Original scientific paper

# Performance Evaluation of Plagiarism Detection Method Based on the Intermediate Language

Vedran Juričić
Department of Information Sciences,
Faculty of Humanities and Social Sciences, University of Zagreb
Ivana Lučića 3, Zagreb, Croatia
vjuricic@ffzg.hr

Tereza Jurić
Department of Information Sciences,
Faculty of Humanities and Social Sciences, University of Zagreb
Ivana Lučića 3, Zagreb, Croatia
tjuric2@ffzg.hr

Marija Tkalec
Department of Information Sciences,
Faculty of Humanities and Social Sciences, University of Zagreb
Ivana Lučića 3, Zagreb, Croatia
mtkalec@ffzg.hr

**Summary**

*This paper presents detection method for source code plagiarism that is based on the intermediate language, and shows its usage in e-learning. Method is tested on the appropriate number of test cases that represent the most frequent code modification techniques. Results and its performance are compared to the existing source code plagiarism detection methods implemented in some of the most known plagiarism detection systems and applications.*

**Keywords**: plagiarism detection, performance, intermediate language

## Introduction

Plagiarism is the act of reproducing, or reusing, someone else's work without acknowledging the source [6]. Academic community deals with a huge problem of plagiarism detection, therefore, in order to protect authorship, many algorithms and strategies have been developed. Source code plagiarism detection represents a big problem in educational courses especially in computer science where programming is the main field of work.
Source code is considered plagiarism even if the code does not contain exactly the same elements of someone else's code. Nowadays it's very easy to copy

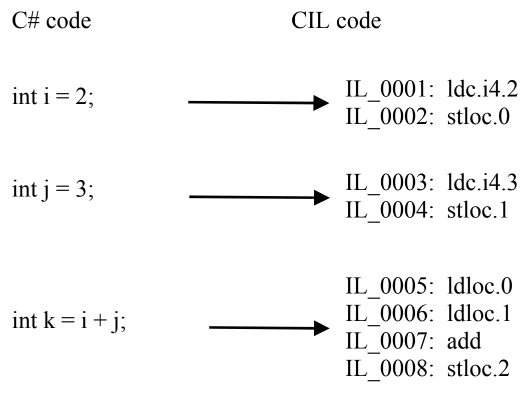
someone else’s code from the Internet, whether a function, an algorithm or a complete application. Transformations can be simple, like changing the variable names, modifying comments, and complex, like adding new functions, replacing code structures with equivalents, etc.
This paper deals with the analysis of plagiarism detection systems whose purpose is to detect unoriginal source code in order to maintain copyright infringement.
In this research, three plagiarism detection systems were used. The research was implemented on a group of test cases written in C# programming language. Obtained results were analyzed and compared with the results of algorithm proposed by authors, in order to evaluate the performance of the used systems.

## Plagiarism detection method

This paper proposes a method for source code similarity analysis, for .Net programming languages. Method does not analyze the original source code but instead, it analyzes low-level language, called intermediate language.

### Intermediate language

All .Net languages are generally compiled twice before finally executed on the operating system. First compiler is language specific, and compiles source code to low-level language called CIL [5] (Common Intermediate Language). For example, code written in C# language is compiled to CIL using C# compiler. CIL is a processor and platform-independent instruction set that can be executed in any environment that supports the Common Language Infrastructure, such as .Net runtime on Windows or cross-platform Mono runtime. CIL code is, upon execution, compiled for the second time using JIT (Just-In-Time) compiler, which generates platform or processor-specific binary code, also known as native code.



```
C# code                 CIL code

int i = 2;      ──►  IL_0001: ldc.i4.2
                     IL_0002: stloc.0

int j = 3;      ──►  IL_0003: ldc.i4.3
                     IL_0004: stloc.1

                     IL_0005: ldloc.0
int k = i + j;  ──►  IL_0006: ldloc.1
                     IL_0007: add
                     IL_0008: stloc.2
```

Figure 1. C# to CIL mapping

Intermediate language code [9] is the lowest-level human-readable programming language, and, therefore, it has lesser commands and simpler structure than third-generation languages like C#. On the other hand, one command of high-level language usually maps to many intermediate language instructions.

**Analysis phase**

Initial step in similarity analysis is recursive pass through the file system from the specified root folder in order to find all the source files that it contains. It is also possible to specify certain search filters, like allowed extensions and file name pattern.

When initial step completes, all files in the same folder (immediate parent, not the top level) are given to the C# compiler that generates one assembly file per the given folder. Although other methods are possible, application calls the compiler as an external process, and gives specific parameters, like source files and destination file name, via command line interface.

The language of generated assembly is the intermediate language, but its format is not readable and not suitable for analysis, so it must be converted in the text format. This is the third step; a process called disassembling, which is realized in a similar way as previous step, by calling external process and executing ILDasm.exe. ILDasm.exe is a tool that is included in installation of Visual Studio or Framework SDK, which loads an assembly containing intermediate language code and generates a text file.

