Available online at www.sciencedirect.com

ScienceDirect

Procedia Engineering 192 (2017) 136 – 141

Procedia Engineering

www.elsevier.com/locate/procedia

TRANSCOM 2017: International scientific conference on sustainable, modern and safe transport

Current trends in source code analysis, plagiarism detection and issues of analysis big datasets

Michal Ďuračík[a], Emil Kršák[a]*, Patrik Hrkút[a]

[a]*University of Zilina, Faculty of Management Science and Informatics, Univerzitná 8215/1, 010 26 Zilina Slovakia*

Abstract

In this work, we analyze the state of the art in source code analysis area with a focus on plagiarism detection and provide a proposal for a future work in this area. Detection of plagiarism combines the detection of clones and methods for determining similarity. Nowadays, there are several approaches that can be divided into three levels. The first one is text based and uses plain text as an input. The second level is token based. The top level is model based and uses models to represent source code. These advanced algorithms (token and model based) can't work with large datasets. We believe the future belongs to the algorithms that will be able to handle large amount of source code. These algorithms should use one of model-based representations. They can be used for formation of large-scale anti-plagiarism systems. They can be used also in the area of source code optimization.

Peer-review under responsibility of the scientific committee of TRANSCOM 2017: International scientific conference on sustainable, modern and safe transport

Keywords: source code analysis; plagiarism; current trends; transport; optimizations

1. Introduction

Source code analysis has existed ever since programming started. First, programming languages did not use to be very complex, and analyzing them did not require much effort. In the course of time the individual methods of analysis developed, as well as programming languages did. Currently, the computing speed of computers (especially

* Corresponding author. Tel.: +421-41-513-4050.
E-mail address: emil.Krsak@fri.uniza.sk

Peer-review under responsibility of the scientific committee of TRANSCOM 2017: International scientific conference on sustainable, modern and safe transport
doi:10.1016/j.proeng.2017.06.024

CPU and the size of RAM), number of source codes, of programming languages and programmers are increasing constantly. This represents new challenges for the analysis of source codes. The trend is likely to keep continuing, and the more people will do programming, the more programming languages there will be, and analyzing them will be more and more demanding from year to year.

In his article *Source Code Analysis: A Road Map* [1] from the year 2007, David Binkley describes methods that are still current and used in the analysis of source codes. Besides them, he deals with its future. According to him, the times when the duration of an analysis has been more serious of an issue than the volume of data has are almost over, and, gradually, the volume of data is going to constitute a problem bigger than the problem of time. The algorithms used currently will have to adapt over time.

2. Source code and its analysis

A source code is a text usually written by human programmers, and except for them, also the computer as a compiler or interpreter needs to understand it. Therefore, the text contains information necessary for both, the programmer and the computer, to understand the given code. For the computer to understand a source code, the code has to meet the syntax and grammar of the given language. As far as programmers are concerned, there are often formal and informal conventions that make the source code readable. Such conventions may include naming identifiers, documentation comments, code formatting etc. Sometimes these conventions use programming languages directly, and they add them to their grammar (e. g. in the programming language Python, indent is used right in the language to define blocks). It often happens that the conventions are not required directly by the programming language but by the frameworks which are built on it. Then the names of the individual identifiers are not a matter of the programmer´s choice but they depend on the tool / environment that the programmer works in. With some languages, also documentation comments have been applied. For example PHP, which is a dynamic-type, and programmers needing types when developing their frameworks (e. g. when using Dependency Injection) used the mechanism of documentation comments. These have become standardized, and they are understood by development environments, however, also a language can process them by reflection.

When analysing a source code, this attribute needs to be considered, to select the right analysis method. Processing the structure of source code does not present a major problem since the source code has to meet the grammar of the given language, so that it is possible to compile. Processing the meaning, respectively the idea a programmer has added into the code constitutes bigger challenge, regarding the fact there are various programmers with various levels of experience. As the problem is a complex one, the current methods of source code processing do not aspire to solve everything at once but they approach the source code from a particular point of view. Some analyze the code from the programmer´s point of view, and they try to understand its meaning, while others explore its structure. We can divide these methods into three levels based on how they approach the source code. The first of them is the analysis of source code as a plain text. With the method, it is presumed the programmer complies with the conventions, and the source code contains sufficient information describing its meaning. The algorithms being used are supposed to extract just these additional information. The next level is similar to the previous one but it does not explore the meaning of the text from the programmer´s viewpoint. It does so from the viewpoint of the computer that ´sees´ the source code as a list of commands. The individual commands sequences have their meaning in the context of the language grammar, and they tell the computer what it is supposed to do. The last level is the analysis of source code model.

