Source code plagiarism detection: The Unix way

Juraj Petrík*, Daniela Chudá*, Branislav Steinmüller*
*Faculty of Informatics and Information Technologies, Slovak University of Technology in Bratislava, Ilkovičova 2, 842 16, Bratislava, Slovakia
juraj.petrik@stuba.sk, daniela.chuda@stuba.sk, branislav.steinmuller@stuba.sk

Abstract— **The paper describes similarity detection method for language independent source code similarity detection. It is based on idea of maximum reusability of standard Unix filters. This method was implemented and benchmarked with different datasets from real world (students' assignments) and also synthetic datasets (perfect plagiarism experiment). Our method achieved significantly better results than competitors, which are considered as gold standard in plagiarism detection.**

I. INTRODUCTION

When we are talking about source code similarity, we are talking about areas such as refactoring support, pattern search, malware detection or software plagiarism detection. Due to a lot of source code is produced every day, it is becoming more and more essential to have supporting software for different tasks in terms of similarity.

Software plagiarism has different levels – exact copy from source, copy of context or even copy of idea for example [9]. This kind of situation (plagiarizing) is not desirable in learning process in schools, but unfortunately plagiarism is common in academic sphere.

In 2010 questionnaire was done at Slovak University of Technology in Bratislava. Students were asked if they have ever plagiarized at the faculty - 33% percent of students admitted plagiarism. Another interesting question was, if students have ever given their work for others to plagiarize – 63% of students admitted, that they have given their work to plagiarize [5]. These numbers are pretty high, so it is really important to deal with this phenomenon.

Common practice on programming courses is usage of some kind of similarity checking. This checking can be done by course teacher and (or) automatically by software tool. There are couple of methods for similarity measurement – none of them is good all-rounder [2]. Additionally, students are using obfuscation techniques to hide plagiarism [7]. These obfuscations can be done in many different ways. For example, there are simple methods such as variables renaming, function names renaming or comments modifications. There are also more advanced ones – functions in lining or functions splitting for example. And also some expert methods, such as adding of redundant source code or slightly changing the functionality of the program [4]. Source codes and therefore programming assignments are written in many different programming languages and when we are stealing ideas it is simple to rewrite code from one language to another, so we have need to develop methods which are language independent.

Software plagiarism is not problem only in academic area, it is trouble in commercial area too, for example case such as Google vs Oracle shows that software plagiarism is even worse than we thought. Google was not found guilty by violating any of Oracle's patents, but it was clear that Google had plagiarized parts of Oracle's source code [10].

We are aiming at software plagiarism detection, but proposed method is also usable for source code refactoring or patterns searching. This paper is divided into several parts. First part describes mainly source code clone detection tools and related work done in this topic. Second part is dealing with our method inspired by standard Unix filters. Third part is trying to set up fair conditions for benchmarking – describing datasets used for plagiarism detection and methods for benchmarking. Fourth part is describing acquired results of benchmarking done according to previous chapter and in last chapter we are discussing about results and future of this project.

II. RELATED WORK

Over recent decades, at universities, multiple tools implementing various methods have been proposed and implemented in order to check students' assignments. Therefore we are here analyzing freely available plagiarism detection systems, which are considered as standards in this area [8]. The tools were chosen by following criteria: if the tool is available, if the tool supports Java and C language and if the tool is well documented, also we were looking at options for batch running of these programs [13].

Moss is shortcut of Measure of Software Similarity. This tool is provided as Internet service and requires registration via email to be able to submit source code files for automatic assessment. Moss is based at Stanford University and is being developed from 1994. It supports multiple programming languages – for instance C, C++, Java, Pascal, Perl, Python [3]. It is using tokenization for source code pre-processing and after that is used winnowing algorithm [14]. Our experience with this tool is mostly positive – it has decent detection rate of plagiarism, it is pretty fast and has good visual output. On the other hand, it does not support console output of results and results needs to be parsed from html pages for automated processing.

JPlag is developed by University in Karlsruhe (Germany) and is also token based like Moss, but for final comparison is using Greedy String Tilling algorithm. It supports C-like languages – Java, C#, C, C++ and also supports natural language text documents. Working with this tool was most satisfying amongst all tools in this chapter – it is multiplatform standalone console application, which has html and also console output. Also detection rate is pretty good [11].