This text file, generated in the third step, is the actual input file for a comparison. It contains certain lines of code that are not relevant for analysis: metadata and module information, comments (generated by ildasm, not developer), stack related data, etc. Those lines are removed in the fourth, preprocessing step, and not included in the further analysis.

Text file containing disassembler code is parsed line by line and then it is verified if they satisfy predefined patterns. Generally, only lines starting with *IL_* contain intermediate language instructions. The example is shown in Fig. 2 which shows the second instruction (*IL_0002*) inside the observed method, that takes value from the stack and stores it (*stloc*) in the local variable 0 (*.0*).

| | | |
|---|---|---|
| IL_0002: | stloc | .0 |
| line position | Instruction | variable 0 |

Figure 2. CIL instruction

The number that follows *IL_* is the position of an instruction inside a method or a property. The exact position of line (instruction) is not relevant for the comparison; what matters is that it exists and occurs, and therefore the whole *IL* part is removed from the line (e.g. *IL_0002*). In the above example, Fig. 2, last number is the index of local variable. Index is based on the position in the original (high-level language) source code, where the variable is introduced for the first time. This number is also removed from the line, because the order in which appear the variables in the original source code is not relevant for comparison. Also, only variable existence is important. After the fourth step, instruction from Fig. 2 is reduced only to *stloc*.

### Comparison phase

When preprocessing phase is completed for all the input assemblies, resulting instruction sets are compared; each processed set is compared to all others sets, and the result is stored in a matrix.

Each line in instruction set can contain one or more elements, but it is taken and compared as one unit, that is, one string. There are numerous methods and algorithms that can be used for string comparison and for calculating similarity between two strings [3], which differ in complexity, calculation time, drawbacks, etc.

The proposed method uses a Greedy String Tilling algorithm [15], which is implemented and used in many of today's plagiarism detection systems. It has worst case complexity O($n^3$), but with running Karp-Rabin matching has an experimentally derived average complexity close to linear [15].

## Plagiarism detection systems

The following section describes three plagiarism detection systems that were analyzed: MOSS, JPlag and CodeMatch. Their performance was compared with the performance of the algorithm proposed by authors, in order to evaluate the best results obtained by these systems.

### MOSS

MOSS (Measure of Software Similarity) is a plagiarism detection software tool developed by Alex Aiken in 1994. MOSS is commonly used in computer science faculties and many other engineering courses. It is provided as a free Internet service hosted by Stanford University and it can be used only if a user creates an account. Files are submitted through the command line and the processing is performed on the Internet server. The current form of a program is available only for UNIX systems.

The program can analyze source code written in 26 programming languages including C, C++, Java, C#, Python, Pascal, Visual Basic, Perl etc. Comparison can be done only between source code files, comparing text files to determine plagiarism between them cannot be done.

MOSS uses Winnowing algorithm based on code-sequence matching and it analyses the syntax or the structure of the observed files.

MOSS maintains a database that stores an internal representation of programs and then looks for similarities between them [10].

The obtained result is displayed in a form of HTML pages or simply in a textual form representing pairs of programs with similar code in an ordered list.

**JPlag**

JPlag is a free plagiarism detection tool used to detect software plagiarism among multiple sets of source code files. It is commonly used in programming education for detecting unallowed copying of student exercise programs, but it can also be used for detecting stolen software parts among large amounts of source text or modules. JPlag was developed in 1996 by Guido Malpohl and it currently supports C, C++, C#, Java, Scheme and natural language text. Program is available through an installation-free Java Web Start client.

JPlag uses Greedy String Tiling algorithm which produces matches ranked by average and maximum similarity. Average similarity is an average of both program coverages and is the default similarity. If it is big, it indicates that observed programs are working in a very similar way. Maximum similarity is the maximum of both program coverages. It is used to compare programs which have a large variation in size which is probably the result of inserting a dead code into the program to disguise the origin.

Obtained results are displayed as a set of HTML pages in a form of a histogram which presents the statistics for analyzed files.

**CodeMatch**

CodeMatch is the commercial software included in a CodeSuite collection of analysis tools produced in 2003 by Bob Zeidman and under the licence of a SAFE Corporation. The program is available as a standalone application. It has a free version which allows only one trial comparison where the total of all files being examined doesn't exceed the amount of 1 megabyte of data. The program supports 26 different programming languages including C, C++, C#, Delphi, Flash ActionScript, Java, JavaScript, SQL etc. CodeMatch is mostly used as forensic software in copyright infringement cases exclusively used for source code plagiarism detection.

CodeMatch determines the most highly correlated files placed in multiple directories and subdirectories by comparing their source code. Four types of matching algorithms are used: Statement Matching, Comment Matching, Instruction Sequence Matching and Identifier Matching. These algorithms produce the ranked CodeMatch score which is a combination of all weights given to an each file.

