Detecting Similarity in Student Multi-procedure Programs using Program Structure

Full Paper

SACLA 2019

©The authors/SACLA

Karen Bradshaw[0000−0003−3979−5675] and Vongai Chindeka

Dept. of Computer Science, Rhodes University, South Africa
`k.bradshaw@ru.ac.za`

Abstract. Plagiarism is prevalent in most undergraduate programming courses, including those where more advanced programming is taught. Typical strategies used to avoid detection include changing variable names and adding empty spaces or comments to the code. Although these changes affect the visual components of the source code, the underlying structure of the code remains the same. This similarity in structure can indicate that plagiarism has taken place.
A similarity detection system has been developed to detect the similarity in the structure of two given programs. The system works in two phases, the first phase parses the source code and creates a syntax tree, representing the syntactical structure for each of the programs. The second phase takes as inputs two program syntax trees and applies various comparison algorithms to detect their similarity. The result of the comparison allows the system to report a result from one of four similarity categories: identical structure, isomorphic structure, containing many structural similarities, and containing few structural similarities. Empirical tests on example programs show that the prototype implementation is effective in detecting plagiarism in source code, although in some cases manual checking is needed to confirm the plagiarism.

Keywords: Plagiarism Detection, Code Structure, Student Code.

1 Introduction

Plagiarism occurs when one person tries to pass off someone else's work as his/her own [1]. This can mean copying word-for-word or copying phrases or smaller parts of the work, which if done without quoting and/or citing the originator of the work, results in plagiarism. This is a common occurrence in academic environments. In undergraduate programming courses, once one student obtains a solution to an assigned programming task, it is often replicated by other students.

Assignments for more advanced programming courses typically involve more complex programs that can be constructed in a variety of ways. Students in these course should be aware of a wider variety of programming constructs available in the programming language being used. Therefore, if there is similarity in the structure of large segments of code it can be a sign that plagiarism has taken place.

Plagiarism of code often involves techniques that try to hide the plagiarism, such as various code obfuscation techniques. Students commonly resort to simple techniques, such as statement reordering, instruction splitting or aggregation, loop unwinding or introducing white spaces and comments [2]. Although these changes affect the appearance of the source code, they do not alter the syntactic structure thereof.

A functional plagiarism detection system is a possible solution to preventing plagiarism in an academic environment by encouraging students to avoid the penalties attached to plagiarism [3]. The main obstacle, however, in detecting plagiarism in an undergraduate programming course is the sheer volume of student programs that need to be assessed. Thus, the probability of detecting similar programs is reduced with larger class sizes and more complex programs.

Plagiarism checkers for text are more common than those for source code, which means that plagiarism in computer programs is often done manually. Manual checking can be tricked by drastically changing the visual appearance of the program; this does not alter the structure of the program, but makes it more difficult for a human to detect.

The aim of this research is to generate a similarity detection system that bases its similarity comparison solely on the static structure of the programs being compared without any preprocessing of the source code or dynamic analysis thereof. Such a system can be useful in the detection of plagiarism in an academic setting and specifically in more advanced programming courses with more complex programming assignments. Such a plagiarism detection system would not be useful in introductory courses, where the structure of the simple programs tends to be much the same even without the occurrence of plagiarism. The effectiveness of the similarity detection system in detecting plagiarism is also investigated.

The rest of this paper is organized as follows: Section 2 introduces and discusses related studies in plagiarism detection and tree comparison. Section 3.1 gives a high level overview of the similarity detection system developed. Section 4 explains the various algorithms used in the comparison of the program structure, while Section 5 discusses the results of simple test cases. Section 6 concludes the paper and also mentions future work to improve the similarity detector.

2 Related Work

2.1 Plagiarism Detection

Most existing similarity detection systems for source code use either a metrics-driven or syntax-based approach to determine the degree of similarity between programs [1]. The metrics used in the former approach can come from a software engineering perspective, such as the number of each data structure type used or the cyclomatic complexity of the program's control flow. Cyclomatic complexity is a metric based on the number of linearly independent paths in the program's control flow. Metrics can also come from a linguistic or technical aspect such as variable names, indentations and other layout conventions used as well as the number of comments in the code and their quality. These types of metrics can help determine a student's program authoring style. The linguistics centered metrics tend to be more useful for smaller and simpler programs such as those written in introductory programming courses.