3. Source code analysis using the means of NLP

We consider **Natural Language Processing (NLP)** as the basic method of acquiring information from text documents. The term natural language refers to the language people use in common communication (Slovak, English ...). NLP allows to process a natural language by using computers. Normally, the process has a few phases during which the entry document is processed gradually. In the first phase terms are extracted from the source document. They are represented in a certain way, and, consequently, by means of various classification methods, they are analyzed. Extracting terms from source code differs from extracting terms from text documents mainly by the form of text tokenization. Fortunately, there are good tokenization tools for each programming language.

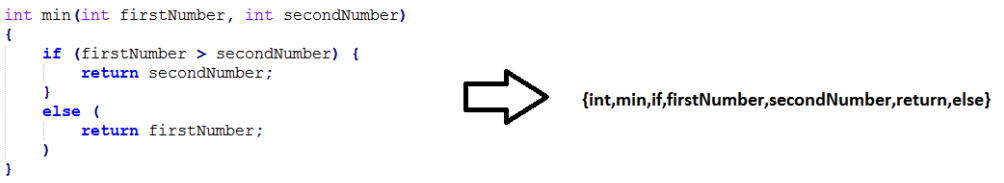

Fig. 1. NLP tokenization.

Besides tokenization, when a source code is being processed, a problem with the code itself arises, too. Programming languages provide a standard library with basic functions, classes and methods. These are usually named in a logical manner, and they follow certain conventions, therefore processing them does not constitute any major problem [2]. However, there might be a difficulty with the programmer´s code itself. Every programmer may name their identifiers using their own ideas and their own language. Figure 2 shows two extracts of a source code that does the same. Code A contains logically named identifiers and comments, and Code B does not. For NLP it is difficult to process Code B since it does not make any sense to it.

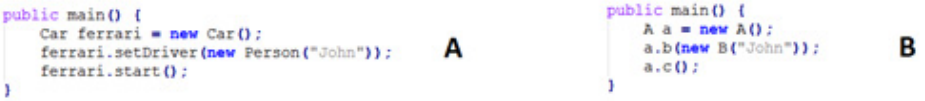

Fig. 2. Good and bad source code comparison.

A lot of works deal with processing text documents by means of NLP, nevertheless, using NLP to process source codes has not been completely common yet. Among the works dealing with the topic there are those describing using n-grams to identify the authors of source codes [3], or to seek malicious codes [4,5]. Another NLP use is to support development [6,7] and to identify the programming language from the source code extracts [8].

Another direction the research in the given field has been taking is the way of representing source codes. The above mentioned works have used in particular the representation of source code as set of terms, respectively n-grams from those terms. When several source codes are being worked with, the standard form of representation is **Document-term matrix (DTM)**. DTM is a matrix the lines of which correspond to the individual source codes, and the columns represent the terms. The matrix describes the occurrence of terms in the individual source codes. Each matrix line is a vector of the given source code.

The main problem of DTM is its volume, which makes demands on memory capacity, which is often solved by using sparce matrices. This, on the other side, adds up to computational complexity. Besides, the efficiency is not very high because if we wanted to be looking up synonyms for instance, then the computational complexity would be increasing again. This problem gets solved by means of **Latent Dirichlet Allocation (LDA)**, and there are works that describe its being used also for source codes [9]. LDA is used to transform the source code representation by means of the terms of the representation that uses the **topic model**. We can picture the topic model as an n of important terms describing the given source code. We may come across using the topic model to analyze source code e. g. in works exploring software evolution [10,11].

