Source-code Similarity Detection and Detection Tools Used in Academia: A Systematic Review

MATIJA NOVAK, Faculty of Organization and Informatics, University of Zagreb, Croatia
MIKE JOY, University of Warwick, Department of Computer Science, United Kingdom
DRAGUTIN KERMEK, Faculty of Organization and Informatics, University of Zagreb, Croatia

Teachers deal with plagiarism on a regular basis, so they try to prevent and detect plagiarism, a task that is complicated by the large size of some classes. Students who cheat often try to hide their plagiarism (obfuscate), and many different similarity detection engines (often called plagiarism detection tools) have been built to help teachers. This article focuses only on plagiarism detection and presents a detailed systematic review of the field of source-code plagiarism detection in academia. This review gives an overview of definitions of plagiarism, plagiarism detection tools, comparison metrics, obfuscation methods, datasets used for comparison, and algorithm types. Perspectives on the meaning of source-code plagiarism detection in academia are presented, together with categorisations of the available detection tools and analyses of their effectiveness. While writing the review, some interesting insights have been found about metrics and datasets for quantitative tool comparison and categorisation of detection algorithms. Also, existing obfuscation methods classifications have been expanded together with a new definition of "source-code plagiarism detection in academia."

CCS Concepts: • **Social and professional topics** → **Computing education**; • **Applied computing** → **Education**; E-learning; • **Information systems** → **Near-duplicate and plagiarism detection**;

Additional Key Words and Phrases: Source-code, plagiarism, similarity, detection, academia, education, programming, systematic review

ACM Reference format:
Matija Novak, Mike Joy, and Dragutin Kermek. 2019. Source-code Similarity Detection and Detection Tools Used in Academia: A Systematic Review. *ACM Trans. Comput. Educ.* 19, 3, Article 27 (May 2019), 37 pages.
https://doi.org/10.1145/3313290

1 INTRODUCTION

Source-code plagiarism research has increased over the years and has become a more popular field. Research into source-code plagiarism detection had already been performed in the 1970s, for example, a paper [130] from Ottenstein was published in 1976, but some mechanisms that he used to detect plagiarism had already been presented a few years earlier. These included counting of operands and operators, which were explained by Halstead in [57] and expanded upon later in his 1977 book *Elements of Software Science* [58], which contained detailed descriptions of the

Authors' addresses: M. Novak and D. Kermek, Faculty of Organization and Informatics, University of Zagreb, Pavlinska 2, Varaždin, 42000, Croatia; emails: matija.novak@foi.hr, dragutin.kermek@foi.hr; M. Joy, University of Warwick, Department of Computer Science, Coventry, CV4 7AL, United Kingdom; email: M.S.Joy@warwick.ac.uk.

1946-6226/2019/05-ART27 $15.00
https://doi.org/10.1145/3313290

mechanisms that were used in the early days of plagiarism detection. Halstead was not focused solely on plagiarism detection, but his work inspired authors like Ottenstein to use the techniques he introduced for that purpose.

Although source-code plagiarism detection papers go back to the 1970s, and research in the field has been continuously undertaken by a few authors, it took 30 years for this field to become more popular, and only in the past 10 years have scientists had a real interest in this field and been able to talk about different views and aspects of plagiarism detection.

In this review article, the focus is on source-code plagiarism detection tools in academia. The complete set of constraints with motivation and research questions is given in the next chapter, while Section 2 presents the review protocol, Section 3 analyses the selected papers in multiple categories, and Section 4 concludes.

2 REVIEW PROTOCOL

The systematic literature review follows protocols presented by Brereton et al. (software engineering) [18], Spolaor and Benitti (learning theories) [159], and Breivold et al. (software architecture) [17]. This section is divided into the following subsections: motivation and research questions; restrictions and selection criteria; search terms, database selection, and paper search process; and data extraction.

2.1 Motivation and Research Questions

Plagiarism is a serious problem in academia and naturally a lot of research is trying to solve that problem form different directions. An initial survey of papers published in 2016 reveal that some try to prevent plagiarism [62], some try to detect it [151], some do improvements [83], some are focused on comparisons [114], and so on.

The interest in analysing these papers is to discover how this field has developed and specifically what the tools used for detection are, which tool is better, how students hide their attempts at plagiarism, and so on. It would be very hard, and probably even impossible, to cover the whole plagiarism research area, so focus is only on source-code plagiarism detection tools in academia and the motivation to choose this specific area is that the authors of this review are all teachers in academia teaching programming courses.

Much work and many different approaches have been applied to source-code plagiarism detection, and the questions that this review seeks to answer in the review are as follows:

- Q1: What is meant by source-code plagiarism in academia?
- Q2: What is meant by source-code plagiarism detection in academia?
- Q3: What obfuscation methods do students use to hide source-code plagiarism?
- Q4: What detection tools are used and which are the best?
- Q5: What algorithms are used in these detection tools?
- Q6: What measures are used to compare the tools?
- Q7: What datasets are used to compare the tools?
- Q8: Where should one search for articles dealing with source-code plagiarism detection in academia?

To be consistent, and to reduce the number of papers under investigation to a feasible number, different restrictions and selection criteria were defined.