The methods that are used by systems using a syntax-based approach, can be categorised into static source code comparison, static executable code comparison, dynamic control flow based, dynamic API based methods as well as dynamic value based methods. JPlag, YAP3 and MOSS are three well-known systems that detect similarities in source code by using a static source code comparison. MOSS (Measure of Software Similarity)[1] is a system developed in 1994 that uses fingerprinting to detect similarity in programs by using a fingerprinting algorithm called winnowing [4]. This algorithm makes detection faster, but at the expense of sacrificing some detection capabilities.

YAP3 [5] works in two phases. In the first phase it removes comments and string constants, changes all letters to lowercase, maps statements that do the same things, reorders the functions in the code to their calling orders by expanding them to their full token sequences and removes all the tokens that are not in the lexicon of the language used. The second phase is the comparison phase, which uses an algorithm that caters for the scrambling of independent segments of code called the Running-Karp-Rabin Greedy-String-Tiling (RKR-GST) algorithm. This is a similar algorithm to that used by the UNIX utility "sdiff".

In JPlag [6], the first phase scans and parses the program and converts it into token strings based on the program structures. In the second phase the tokens of the two programs are compared using the "Greedy String Tiling" algorithm. Token strings are compared according to the following rules: any token from one program must match with only one token in the other program, substrings are matched without relating to their positions (so that changing the positions of code segments is ineffective) and the matching of long substrings is more indicative than the matching of short substrings so they are favoured more.

The JPlag system is a web-based application so the result is given as a set of HTML pages providing an in-depth description of the similarity. Assuming a pair of programs as input, the system provides results (in the form of a histogram) for each possible match found in the files. Percentages less than 5% do not indicate plagiarism, while a similarity of 100% shows definite plagiarism. Any percentage in between requires further manual investigation to determine if it is plagiarism.

Systems like MOSS, YAP3 and JPlag were designed to work with a number of languages and generally provide good results. However, small changes in the

[1] https://theory.stanford.edu/ aiken/moss/

source code such as reordering statements in the case of JPlag and slightly more complex reorderings in the case of MOSS, cannot be detected.

CSPLAG [7] is a solution that attempts to eliminate these shortcomings by using both syntax and semantic knowledge to detect copied code. CSPLAG however, focuses only on languages compiled within the .NET framework. Comparisons of the source code, the abstract syntax trees, as well as the .NET produced intermediate code are carried out to produce results that are superior to those of JPlag and MOSS in detecting a variety of different plagiarism scenarios.

2.2 Tree Comparison Studies

A tree is a special form of a graph, with only one edge connecting any two nodes, that is, without cycles or loops [8]. Tree structures are typically used to represent hierarchical data. A tree is considered rooted if it has a node that is selected to be a root node and is ordered if the children of each node are in a specific order, for example, increasing values of the nodes from left to right.

Isomorphism is a useful concept in comparing trees; two trees are isomorphic if the nodes in one tree can be mapped to the nodes in the other tree [9]. This means that an isomorphic tree can be obtained by switching around the children of the nodes of another tree. The two trees shown in Fig. 1 are isomorphic.

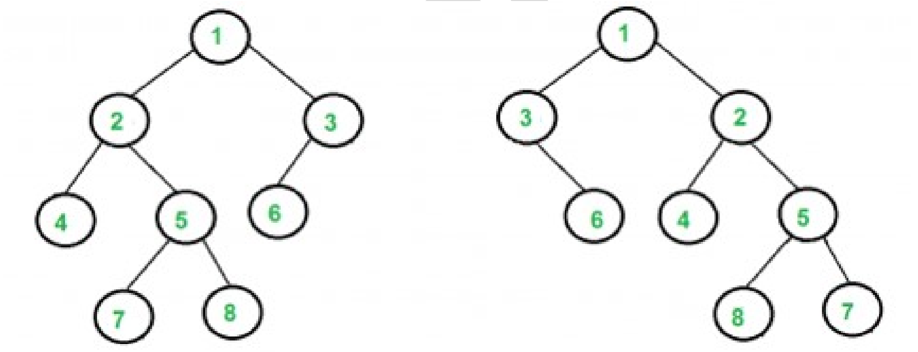


Fig. 1. Isomorphic trees.