4. Token analysis

A token represents an indivisible sequence of characters which has a certain significance for the purposes of parsing the source text. With the analysis of source code by means of tokens, similar methods to those used with processing by means of NLP are utilized. The difference is in understanding the meaning of a particular characters sequence. In the case of NLP, the meaning inserted into the given token by the programmer is used. On the other hand, with token analysis this meaning is unimportant, and only the meaning based on the language grammar is used.

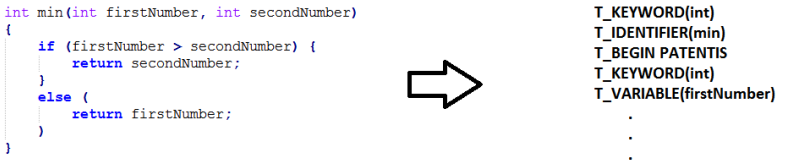

Fig. 3. Source code tokenization.

This level of source code processing is the oldest one among the others mentioned above. It has been used since the beginnings of programming languages. Over time, it is getting improved by new knowledge from other areas, or new, more modern methods are being built on it. It has been used widely, from the programming supporting tools, which utilize it for instance to check syntax correctness, up to large systems, to check the quality and safety of source code.

An interesting area is the analysis of plagiarism and searching for similarities of source codes. This, again, is based on the knowledge from searching similarities of text documents. There are various systems and solution approaches regarding this problem, however, JPlag and Measure of software similarity (MOSS) are considered as the main ones. What most of these systems have in common are their initial stages (code purification, tokenization), and the postprocessing can be divided into several groups, depending on what they use (n-Grams, Winnowing, Running–Karp–Rabin Greedy-String-Tiling, Fuzzy-based).

The method **n-grams** is one of the basic methods to analyze similarities, and it is used by, e. g., PIY (program it yourself) [12]. It is based on dividing the entry document into the list of substrings of length n. In the result the document is presented as a list of strings of length n and by their frequency.

Winnowing is the basis of the system MOSS. It constitutes an improvement of the method n-grams, and it uses so called n-grams fingerprinting which uses hashing to select a subset from n-grams of the document as an imprint of the document. The most frequently used way is usually selecting n-grams the hash of which is 0 mod p. Thus, if we choose p = 5, then each fifth hash will be used.

Running–Karp–Rabin Greedy-String-Tiling (RKR-GST) is the most popular alternative to Winnowing algorithm. It consists of two parts: Greedy-String-Tiling (GST) is an algorithm for finding the maximum common substring of two strings. Running–Karp–Rabin (RKR) is an algorithm for quick substring matching, and for searching for short substrings in long strings using hashing. When looking for similarities, at first, RKR is used to find common substrings, consequently, the substrings GST are used for more precise search for substrings. We can find its usage in the system JPlag.

In addition to standard comparison methods we will also find algorithms that are based on fuzzy logic. An example of this may be Self-Organising Map (SOM). The main advantage of these systems is that they can be independent on the programming language [13].

5. Creating and analyzing a source code model

A tokens stream is an effective form of a source code representation. With its simple analysis, usually the design pattern visitor is used. The algorithm goes step by step through the individual tokens and it executes the relevant operations over them. Nevertheless, it is hard to analyze more complex problems using this approach. If we want to carry out more complicated analyses, then we have to transform the current into a suitable form. Almost each tool that works directly with tokens current utilizes its own form which it saves the tokens into. However, there is one universal representation – **Abstract Syntactic Tree (AST)**. AST describes the logical structure of source code without pointless platform-specific parts. AST is usually created from a tokens stream. It is used with most compilers, IDE, as well as for various analyses of source codes.