The results come in a form of HTML basic report that lists the most highly correlated pairs of files.

**Performance evaluation**

Test cases used in this research were written in C# programming language and all of them were created by the authors. Test cases were placed in 6 different

categories and their total number is 50. Categories were constructed considering variable names, types, properties, methods and classes.
All the variable test categories included checking the behaviour of the algorithm when the variable names, types, assigned constant values and location of their declaration varied. Various property definition styles, their type, name and returning values were tested.
Although it tests some of the most common variations with variables, including usage of various existing methods for converting one variable type to another, the syntax itself contains a relatively small number of test cases. Cases that test changing method name, returning type and various parameter reordering, insertions and deletions are also significant parts of the method in question.
The loops category contains test cases that check various loop replacements and definitions while the class category deals with cases that test changing class name, namespace, and reordering and renaming of class members.
Authors conducted manual inspection of all test cases used in the research. The results of the aforementioned manual comparison are shown in a 50x50 matrix the rows and columns of which are correspondent to the test cases used. Each cell represents the value between two test cases. This matrix represents a reference matrix to which all results obtained by plagiarism detection systems are compared.
The two used evaluation methods were precision and recall including their harmonic mean, the F measure. Those methods evaluate the algorithms' behaviour and sensitivity to various code modification techniques. Precision is defined as a fraction of correctly categorized test cases divided by the number of test cases claimed to be similar [12]. Recall is defined as fraction of correctly categorized test cases divided by the number of test cases manually categorized as similar [12]. F measure is defined as a harmonic mean of precision and recall, so that both measures are equally represented [12].

## Results

The reference matrix contains only values one and zero, where one indicates that the similarity between test cases is relevant and that they should be treated as similar or equal. Similarity matrices obtained by the plagiarism detection systems contain decimal values in range from zero to one, so they were converted to the suitable values in order to analyse performance. Conversion is based on the threshold, so that all values above threshold are converted to one; otherwise, they are converted to zero.
Authors tested the relation between a threshold value and the calculated precision, recall and F measure at this threshold, which enabled the authors to identify the best values and performance for each plagiarism detection system. Results for each system are presented in the graphs below. Graphs display precision (p), recall (r) and F measure (F) in relation to threshold which is displayed on the horizontal axis.

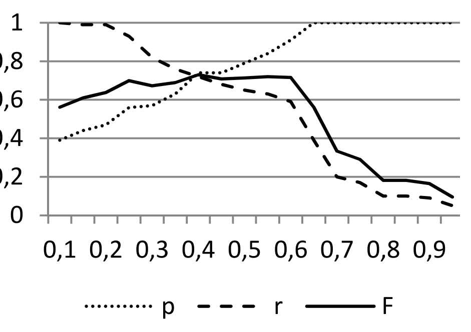


Figure 3 - MOSS results

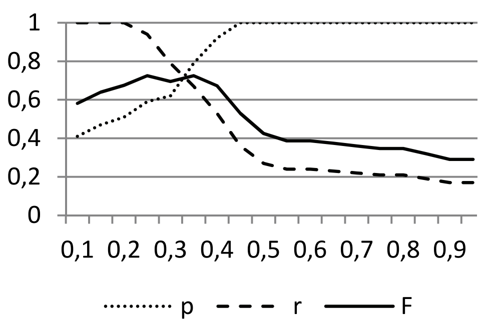


Figure 4 - JPlag results

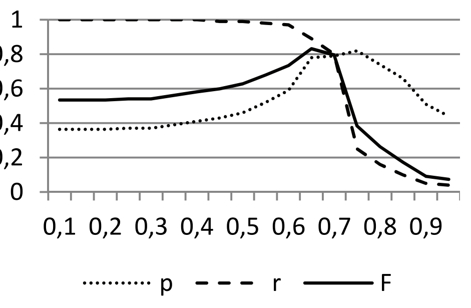


Figure 5 - CodeMatch results

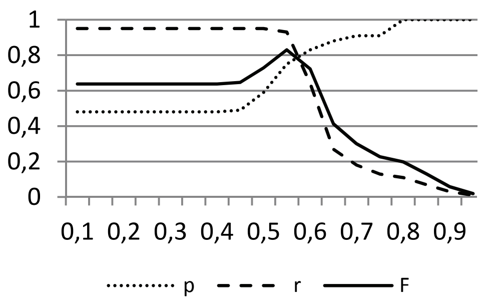


Figure 6 - ILMatch results

As it is expected, all plagiarism detection systems have very high recall when the threshold is low, and it decreases as the threshold increases. ILMatch is the only system that does not reach recall of 100%; its highest value is 95%. On the other hand, precision rises with a threshold, and reaches 100% on the high threshold, except for the CodeMatch system, whose maximal precision is 81%.
The best identified F-measures for tested plagiarism detection systems are shown in the Fig. 7. Performance of MOSS and JPlag is almost the same: the best F-measure for those systems is about 73%, while CodeMatch and ILMatch show the best performance, their best F-measure is about 85%.
By analyzing ILMatch behaviour on individual test cases, the authors concluded that changes in comments do not have impact on similarity, because user comments do not appear in intermediate language code. Also, because source code is not analyzed, modifications of code formatting have no impact on similarity. Modifications to intermediate language code that are made in preprocessing phase, ensure that these transformations do not affect the results of comparison: modifications of variable and class names, changing names of class members, changing data type of variables and constants and changing values of constants.