An algorithm to determine whether two trees are isomorphic was developed by Aho et al. [9]. The algorithm applies to trees that are rooted and unordered. In the algorithm, an integer is allocated to each node, beginning at the leaf nodes of the trees, in such a way that the trees are isomorphic if and only if the same integer is assigned to the roots of the trees. This algorithm works in $O(n)$ time for n nodes.

Itokawa et al. [10] proposed an algorithm for tree pattern matching using succinct data structures. Succinct data structures are a representation of data that takes a minimal amount of space but remains usable. This representation (of which there are many forms) is an efficient encoding that does not require decoding so that it may be used in query operations. The succinct representation for trees defined in the algorithm proposed by Itokawa et al. uses matching pairs of parentheses to represent a node's information. This depth-first unary degree sequence (DFUDS) representation is a succinct representation for ordered trees. For a tree with n nodes, the representation is a sequence of $2 * n$ opening and closing parentheses, where an opening parenthesis is emitted when a node is first encountered, and the closing parenthesis is emitted when returning to this node after the depth first traversal of the respective subtree. Moreover, given the DFUDS representation of two trees p and t, the algorithm returns true if a substitution θ of one tree, called $p\theta$, exists so that t and $p\theta$ are isomorphic.

Comparison of trees is often done using methods that calculate the maximum agreement sub-tree of the two trees being compared, as shown in Fig. 2. The maximum agreement sub-tree is a tree that includes all possible matching nodes of the two trees based on common ancestors [11]. The authors presented an algorithm for comparing trees that they claim is faster than pre-existing algorithms. This algorithm applies to graphs that have nodes that are labelled. The way that the graphs are labelled is not limited to a specific way for the algorithm to work.

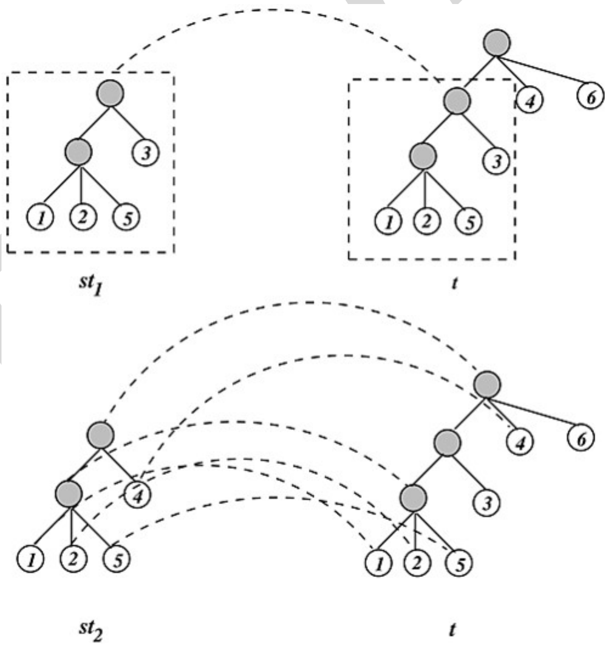


Fig. 2. Graph comparison, taken from [12].

3 Proposed Similarity Detection System

3.1 Design Overview

For the prototype implementation of the proposed detection system, an experimental language Parva [13] with a small set of programming constructs, was used as the source language. Similar to YAP and JPlag, the proposed system comprises two phases, but unlike these systems, syntax trees are used as the representation of the source programs.

In the first phase, a parser translates the given input programs into their respective syntax trees, which are subsequently input to the second phase, where various comparison algorithms are applied to determine the similarity score of the input programs.

To implement the first phase, a Parva parser, written in C#, was developed using a compiler generator, Coco/R[2]. The output from this parser is a concrete syntax tree, which is serialised and stored in eXtensible Markup Language (XML) format. XML was chosen due to its suitability for use in phase two of the similarity detection system, as well as for ease of viewing the tree structures during manual validation of the similarity of the test programs. Although there are C# libraries that implement XML serialising and deserialising, the proposed system uses a custom implementation, which allowed a more flexible and accurate representation of the hierarchical format of the tree structures.