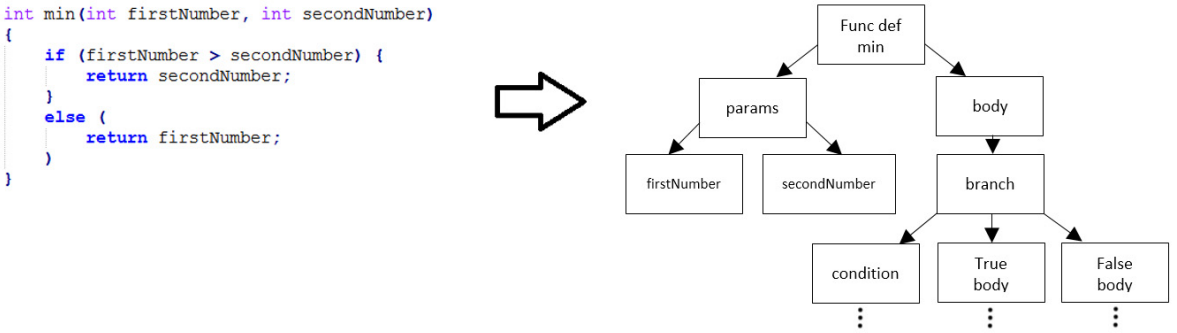

Fig. 4. AST representation of source code.

AST naturally finds its use also with the plagiarism detection algorithms. Compared to the algorithms that work with texts or tokens, the main advantage of AST is that it directly describes the structure of the program, without any particular implementation details. CodeCompare [14] and AST-CC [15] belong to algorithms which have been successfully using AST to detect plagiarism in source codes. Except for this one, there are works describing miscellaneous uses of AST for detecting similarities in source codes [16].
The algorithms are based on a simple idea – comparing tree structures. Since comparing tree structures is a demanding operation, the algorithms use hashing methods for comparing. That enables them to work relatively quickly. Thanks the hashing they can simply implement the detection of the most various transformations used for plagiarism (sequence reversing, logical expressions inversion, ...).

6. Problems of the current source code analysis methods

Current algorithms used for plagiarism detection are significantly advanced, however, they are mostly limited by a number of source codes that they can process. There are algorithms which use the techniques of indexation known from NLP but they work only over a tokens stream [17]. The algorithms cannot work effectively with structures such as a source code. That is why we see the future in algorithms that could process and index a large number of source codes, which would consequently allow to look up structures in the codes very fast. The algorithms should be using an abstract syntactic tree as a base, since it suits the source code representation the best. Last but not least, the rate of what will be also equally important is the level of parallelization of the algorithms. We can see the use of the algorithms with creating extensive anti-plagiarism systems, as well as with the possibility of searching for certain patterns across a large code base, which can be used for example to find anti-patterns or to follow various trends.

7. Using anti-plagiarism algorithms for code optimization

The research we describe in the article aims at seeking similarities among source codes. This attribute can be used also in the area of source code optimization. As we have mentioned before, the systems often have common parts, and identifying the parts would help us with more effective development of the systems in the future.

The Department of Software Technologies has been dealing with creating information systems for transportation for a long time. These includes KANGO [18] and ZONA [19] systems, designed for creating transportation timetables. The information system VIS [20] for finding an optimal route and graphic-technological extension (GTN) [21] of an electronic lock-off device. Even though the systems are various, similar parts, modules respectively, can be found in their architectures. If we were able to identify the modules successfully, it would be possible to create a general framework which, in the future, would allow to create a skeleton of information systems bringing the advantage of the recurring system part being available from the very beginning, and that would make the development faster and more effective.

8. Conclusion

In our work we deal with the algorithms for source code analysis. The algorithms we have described were aimed at finding similarities among several source codes. We will find them being used primarily in anti-plagiarism systems. Moreover, we also deal with the possible usage of the algorithms in transportation systems. The systems often share a lot of common elements, and identifying the elements could help us further develop the systems. Our goal is to design an algorithm which would allow to process a number of source codes, so that we can apply it to the systems.