Plaggie is standalone source code clone detection software for java programs in Java programming language. Plaggie was developed as standalone open-source

978-1-5090-5655-2/17/$31.00 ©2017 IEEE

alternative to JPlag. Nowadays JPlag is standalone open-source application too, but in past it was not [1].

SIM is token based tool too. It supports languages like Java, C, Pascal, Modula-2 Miranda and List. It is being developed from year 1989 at University of Amsterdam. This tool is using dynamic programming string alignment for token comparison. Unfortunately, this technique is performance hungry, so for larger datasets it can take a while to get results [6]. From our experience this tool has quite good results, it is also console standalone application, but for windows platform only and it is also not open-source software. Also there is lack of pretty visualization of similar pairs.

Simian is tool for identifying duplicates in programming codes and also in plain text files. Simian is not intended for plagiarism detection, but for fast search for code duplicates to support refactoring of big projects. It is extremely fast and memory efficient – it can find duplicates in 390 309 lines of source code in less than 10 seconds using only 48 MiB of memory. This tool is not designed as plagiarism detection system, so we are using this tool as comparison with real plagiarism detection tools – to see if plagiarism detection tools are better than just plain code clone detector software.

SIDplag is tool developed at Slovak University of Technology in Bratislava. It is standalone Java application with graphical user interface. It supports C and Java language, but can be easily extended to any language via special configuration files. The tool is using tokenization as pre-processing method and Greedy String Tiling for comparison of these tokens. Unique property of this tool is usage of synonyms for tokens [4].

III. OUR METHOD

The philosophy of our method and plagiarism detection tool is to maximize usage of standard Unix filters (for text processing) and to enable easy and scalable parallelization of the solution. Big advantage of using this approach is programming language independence, which is reached by using techniques described in this chapter, hence we do not need to customize tool for every programming language which we want to test. The tool itself consists of 2 parts - assignments checker and coefficients normalizer.

A. Assignments checker

This part is comparing all source code assignments as pairs of files – this means there is total of $\frac{n^2}{2}$ comparisons done. We are utilizing multiple levels of comparisons between each pair of checked source code files. These levels are representing set of checks for plagiarism detections from simple to more sophisticated ones. The purpose of this levels is to progressively and transparently discover different types of source code obfuscations. The transparency of this approach is achieved by using standard well known tools in scripting language using pipes (tools chaining).

There are five levels of pre-processing:

First level is simple pair diff of two source code files (pairs) (diff tool used)

Second level is pair diff with empty lines excluded with case of characters, whitespaces and blank lines ignored (grep and awk tools used).

Third level is same as second level. Additionally, all alphanumerical character sequences are replaced by same character ('X' character for example – tr tool used). This technique gives us skeleton of the tested program (structure of the program).

Fourth level is same as second level with some upgrades. These upgrades are as follows: comments are completely removed and there is one word per line – so we are doing word based diff in fact.

Fifth level is same as fourth level, plus there is added alphabetical sorting of pre-processed statements (sort tool used).

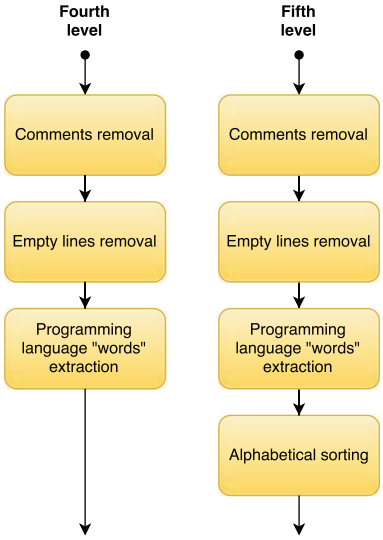

Figure 1. Fourth level and fifth level pre-processing

Figure 1 shows fourth and fifth level of pre-processing. This figure emphasizes reusability of components between different levels.