Replacing expressions and loops with equivalents and changing the structure of selection statements has slight impact on comparison results. Rewriting code in different programming language also has little impact similarity. Transformations that can cause significant differences in calculating similarity are reordering operands in expressions, changing the order of class members, changing the order of statements and adding redundant statements and variables.

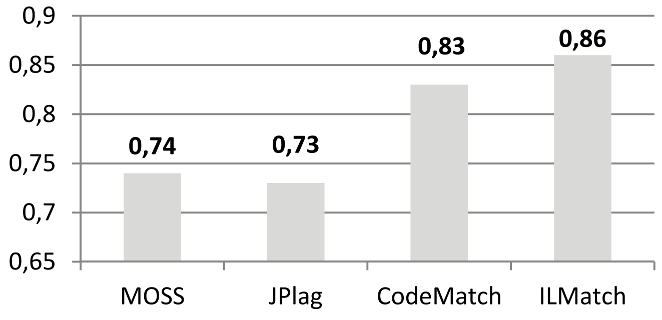


Figure 7 - Highest F-measures

## Conclusion

This paper presented a plagiarism detection method based on the intermediate language. The system based on the proposed method was compared with the most used plagiarism detection systems that were available to the authors and that supported the language the test cases were written with: MOSS, JPlag and CodeMatch.

Test cases were designed and written so that they analyse systems’ behaviour under different types of code modifications, which are commonly used to mask code reuse and copying. By analysing their performance, authors determined that the system based on the intermediate language showed the best results, that is, it had the best F-measure.

## References

[1] Bowyer K.W, Hall L.O. Experience Using ”MOSS” to Detect Cheating On Programming Assignments. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.73.7062&rep=rep1&type=pdf

[2] Chapman S. SimMetrics, Open source library of Similarity Metrics. 2006. http://staffwww.dcs.shef.ac.uk/people/S.Chapman/stringmetrics.html

[3] Clough P, Plagiarism in natural and programming languages: an overview of current tools and technologies. 2000. ftp://www.dlsi.ua.es/people/armando/maria/Plagiarism.rtf

[4] Cohen W.W, Ravikumar P, Fienberg S.E. A Comparison of String Distance Metrics for Name-Matching Tasks. http://www.cs.cmu.edu/~wcohen/postscript/ijcai-ws-2003.pdf

[5] Common Intermediate Language. 17.12.2010. http://en.wikipedia.org/wiki/ Common_Intermediate_Language
[6] Goel S, Rao D et al. Plagiarism and its Detection in Programming Languages. http://www.stanford.edu/~drao/Resources/Plagiarism.pdf
[7] JPlag Detecting Software Plagiarism. 16.6.2010. https://www.ipd.uni-karlsruhe.de/jplag/
[8] Kwan R, Fox R, Chan F.T. Enchancing learning through technology: research on emerging technologies and pedagogies. World Scientific Publishing Co. Pte. Ltd. Singapore, 2008.
[9] List of CIL Instructions. 17.12.2010. http://en.wikipedia.org/wiki/List_of_CIL_instructions
[10] Plagiarism Detection. 20.6.2011. http://theory.stanford.edu/~aiken/moss/
[11] Prechelt L, Malpohl G, Philippsen M. Finding Plagiarisms among a Set of Programs with JPlag. Journal of Universal Computer Science; 2002. http://pswt.informatik.uni-erlangen.de/EN/ publication/download/jplag.pdf
[12] Precision and recall. 15.6.2011. http://en.wikipedia.org/wiki/Precision_and_recall
[13] S.A.F.E. Software Analysis & Forensic Engineering Corporation. CodeSuite Version 4.2 User's Guide. http://www.safe-corp.biz/documents/CodeSuite_Users_Guide.pdf
[14] Schleimer S, Wilkerson D.S, Aiken, A. Winnowing: Local Algorithms for Document Fingerprinting; 2003. theory.stanford.edu/~aiken/publications/papers/sigmod03.pdf
[15] Wise M.J. String Similarity via Greedy String Tiling and Running Karp−Rabin Matching. 1993. http://vernix.org/marcel/share/RKR_GST.ps