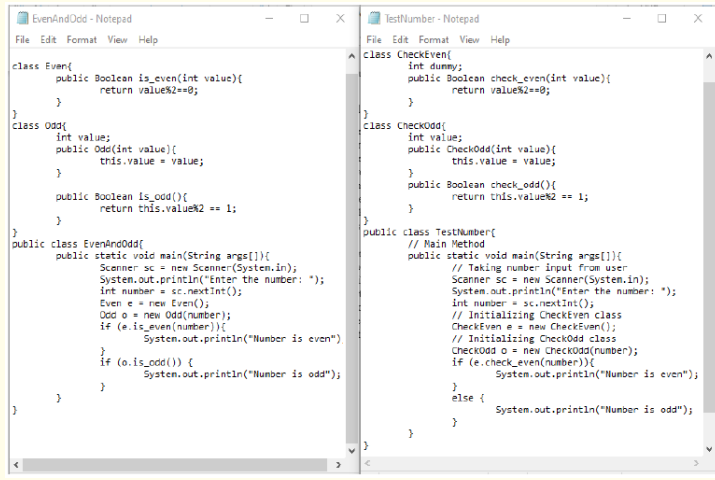

Figure 1: Original and Plagiarized Code.

Fig. 2 shows the results of running the proposed technique on the above code. JRG-1 and JRG-2 corresponds to EvenAndOdd.java, the original file and TestNumber.java, the plagiarized file, respectively. It depicts the similarity index by call graph, structural and elemental comparison techniques, and overall similarity % for the two JRGs. It also compares and displays the counts of different types of nodes and edges in the JRGs. The analogous method nodes in the two graphs are also presented. As the call-graph based similarity is 100%, so one may misinterpret that this is the best technique, but the fact is it does not consider many aspects of the code as discussed earlier. The overall similarity % is 99.26% and as our threshold is 90%, so we accept the two codes to be plagiarized. Individually also, the similarity % obtained using the structural comparison technique and elemental comparison technique are above 90%. Even after introducing one attribute, replacing if with else expression, renaming classes and methods and adding comments, we get a high similarity %. Hence, the proposed technique is effective.

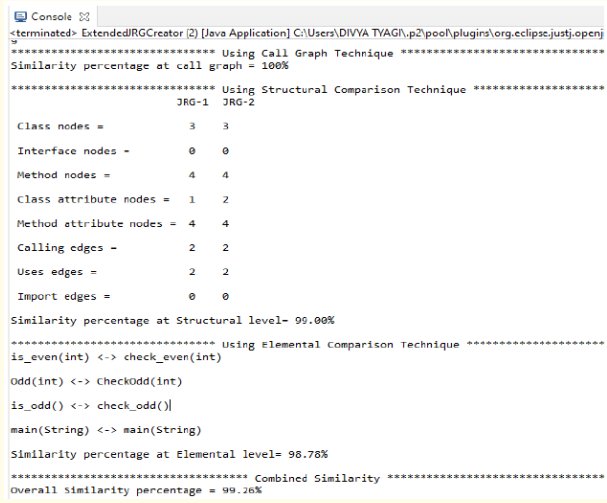

Figure 2: Results of proposed technique.

Conclusion and Future Work

This paper proposes a novel structural and elemental comparison-based approach for detecting plagiarism in Java programs. The

technique makes use of dependency graphs generated from the Java program to make the overall plagiarism detection more robust. This technique uses the Neo4j graph database at its core. The proposed technique and the associated algorithm calculate the similarity index between two programs. Using this technique, a large category of changes can be captured that may be introduced to the code for plagiarism. However, there exists a limitation of the proposed elemental comparison technique used to compare method-body for method nodes. It will give false results if there are overridden methods in the classes, since overridden methods have same signature. This can be overcome by prefixing the method names with the all the parent class names at the time of creation of JRG.

There are various directions for future work in this area. Firstly, the technique needs to be evaluated against diverse changes and real plagiarized program datasets. Secondly, as the proposed method deals only at the structural and elemental comparison level, it can also be combined with semantic and behavioral tools to enhance the results. Thirdly, different graphs like inheritance graph, control-flow graph, etc. could be extracted from JRG and used further to detect plagiarism. All these future works combined can help to investigate further if two programs are plagiarized or not.

Acknowledgements

The authors would like to convey their sincere thanks to the Department of Science and Technology (ICPS Division), New Delhi, India, for providing financial assistance under the Data Science (DS) Research of Interdisciplinary Cyber Physical Systems (ICPS) Programme [DST/ICPS/CLUSTER/Data Science/2018/Pro-posal-16:(T-856)] at the department of CSIS, BITS, Pilani, India. Authors are also thankful to the authorities of BITS, Pilani, to provide basic infrastructure facilities during the preparation of the paper.

References

1. McCabe DL. "Cheating among college and university students: A North American perspective". International Journal for Educational Integrity 1.1 (2005).
2. Arora R, Goel S and Mittal RK. "Using dependency graphs to support collaboration over GitHub: The Neo4j graph database approach". In 2016 Ninth International Conference on Contemporary Computing, IEEE (IC3) (2016): 1-7.
3. Arora R and Goel S. "Java Relationship Graphs (JRG) Transforming Java Projects into Graphs using Neo4j Graph Databases". In Proceedings of the 2nd International Conference on Software Engineering and Information Management (2019): 80-84.
4. Arora R, Maurya AM and Sharma Y. "Application of Java Relationship Graphs (JRG) to plagiarism detection in Java Projects: A Neo4j Graph Database Approach". In Proceedings of the 4th International Conference on Software Engineering and Information Management (ICSIM), Yokohama, Japan, ACM (2021).
5. Cavnar WB and Trenkle JM. "N-gram-based text categorization". In Proceedings of SDAIR-94, 3rd annual symposium on document analysis and information retrieval 161175 (1994).
6. Steier M. Discovering plagiarism in entry level programming assignments (2017).
7. Song HJ, Park SB and Park SY. "Computation of program source code similarity by composition of parse tree and call graph". Mathematical Problems in Engineering (2015).
8. Cheers H and Lin Y. "A Novel Graph-Based Program Representation for Java Code Plagiarism Detection". In Proceedings of the 3rd International Conference on Software Engineering and Information Management (2020): 115-122.
9. Cheers H, Lin Y and Smith SP. "Academic Source Code Plagiarism Detection by Measuring Program Behavioral Similarity". IEEE Access 9 (2021): 50391-50412.
10. Ji JH, Woo G and Cho HG. "A plagiarism detection technique for Java program using bytecode analysis". In 2008 Third International Conference on Convergence and Hybrid Information Technology, IEEE 1 (2008): 1092-1098.
11. Menai MEB and Al-Hassoun NS. "Similarity detection in Java programming assignments". In 2010 5th International Conference on Computer Science & Education, IEEE (2010): 356-361.
12. Wise MJ. String similarity via greedy string tiling and running Karp-Rabin matching". 119.1 (1993): 1-17.

Volume 2 Issue 4 April 2022