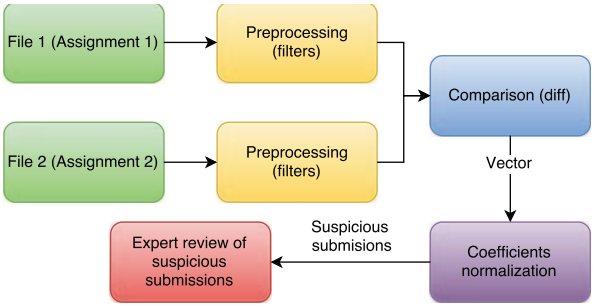

Figure 2. Overall look at submissions processing for one level

Figure 2 describes how the whole process of plagiarism detection is done using this tool.

$$V=(L_1, L_2, C_1, C_2, C_T, N_C) \quad (1)$$

Output of this stage is vector consisting of six numerical values, this vector is specified in equation 1. For each level of every pair there is one vector. Another output is information for each vector - if the pair is suspicious. We are using three suspect levels (ordered from least suspicious to the most suspicious), these levels are based on heuristic observations done on assignments assessments:

If $N_C < L_1$ or $N_C < L_2$ – suspicious ones.
If $N_C < L_1$ and $N_C < L_2$ – the more suspicious ones.
If C_1 or $C_2 < \frac{L_1}{4}$ or $\frac{L_2}{4}$ – the most suspicious ones.

Where L_1 and L_2 are total number of lines produced by specific level of pre-processing for file one and file two. C_1 and C_2 represent number of changes (in lines) need to be done to file one or file two to make these files identical. C_T is $C_1 + C_2$, that means it represents total number of changed lines in both files. Next and not commonly used property in other plagiarism detection systems is last part of vector N_C, which is representing total number of changes, which need to be done to make the files identical. Here one change is longest sequence of similar lines (diff like).

The similarity in this tool is defined as asymmetric relation between two files – what means that similarity percentage between A and B is number of lines/characters which needs to be changed in file A to be identical with file B and vice versa.

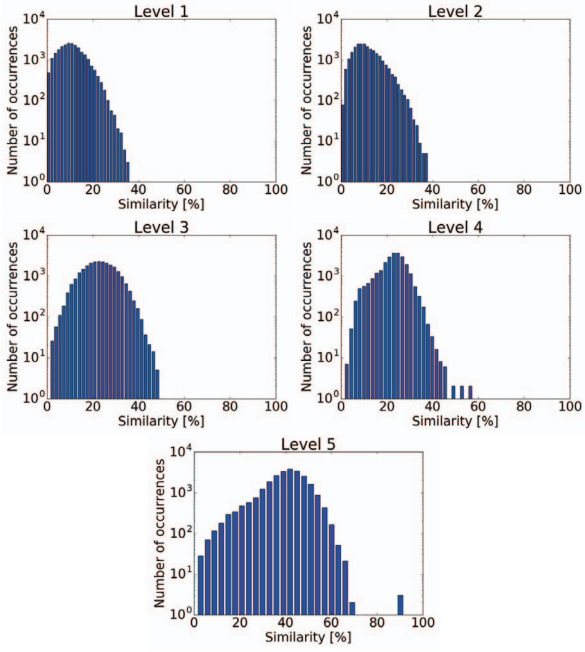

Figure 3. Histograms of similarity percentage for different levels of processing (be aware of logarithmic scale on y axis) - see equation 2

$$S_i = \frac{L_1 + L_2 - C_t}{L_1 + L_2} * 100 \quad (2)$$

Figure 3 represents average similarity percentage amongst all pairs in sample dataset of students' assignments (this dataset is described in next chapter). Here is clearly visible, that higher the level is, the more similarity in documents is revealed (level 4 and level 5 in this figure).

Figure 4 illustrates average size per change per each level for each pair. Big average change size per submission means, that a small amount of big changes (continues sequences of lines/words) needs to be done to have two identical files. Small average change size usually means, that a lot of small changes needs to be done to have two identical files. So we need to find balance between these two extremes to reliably find plagiarism copies.

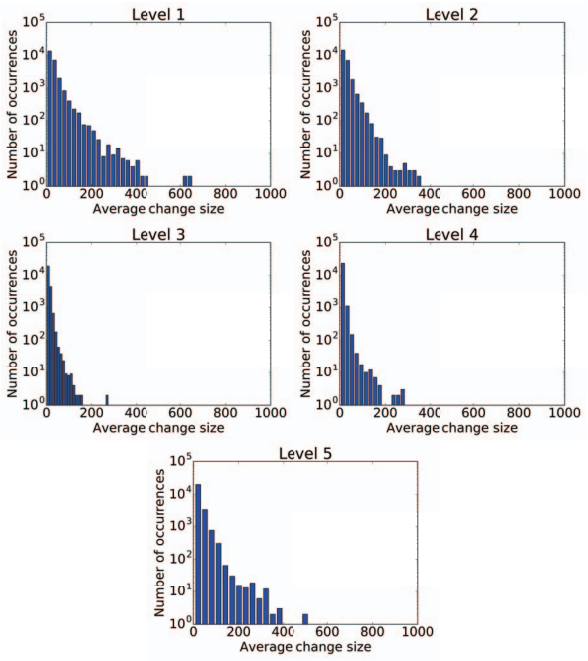

Figure 4. Histograms of average change size $\left(\frac{C_T}{N_c}\right)$ for different levels