The similarity detection phase takes in the XML representations of the Parva source code files and deserialises these into the `Tree<string>` data structure form consisting of a collection of nodes of type `TreeNode`. Each node has a value field of a generic type, a property indicating the level in the tree where the node is located, a pointer to its parent node and finally, a list of its children nodes.

During the deserialisation, a new node is created by providing the value of the node as well as the node's parent node. This creates a level 0 node with the given value and parent, as well as an empty list of children. Functionality for adding a child to the node, using the `AddChild(TreeNode<T> child)` method, is provided for. The tree can also be represented in a string form using the `ToString()` method, which does a depth first traversal of the tree rooted at the current node and returns a string containing the values of the node itself, its parent node and its children nodes, for each node that is encountered during the traversal. This string form is a version of succinct representation that is useful because simple string handing operations can replace handling bulky tree data structures during the comparison.

The resulting trees are compared using three algorithms: a brute force algorithm, an isomorphism algorithm as well as an algorithm based on succinct representation. An overview of the similarity detection phase is illustrated in Fig. 3.

[2] http://www.ssw.uni-linz.ac.at/coco/

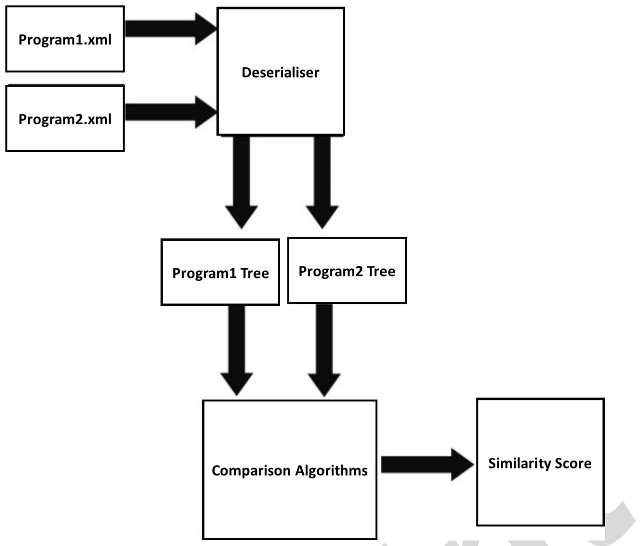


Fig. 3. Similarity detection phase.

3.2 Example Program

```
void Main ()
 {  const votingAge = 18; }
```

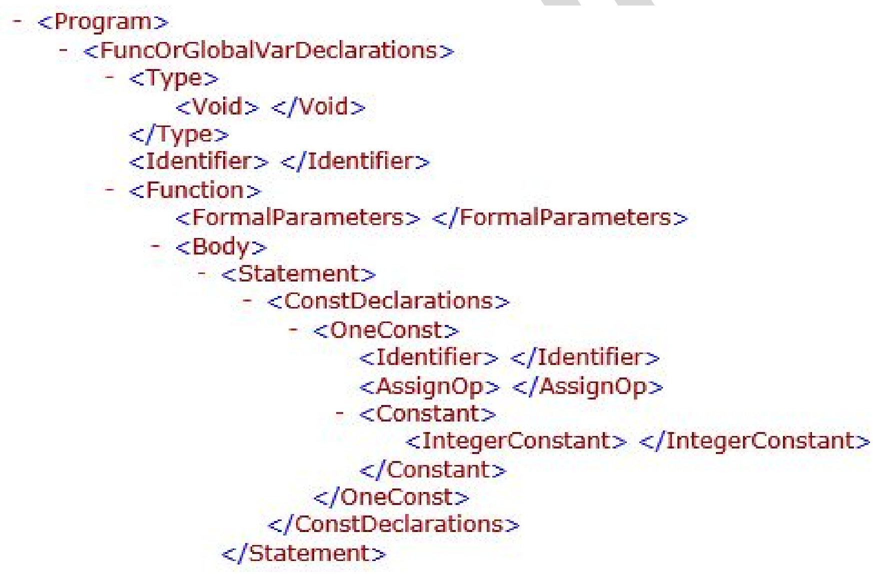
Listing 1.1. Minimal Parva program.

