Means of Computerized Analysis for Detection of Program Code Appropriation

Taras Mohyliak

Software Department, Lviv Polytechnic National University, UKRAINE, Lviv, S. Bandery street 12, E-mail: taras.mogyliak@gmail.com

Abstract – Plagiarism is a common problem for educational institutions. Since it is difficult to detect plagiarized work manually, there is a need of an efficient computerized system, which can detect a theft of program code.

For academic purposes, the best results usually give algorithms, based on tokens comparison. However, they can give false positive results in cases of similar code structure, despite having different domain and purpose.

We propose to introduce weighting coefficients for tokens and calculate code similarity percentage using those coefficients. Our experiment study should confirm the quality improvement of software plagiarism detection.

Key words – code appropriation, software plagiarism detection, token comparison, weighting coefficients, code similarity.

I. Introduction

Text-plagiarism detection algorithms are not efficient for detecting program code theft. Algorithms for software plagiarism detection usually use one of the next approach for comparison: program code strings, tokens, syntax trees, program dependency graphs, numeric metrics. Some of them may combine these approaches in order to receive a better result.

However, the most efficient approach for academic purposes are algorithms based on tokens comparison. The most popular systems, which use such approach, are MOSS [1] and JPlag [2].

Modern integrated development environments give an ability to modify a program code without any problem. A developer can easily rename variables, methods or classes modify or delete existing comments or insert new ones, reorder definitions inside a class or a method. None of these changes has an influence on the execution plan of the software. However, it makes difficulties for detecting a program code theft.

The main purpose of token-based algorithms is to minimize an influence of such minor changes in the code. Instead of comparing original lines of code, the algorithm does the lexical analysis of the text and split it into a sequence of pre-defined tokens. For example, instead of analyzing a variable type and name during the declaration, the algorithm introduces VARIABLE DECLARATION token, which ignores the information, which can be easily modified by the developer. Instead of comparing a for loop with a while loop, the algorithm uses a LOOP token, which is common for every loop type.

Such approach gives an ability to compare a program code, written on different program languages (for example, C# with Java code). Different lexical analyzers use a common set of tokens, which is used during the comparison.

However, the approach of comparing two sets of tokens has its disadvantages. The main problem is that different pieces of program code, which have almost nothing in common, may receive the high percentage of tokens similarity. The vivid example of such issue is the comparison of data transfer objects – DTOs. This type of classes is used only for storing an information and does not do any calculations. Since the token-based algorithms ignore variable and class naming, different data transfer objects will generate the same set of tokens. For example, a class, which contains an information about http response, may be detected as a program code theft from a class, which stores information about an application user.

Another example of token-based solution weakness is a developer’s ability to add extra code, which does not have any impact to the algorithm, to the plagiarized sample. The short piece of code may be much more important for the application rather than a numerous of lines with a useless code. Existing algorithms assume the equality of the importance of every single line of code, unless it is a comment.

To deal with these issues we propose to introduce weighting coefficients for every single token in the analyzed program code. The calculation of the similarity percentage will be based on the weighting coefficient ratio of the matched code to the analyzed program code.

II. Software plagiarism detection

To split a program code to a set of tokens, the system should provide a lexical analysis of the input file. This part of the system is dependent on program language. Every language should have its own parser, which knows how to find specified tokens in the program code. It will use regular expressions to find matching patterns for every token type.

Although this stage is a language-dependent, the result of the parsing process is an array of language-independent tokens. Due to this fact, the system will be able to provide a comparison of program code, written on different languages.

To start with, the C# parser was selected to be implemented firstly. The system architecture, described in Section III, allows expanding the algorithm with new program language lexical analyzers without a need to modify other parts of the system.

The next stage provides a language-independent comparison of tokens, generated on previously stage. First of all, the algorithm in the loop tries to find the longest match between two sets of tokens. As soon as a match was found, the algorithm marks it. In the next iteration, the algorithm finds the longest match within unmarked tokens. The loop continues until the bottom limit of the minimal match is reached.

When the all matches between program code A and B are found, the algorithm calculates the similarity percentage sim(A, B) using the Eq. 1, where m is a number of matches, L is a total number of tokens, w is a weighting coefficient:

\[ sim(A, B) = \frac{\sum_{i=1}^{m} \text{length}_i \cdot \text{sum}(w)}{\sum_{j=0}^{L} w_j} \]  

(1)
As a result, the long match with small coefficients will have a lower impact on the results than a short match with high weighting coefficients.

The usage of the weighting coefficients allows system to be stable in case of different program code modifications. The result of the calculation will not change in case of formatting changes (adding whitespaces or tabulation), inserting or deleting of comments, variable, method or class renaming, constant values modification, replacing loops or conditions with their equivalents, reordering of variable declarations, adding a useless blocks of code, changing of access modifiers etc.

In the Table I, there is a set of weighting coefficients for the most frequently used tokens.

TABLE 1

| ASSIGNMENT       | 4 |
| VAR_DECLARATION  | 3 |
| INVOKE           | 3 |
| BEGIN_LOOP       | 4 |
| COMMENT          | 0 |
| BEGIN_TRY        | 5 |
| THROW_EXCEPTION  | 7 |
| BEGIN_CLASS      | 1 |
| NAMESPACE        | 0 |
| BEGIN_ENUM       | 2 |
| BEGIN_CRITICAL_SECTION | 8 |

III. System architecture

The system is developed as a .NET plugin-based library. It contains two independent components – lexical analyzer and tokens comparer. Each of these components is plugin-based. It gives an opportunity to easily add new program language parser or a comparer with new algorithm without a need to recompile the entire library. Having the access to the abstract interface for lexical analyzers or token comparers, independent developers may add their own implementations for detecting software plagiarism.

The abstract layer contains the set of tokens with predefined weighting coefficients and the interfaces for common lexical analyzer and comparer behavior.

Lexical analyzer abstraction contains the description of the program language (name, file extensions) and the interface for implementation of program code parser. In addition, it contains the program language configuration for parser to find matches within the lines of code.

Comparer abstraction contains the interface for the comparison algorithm implementation, which receives a set of tokens and returns the similarity value.

Conclusion

Token-based software plagiarism detection algorithms are efficient for academic purposes. In this paper, we provided an investigation of existing systems, their strengths and weaknesses.

Introducing weighting coefficients for tokens can improve the quality of program code theft detection. Based on the program code importance for the result of the program execution, these weighting coefficients should allow to detect program code theft in case of adding a useless code more precisely. In addition, it should minimize the number of false positive results when detecting non-important code as a plagiarism.

Described system may be used for multiple purposes in educational institutions, for example, it can be used in the distance education portal for detecting software plagiarism.

References

[5] Chao Liu, Chen Chen, Jiawei Han and Philip S. Yu. GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis. In the Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD’06), pp. 872-881, Philadelphia, USA, August 2006.