B. *Coefficients normalizer*

This part is adjusting results of previous part for specific study course. The idea of doing this comes from that every course has some specific types of assignments each year, so statistically each year's submissions should have similar distribution of similarity percentages amongst levels. Utilising this knowledge, we can further improve detection rate.

For each level we got coefficient value, which is defined heuristically by experts. We are applying these coefficients on results of previous steps and checking whatever the results are still suspicious.

IV. BENCHMARKING METHODOLOGY

Truly objective benchmarking of plagiarism detection tools is hard, maybe impossible at the moment - because even human experts are not in agreement with each other, when they are evaluating same source code [12].

Therefore, we have chosen multiple techniques and datasets (synthetic and real world) for the evaluation.

1) *Refactoring dataset*

This dataset consists of 45 files in java programming language. This dataset is focused on testing tools resistance from intentional obfuscations (plagiarism detection attacks). It was created manually, from students submitted assignments and contains these types of obfuscation modifications:

Copy – this is just plain copy; it is used for comparing what is the similarity of the exactly same files

Comments – modifications such as deleting comments, reordering of comments, changing content of comments and so on.

Print – changing output text/format of the program

Variables – renaming and types changes of variables

Structure – if instead of switch and for instead of while and vice versa

Block sequences – changing order of blocks

General – all above

Guru – all above plus change of functionality and adding of redundant code

2) Perfect plagiarism dataset

This dataset is taken from experiment of creating "perfect" plagiarism – the aim of this experiment was to create such a plagiarism copy, that none of tested tools would mark the obfuscated assignment as clone of original copy. This dataset was generated by expert simulating, that he does not know what the program is doing (to simulate steps, which can be done by automatic plagiarism tool). It is red-black tree implementation in Java programming language.

The dataset consists of 3 parts (levels), according to level of knowledge required to do these kind of obfuscations:

Basic level - just basic changes were done to the source code:

- Code formatting
- Comments removal
- Renaming of classes, methods and variables

Advanced level - first level changes were done, plus some advanced changes:

- Loop changes (for->while, while->for…)
- Added new constants
- Negation of conditions
- Variable types
- Line ordering

Expert level – second level changes were done, plus some more advanced changes:

- Parameters order
- Variable modifiers
- Wrapping of return values
- Merging of some methods
- Splitting of some methods
- Students submissions dataset

3) Students' submissions dataset

This dataset is real world set of assignments from students from data structures and algorithms course. The dataset consists of 223 student's assignments written in C language. Students task was to implement memory manager simulator. The average number of lines per file (assignment) is 120 lines.

V. Results

For benchmarking purposes, we have chosen command line plagiarism detection systems - because of easier batch processing and evaluating, benchmarking multiple files datasets with GUI only tools is quite uncomfortable. When testing our tool we do not apply second part of the tool (coefficients normalizer), because this part is designed for usage in real world assignments from study courses, not for synthetic tests.

On the other hand, results from perfect plagiarism dataset are interesting – see Figure 5. Basic and advanced level changes are revealed by all of benchmarked tools. However, expert level changes are real problem for all tested tools. Similarity percentage of our solution is 20 percent for level 5 pre-processing, other tools achieved about 5 percent similarity value, which is below threshold for being suspicious. Even our solution does not find high percentage of similarity, but we need to realize, that our solution is not using only similarity percentage for marking files as suspicious, but it is using other parameters too. Our solution marked even expert level changes as more suspicious – so human expert would check this files manually and find that they are plagiarism copies

VI. Conclusions and Future Work

Our approach achieved better similarity detection sensitivity for expert level obfuscations in source codes than other benchmarked tools, which are commonly used for plagiarism detection at universities.

This tool is also programming language independent, so it can be easily used even for exotic languages, which are commonly not supported by common similarity (plagiarism) detection software.

But we need to realize the fact, that all of these tools and methods (our too) are used only as support software for human experts, because final decision has to be done by expert – he has the responsibility for marking assignment as plagiarism or not. Because in some cases similar parts of assignments are occurring naturally (some well-known problem – pseudo codes).