The minimal example Parva code in Listing 1.1 contains a main function with only one statement, a constant variable declaration. The string representation of the parse tree generated by parsing this declaration, shown in Listing 1.2, gives a list of the nodes of the tree iterated through in a depth-first order. Each node is represented by the level and value of the node followed by the value of the parent node as well as values of the children nodes if they exist. The XML representation of the parse tree is illustrated in Fig. 4.

```
0 Node Value: Program Parent Value: null  Children:
    FuncOrGlobalVarDeclarations
1 Node Value: FuncOrGlobalVarDeclarations Parent Value:
    Program Children: Type Identifier Function
2 Node Value: Type  Parent Value: FuncOrGlobalVarDeclarations
    Children: Void
3 Node Value: Void  Parent Value: Type  Children:
2 Node Value: Identifier  Parent Value:
    FuncOrGlobalVarDeclarations Children:
2 Node Value: Function  Parent Value:
    FuncOrGlobalVarDeclarations Children: FormalParameters
    Body
```

```
3 Node Value: FormalParameters  Parent Value: Function  Children:
3 Node Value: Body  Parent Value: Function  Children: Statement Statement Statement Statement Statement Statement Statement Statement
4 Node Value: Statement Parent Value: Body  Children: ConstDeclarations
5 Node Value: ConstDeclarations Parent Value: Statement  Children: OneConst
6 Node Value: OneConst  Parent Value: ConstDeclarations  Children: Identifier AssignOp Constant
7 Node Value: Identifier  Parent Value: OneConst  Children:
7 Node Value: AssignOp  Parent Value: OneConst  Children:
7 Node Value: Constant  Parent Value: OneConst  Children: IntegerConstant
8 Node Value: IntegerConstant Parent Value: Constant  Children :
```

Listing 1.2. String representation of the syntax tree for the minimal Parva program.



```
- <Program>
   - <FuncOrGlobalVarDeclarations>
      - <Type>
           <Void> </Void>
        </Type>
        <Identifier> </Identifier>
      - <Function>
           <FormalParameters> </FormalParameters>
         - <Body>
            - <Statement>
               - <ConstDeclarations>
                  - <OneConst>
                       <Identifier> </Identifier>
                       <AssignOp> </AssignOp>
                     - <Constant>
                          <IntegerConstant> </IntegerConstant>
                       </Constant>
                    </OneConst>
                 </ConstDeclarations>
              </Statement>
```

Fig. 4. XML representation of part of the parse tree for minimal Parva program.

4 Comparison Algorithms

The report on the similarity output by the similarity detection phase involves one of four categories: category 1 (identical), category 2 (isomorphic), category 3 (contains many similarities) and category 4 (contains few similarities). The three chosen comparison algorithms give results that classify the program similarity into these categories.

In addition to the basic string representation, a succinct representation was also created for use in some of the comparison algorithms. As discussed in Section 2.2, a succinct data structure represents the data in a way that takes up minimal space, but allowing operations on the data to be possible without the requirement of decoding the data. The succinct representation that was chosen is a string that represents each node as a pair of opening and closing parentheses preceded by the value of the node and the succinct representations of the children nodes, enclosed in the parentheses. A depth-first traversal is done through the tree, starting at the root. When a leaf node is reached, its representation is passed back up the tree resulting in the encoded version of the tree. For the example tree shown in Fig. 5, the resulting encoding is: 1(2(4(5())6())7()8()).

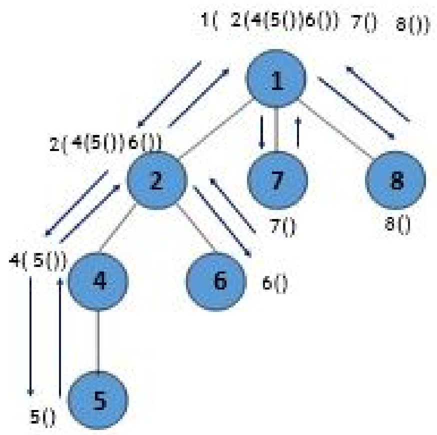


Fig. 5. Example showing the succinct representation used.

The succinct string encoder recursively does a depth-first traversal of the tree returning the encoding of each node. The result is a string, which means that normal string operations can be applied to the representation. This representation takes up less space, compared with the basic string and XML representations of the syntax tree.