References

[1] D. Binkley. Source code analysis: A road map. In ICSE - Future of SE Track, 2007.
[2] N. R. Carvalho, J. J. Almeida, P. R. Henriques, and M. J. Varanda, From source code identifiers to natural language terms [J]. The Journal of Systems and Software, 2015, 100, 117–128.
[3] G. Frantzeskou, E. Stamatatos, S. Gritzalis, and S. Katsikas. Source code author identification based on N-gram author profiles. IFIP International Federation for Information Processing, 2006, 204, 508-515.
[4] J. Upchurch and X. Zhou. First byte: Force-based clustering of filtered block N-grams to detect code reuse in malicious software. In 2013 8th International Conference on Malicious and Unwanted Software: "The Americas" (MALWARE), 2013, 68-76.
[5] Y. Li, K. Bontcheva, and H. Cunningham. Adapting SVM for data sparseness and imbalance: a case study in information extraction. Natural Language Engineering, 15(1):241, 2009.
[6] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In Proceedings – International Conference on Software Engineering, volume 20, pages 837-847, 2012.
[7] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Learning natural coding conventions. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014, pages 1-13, 2014.
[8] J. K. van Dam. Identifying source code programming languages through natural language processing. 2016.
[9] D. Binkley, D. Heinz, D. Lawrie, and J. Overfelt. Understanding LDA in source code analysis. In Proceedings of the 22Nd Int'l Conf. on Program Comprehension, pages 26–36, New York, NY, USA, 2014. ACM.
[10] S.W. Thomas, B. Adams, A.E. Hassan, D. Blostein, Studying software evolution using topic models, Sci. Comput. Program. 80 (2014) 457-479.
[11] E. Linstead, C. Lopes, and P. Baldi. An application of latent dirichlet allocation to analyzing software evolution. In Proc. 7th Intl. Conf. on Mach. Learn. and App. , pages 813–818, 2008.
[12] T. Ohmann and I. Rahal. Efficient clustering-based source code plagiarism detection using PIY. Knowl. Inf. Syst. , vol. 43, no. 2, pp. 445–472, May 2015.
[13] G. Cosma and G. Acampora. A Fuzzy-based Approach to Programming Language Independent Source-Code Plagiarism Detection. IEEE International Conference on Fuzzy Systems (FUZZ-IEEE), Istanbul. pages 1-8, 2015.
[14] G. Tao, D. Guowei, Q. Hu, C. Baojiang. Improved plagiarism detection algorithm based on abstract syntax tree. In: Emerging Intelligent Data and Web Technologies (EIDWT), 2013 Fourth International Conference on. IEEE, 2013. p. 714-719.
[15] J. Zhao, K. Xia, Y. Fu, B. Cui. An AST-based Code Plagiarism Detection Algorithm. In: Broadband and Wireless Computing, Communication and Applications (BWCCA), 2015 10th International Conference on. IEEE, 2015. p. 178-182.
[16] FM. Lazar, O. Banias. Clone detection algorithm based on the Abstract Syntax Tree approach. In: Applied Computational Intelligence and Informatics (SACI), 2014 IEEE 9th International Symposium on. IEEE, 2014. p. 73-78.
[17] BURROWS, Steven; TAHAGHOGHI, Seyed MM; ZOBEL, Justin. Efficient plagiarism detection for large code repositories. Software-Practice and Experience, 2007, 37.2: 151-176.
[18] Informačný systém KANGO v rokoch 2014 a 2015 / Karel Šotek ... [et al.]. - 2015 In: Horizonty železničnej dopravy 2015 S. 220-230 medzinárodná vedecká konferencia zborník príspevkov Horizons of railway transport 2015 Strečno 30. september - 1. október 2015 Žilina Žilinská univerzita 2015, ISBN 978-80-554-1097-5.
[19] Nové trendy v IS ZONA v prostředí železnice na Slovensku / Karel Šotek, Hynek Bachratý, Emil Kršák. - 2010 Obr. In: Doprava ekonomicko - technická revue Roč. 52, č. 6 (2010), s. 5-9, ISSN 0012-5520.
[20] VIS - nový informačný systém pre vyhľadávanie spojení v dopravných sieťach / Marek Tavač, Hynek Bachratý. - 2005 Obr. In: Infotrans 2005 informační technologie v dopravě a logistice IV. Ročník mezinárodní konference, Pardubice, 21.-22.9.2005 S. 289-297 elektronický zdroj Pardubice Univerzita Pardubice 2005, ISBN 80-7194-792-X.
[21] GTN - information system supporting the dispatcher and remote tracks control / Emil Krsak - Hynek Bachraty - Vlastimil Polach. - 2010 Fotogr., obr. In: Communications Vol. 12, No. 3A (2010), s. 65-73 scientific letters of the University of Žilina, ISSN 1335-4205.