Also this tool needs enhancements such as automatic determination of threshold values for marking something as suspicious or automatic generation of normalization vectors for each course.

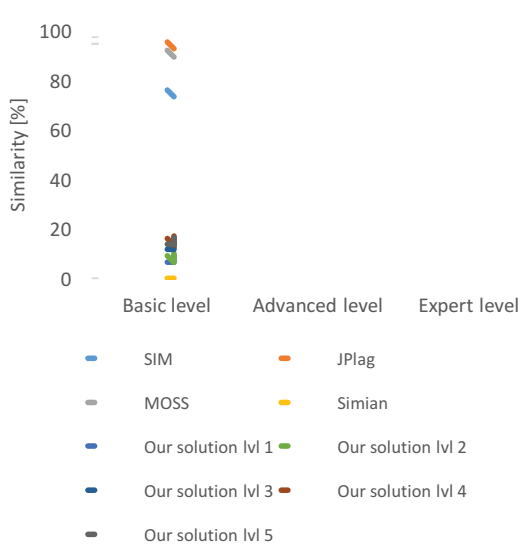

Figure 5. Perfect plagiarism dataset benchmark results

Unfortunately, in area of similarity and also plagiarism detection tools there is problem - lack of standard all-around large enough datasets. The consequence of this unhappy situation is that (not only) we cannot compare the results between multiple papers with other methods.

References

[1] a. Ahtiainen, S. Surakka, and M. Rahikainen, "Plaggie: GNU-licensed source code plagiarism detection engine for Java exercises," *Proc. 6th Balt. Sea Conf. Comput. Educ. Res. Koli Call. 2006*, pp. 141–142, 2006.

[2] B. Beth, "A Comparison of Similarity Techniques for Detecting Source Code Plagiarism," 2014.

[3] K. W. Bowyer and L. O. Hall, "Experience using MOSS to detect cheating onprogramming assignments," *Front. Educ. Conf. 1999 FIE99 29th Annu.*, vol. 3, no. Piscataway, NJ, United States, pp. 13–18, 1999.

[4] D. Chudá and B. Kováčová, "Checking plagiarism in e-learning," *Proc. 11th Int. Conf. Comput. Syst. Technol. Work. PhD Students Comput. Int. Conf. Comput. Syst. Technol. - CompSysTech '10*, vol.

[5] D. Chuda, P. Navrat, B. Kovacova, and P. Humay, "The issue of (software) plagiarism: A student view," *IEEE Trans. Educ.*, vol. 55, no. 1, pp. 22–28, 2012.

[6] Z. Duric and D. Gasevic, "A Source Code Similarity System for Plagiarism Detection," *Comput. J.*, vol. 56, no. 1, pp. 70–86, 2013.

[7] A. C. Juan, "Studying the Impact of Obfuscation on Source Code Plagiarism Detection," no. January, pp. 1–39, 2014.

[8] V. T. Martins, D. Fonte, P. R. Henriques, and D. Da Cruz, "Plagiarism detection: A tool survey and comparison," *OpenAccess Ser. Informatics*, vol. 38, pp. 143–158, 2014.

[9] M. Mozgovoy, T. Kakkonen, and G. Cosma, "Automatic Student Plagiarism Detection: Future Perspectives," *J. Educ. Comput. Res.*, vol. 43, no. 4, pp. 511–531, 2010.

[10] S. B. Nick Fiducia, "When Two Worlds Collide: The Oracle And Google Dispute - Media, Telecoms, IT, Entertainment - United States," 2013. [Online]. Available: http://www.mondaq.com/unitedstates/x/271942/. [Accessed: 28-Feb-2016].

[11] L. Prechelt, G. Malpohl, and M. Philippsen, "Finding Plagiarisms among a Set of Programs with JPlag," *J. Univers. Comput. Sci.*, vol. 8, no. 11, pp. 1016–1038, 2002.

[12] D. Rattan, R. Bhatia, and M. Singh, *Software clone detection: A systematic review*, vol. 55, no. 7. Elsevier B.V., 2013.

[13] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Sci. Comput. Program.*, vol. 74, pp. 470–495, 2009.

[14] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: Local Algorithms for Document Fingerprinting," *Proc. 2003 ACM SIGMOD Int. Conf. Manag. data - SIGMOD '03*, pp. 76–85, 2003.