4.1 Brute Force Comparison

The brute force algorithm compares the basic string representations of the trees rooted at the given nodes; it returns true if the strings are identical and false if they are not. The string representation is obtained using a method that recursively does a depth-first traversal of the tree rooted at the current node. The depth-first traversal means that if the resulting strings are identical, the trees that are traversed are identical. A `true` result output by this algorithm gives a category 1 result.

4.2 Isomorphism

The succinct string encoder first sorts the children of a node before recursively traversing them. This is done so that the rearranging of statements, which results in the rearranging of nodes in the tree, is ineffective in obscuring the similarity. This sorting is done using a method that compares two nodes by checking if the basic string representing the current node is alphabetically less than, equal to or greater than the string representation of the next node. This means that the children of a node are sorted according to the alphabetical order of the respective string representations.

What is achieved by sorting the children before traversing through them, is that the order of the nodes in the tree is inconsequential. Trees are isomorphic if one tree can be obtained by switching the order of siblings, that is, nodes at the same level. By removing order as a factor, the isomorphism of the trees can be exposed. Given the root nodes of two trees being checked for isomorphic similarity, the algorithm encodes both the trees and checks if the resulting trees are identical. Classifying the trees as being isomorphic means that the two trees contain the same elements structurally and if they do not return true for the brute force algorithm it means that the statements of one program were rearranged to produce the other program.

4.3 Individual Node Comparison

If the programs being compared have not been declared identical or isomorphic, additional comparisons using the succinct representation are carried out. Consider two syntax trees `temp0` and `temp1`, where `temp0` has fewer nodes than `temp1`. First `temp1` is encoded and for each node of `temp0`, the encoding of `temp1` is checked to see if it contains an encoding of the node. If the encoding of one tree contains the encoding of a node in the other tree it means that a similar node has been found in the trees. The system keeps track of all the similar nodes and if the node is a global level node, that is, either a global variable declaration or function declaration, this is also noted.

Based on the percentage of similar nodes that are found, a category 3 or category 4 result is given. A category 3 result is given for similarity greater that 60% (more than 60% of nodes are similar) for either tree. A category 4 result is given for similarity less than 60%. For either category, if a global level node is similar it is reported as such. The threshold percentage between categories 3 and 4 was decided arbitrarily, purely to distinguish the two categories.

5 Experimental Results

The measure of similarity that is used to give the result, is based on four categories as mentioned in Section 4. Listing 1.3 shows an example of the result produced by programs that are structurally the same and return `true` for the brute force algorithm described in Section 4.1. The Parva translator disregards white spaces and comments. When the syntax tree is generated by the parser, identifier names, string literals, character literals and other values are disregarded. This means that making simple visual changes, such as changing identifier names and string literals as well as adding or removing comments or white spaces, does not trick the similarity detection system.

```
Result
Category 1:
Programs are identical
Similarity is 100%
```

Listing 1.3. Output showing a category 1 result.

Listing 1.4 shows an example of the result produced by trees that are isomorphic. This means that the obfuscation technique of rearranging statements does not prevent the similarity check from working correctly.

```
Result
Category 2:
Programs are the same; statements have been switched around
Similarity is 100%
```

Listing 1.4. Output showing a category 2 result.

Additional comparisons using the succinct representation results in either a category 3 result shown in Listing 1.5 or a category 4 result shown in Listing 1.6. Category 3 and 4 results also record the existence of a global level node that is structurally the same, if present.

```
Result
Category 3:
The programs contain large/many similar parts
Program 1 (the first argument) contains: 85.06% similar nodes
Program 2 (the second argument) contains: 52.85% similar nodes
```

Listing 1.5. Output showing a category 3 result.

```
Result
Category 4:
The programs contain few similar parts
At least one global level element (function of variable) is
    the same
Program 1 (the first argument) contains: 32.51% similar nodes
Program 2 (the second argument) contains: 53.70% similar nodes
```

Listing 1.6. Output showing a category 4 result.

5.1 Test Cases

Experiments using three source code files were used to validate the results given by the system. The first file used is `Test0.pav` shown in Listing 1.7. The second file, `Test1.pav` shown in Listing 1.8, was produced by copying the code in the first program and applying obfuscation techniques that rely on semantic meaning such as, changing the type of loop used, expanding variable declarations and assignments and using a chain of redundant functions to call a function. The third file, `Test2.pav` shown in Listing 1.10, was obtained by applying obfuscation techniques that affect the visual appearance of the code, such as changing white spaces and comments, changing variable names and shuffling statements around or hiding blocks of copied code in other functions.

```
void voter()
 {  int votingAge = 18;
  writeLine("Voting age = ", votingAge);
 }

void voter1(){
// voter.pav
// Simple voter example
  voter(); //Write voting age
  const votingAge = 18;
  int age, eligible = 0, total = 0;
  bool allEligible = true;
  int[] voters = new int[100];
  read(age);
  while (age > 0) {
    bool canVote = age > votingAge;
    allEligible = allEligible && canVote;

    if (canVote) {
      voters[eligible] = age;
      eligible = eligible + 1;
      total = total + voters[eligible - 1];
    }
    read(age);
  }
  if (allEligible) write("Everyone was above voting age");
  write(eligible, " voters.  Average age is ", total /
    eligible, "\n");
}

void Main ()
  { voter1(); }
```

Listing 1.7. Test0.pav source code.

```
void voter()
  { writeLine("Voting age = ", 18); }

void voter1(){
// voter.pav
// Simple voter example
  voter(); //Write voting age
  int votingAge = 18;
  int age, eligible = 0, total = 0;
  bool allEligible;
  allEligible = true;
  int[] voters;
  voters = new int[100];
  read(age);
  loop {
    bool canVote = age > votingAge;
    allEligible = allEligible && canVote;
    if (canVote) {
      voters[eligible] = age;
      eligible = eligible + 1;
      total = total + voters[eligible - 1];
    }
    read(age);
    if(age < 0) break;
  }
  if (allEligible) write("Everyone was above voting age");
  write(eligible, " voters.  Average age is ", total /
    eligible, "\n");
}

void calling()
  { voter1();   }

void calling1()
  { calling();  }

void calling2()
  { calling1(); }

void Main()
  { calling(); }
```

Listing 1.8. Test1.pav source code.

The output given when comparing `Test0.pav` and `Test1.pav` is shown in Listing 1.9. `Test0` has 90.05% similar nodes while `Test1` has 82.27% of its nodes similar.

```
Result
Category 3:
The programs contain large/many similar parts
At least one global level element (function of variable) is
    the same
Program 1 (the first argument) contains: 90.05% similar nodes
Program 2 (the second argument) contains: 82.27% similar nodes
```

Listing 1.9. Result of comparing Test0.pav and Test1.pav.

```
void validVoters() {
// validVoters
// Simple voter example
//Descriptions
  const AgeOfVoting = 18;
  int age, eligibleCount = 0, total = 0;
  bool allEligible = true;
  int[] voters = new int[100];
//Descriptions
  writeLine("Voting age = ", AgeOfVoting);  //Output voting
    age
  read(age);
  while (age > 0) { //Descriptions
    bool canVote = age > AgeOfVoting;
    allEligible = allEligible && canVote;
    if (canVote) { //Descriptions
      voters[eligibleCount] = age;
      //Descriptions
      eligibleCount = eligibleCount + 1;
      //Descriptions
      total = total + voters[eligibleCount - 1];
    }
    read(age);
  }
  //Descriptions
  if (allEligible) write("Everyone was above voting age");
  //Descriptions
  write(eligibleCount, " voters.  Average age is ", total /
    eligibleCount, "\n");
}

void Main ()
 {  validVoters(); }
```

Listing 1.10. Test2.pav source code.

Listing 1.11 shows the results when comparing `Test0.pav` and `Test2.pav`. `Test0` has 89.05% similar nodes while `Test2` has 96.24% of its nodes the same. The average of the percentages given indicates that the similarity is higher in the `Test0` and `Test2` comparison than the `Test0` and `Test1` one.

```
Result
Category 3:
The programs contain large/many similar parts
At least one global level element (function of variable) is
    the same
Program 1 (the first argument) contains: 89.05% similar nodes
Program 2 (the second argument) contains: 96.24% similar nodes
```

Listing 1.11. Result of comparing Test0.pav and Test2.pav.

5.2 Discussion of the Results

As in the case for JPlag [6], a result of 100% similarity such as that given in categories 1 and 2 is indicative of plagiarism whereas low percentages of similarity show the absence of plagiarism. For results in between, however, a manual check is required to confirm if plagiarism has taken place. Category 3 and 4 results therefore require manual intervention.

The test cases compare the effect of semantic-based and visual-based obfuscation techniques on the system. The semantic-based techniques seem to be detected by the system but it is possible that some of the similar nodes are detected because of the minimal number of constructs available in the Parva programming language, or due to the simplicity of the example used. The same statements are often chosen because of lack of choice, resulting in unintentional similarity. The percentage of similarity is generally higher for the visual-based techniques; this is most likely because semantic-based techniques change the structure of the program and require the inclusion of semantic meaning in the process of detecting similarity.

6 Conclusion

The aim of this research was to produce a prototype similarity detection system that compares programs for similarity using only the structure of the programs. The system consists of a parsing phase, which outputs a syntax tree represented as XML, and a comparison phase. The XML tree representation is converted into both a string and a succinct representation for use in the second phase.

The usefulness of this prototype system in detecting plagiarism was tested using simple, yet realistic code examples. Results for all four categories of plagiarism were correctly given. Thus, the system can be used both to detect 100% similarity as well as to narrow down the submissions from a large group of students that need to be manually checked.

Future work involves extensive testing on larger code samples and samples taken from real programming languages, as well as optimising the comparison, serialising and encoding algorithms to make the system more efficient. In addition, comparisons with other plagiarism detection systems need to be completed.

References

1. Paris, M.: Source code and text plagiarism detection strategies. In: 4th Annual Conference of the LTSN Centre for Information and Computer Sciences. pp. 74–78. LTSN Centre for Information and Computer Sciences (August 2003)
2. Zhang, F., Wu, D., Liu, P., Zhu, S.: Program logic based software plagiarism detection. In: IEEE 25th International Symposium on Software Reliability Engineering (ISSRE). pp. 66–77. IEEE (2014)
3. Whale, G.: Identification of program similarity in large populations. The Computer Journal **33**(2), 140–146 (January 1990). https://doi.org/10.1093/comjnl/33.2.140
4. Schleimer, S., Wilkerson, D.S., Aiken, A.: Winnowing: Local algorithms for document fingerprinting. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data. pp. 76–85. SIGMOD '03, ACM, New York, NY, USA (2003). https://doi.org/10.1145/872757.872770
5. Wise, M.J.: YAP3: Improved detection of similarities in computer program and other texts. SIGCSE Bulletin **28**(1), 130–134 (1996). https://doi.org/10.1145/236462.236525
6. Prechelt, L., Malpohl, G., Phillippsen, M.: JPlag: Finding plagiarisms among a set of programs. Tech. rep., Karlsruhe Institute of Technology (2000)
7. Puflović, D., Gligorijević, M.F., Stoimenov, L.: CSPlag: A source code plagiarism detection using syntax trees and intermediate language. In: Proceedings of the 52nd International Scientific Conference on Information, Communication and Energy Systems and Technologies (ICEST 2017). pp. 102–105 (2017)
8. Wilson, R.J., Watkins, J.J.: Graphs: an Introductory Approach: a First Course in Discrete Mathematics. John Wiley & Sons Inc (1990)
9. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The Design and Analysis of Computer Algorithms. Addison-Wesley (1974)
10. Itokawa, Y., Wada, M., Ishii, T., Uchida, T.: Tree pattern matching algorithm using a succinct data structure. In: Proceedings of the International MultiConference of Engineers and Computer Scientists. vol. 1, pp. 206–211 (2011)
11. Kao, M.Y., Lam, T.W., Sung, W.K., Ting, H.F.: An even faster and more unifying algorithm for comparing trees via unbalanced bipartite matchings. Journal of Algorithms **40**(2), 212–233 (2001). https://doi.org/https://doi.org/10.1006/jagm.2001.1163
12. Zhang, S., Wang, J.T.: Discovering frequent agreement subtrees from phylogenetic data. IEEE Transactions on Knowledge and Data Engineering **20**(1), 68–82 (2008)
13. Terry, P.: Compiling with C# and Java. Pearson Education (2005)