2.2 Restrictions and Selection Criteria

The restrictions criteria were divided into two groups: general and domain specific. General critera can be applied in any review while domain specific critera are useful for this article and maybe similar articles in the plagiarism detection domain. From this point, the restrictions criteria are referred sometimes as exclusion criteria, because it defines what types of articles will *not* be included in the review.

2.2.1 General Exclusion Criteria.

- Papers that are not written in English language and papers that have only title and/or abstract written in English, although there are good papers in other languages dealing with source-code plagiarism detection.
- Articles that were added to the searched databases after 29.08.2016.
- Multiple versions of the same article: Only one version of the article from the same author is included. For example, if an author writes a conference paper with partial results and after that a journal paper that has full results, then the conference paper is excluded. Also, if author has one paper that mentions briefly a system for plagiarism detection but also has an article focusing on this system for plagiarism detection, then the former is excluded.
- Unreviewed reports and technical papers: Although there are some interesting technical papers or reports, if reports have scientific value, then usually there is a follow-up paper that is then included if it meets the other criteria.
- Theses: All thesis topics sooner or later have follow-up papers that cover the same topic.
- Covers (journal, conference proceedings covers), posters, discussions.

2.2.2 Domain Specific Exclusion Criteria.

- "Industrial" articles: Papers about source-code plagiarism detection that are not from an academic environment, like open source projects and source-code from companies.
- Papers that are not focused on *plagiarism*: This excludes papers (for example) that develop new algorithms for string matching that are not specifically used for plagiarism detection. So, if there are no actual results that show that some algorithm works well on plagiarism detection, then that paper is excluded. This also means excluding algorithm-focused papers that do not contain evaluation of potential plagiarized documents.
- Papers that are not focused on plagiarism *detection*: This means papers that are focused on prevention of plagiarism and discussion of pedagogical issues are excluded. The only exception from this rule are papers that talk about student/staff perceptions of source-code plagiarism and help us to understand what is meant by source-code plagiarism in academia and are not excluded based on other exception criteria.
- Papers that are not focused on *source-code* plagiarism detection: There are techniques from related fields that could be used in source-code plagiarism detection but that have not actually been used. In some papers, authors state "this can be used in plagiarism detection," but no research with plagiarism detection in academia was performed.
- Malware detection, source-code refactoring, bug fixing, assignment evaluation, or assessment: These arefour domains from which techniques could have some value to source-code plagiarism detection but for now are not used for plagiarism detection.
- Source-code reuse: The focus of code reuse is slightly different, because in code reuse the user copies similar code and makes changes to that code to make it work for another purpose, and the user does not want to hide the copying as he does when plagiarizing. That is why this kind of similarity detection is different than similarity detection in plagiarism.

However, if an article uses code reuse principles for plagiarism detection then the article is included.

- Authorship attribution: "… is not equivalent to the problem of detecting copied programs … often characterized as plagiarism detection …" [162]. "Source code authorship attribution … is determining whether a program is significantly similar (usually in a stylistic sense) to other programs written by the purported author, where the programs being compared were meant to have solved completely unrelated problems" [162]. In other words, the primary idea of authorship attribution is to identify the author of a program. One might be interested if two programs were written by the same author, which can be used for plagiarism detection. So if not explicitly stated that authorship attribution technique is used for source-code plagiarism detection in academic environment the article is excluded from the review.
- Code cloning: "… detection techniques analyze single programs developed by a team of developers, whereas plagiarism detection techniques compare multiple programs implemented by independent developers; programmers unintentionally introduce code clones in software programs, whereas plagiarism is the result of programmers who intentionally copy another program and then attempt to obscure their unethical activity; finally, code clone detection techniques aim to identify the (few) cloned fragments that exist in (large) applications, whereas plagiarism detection techniques generally aim to establish whether entire programs result from plagiarism"[107]. It might be that some code cloning techniques would also be good for plagiarism detection, but papers in which they were not actually used for plagiarism detection are not analysed.
- Textual plagiarism: This is another related field and some techniques from textual plagiarism are used for source-code plagiarism detection. But, as with other related fields, if there is not a focus on source-code plagiarism detection in academia, then it is excluded.
- Papers with only one paragraph about source-code plagiarism detection: Papers that have used source-code plagiarism detection in academia and mention it very briefly, but the real focus of such papers is different. These papers, if they do not have some evaluation of the system used with actual plagiarism related results, are excluded.

2.3 Search Terms, Database Selection, and Paper Search Process

To find articles that will answer the research questions databases of scientific papers where queried. The decision as to what databases to choose was made in three steps. First, systematic review papers such as Reference [17] were consulted for what databases they used. To this list, the authors added databases that they personally use when searching for relevant papers. Second, these databases were queried for important papers in the field of plagiarism detection that the authors know about, and reference scanning was performed to see if any important material was missed. Third, to reduce the number of databases to a feasible amount some were removed based on the following criteria. If a database contains only papers (that were searched) that were also found in some other databases, then it is excluded. Note that this does not mean that these databases do not contain *at all* relevant papers that are not found in other databases. The databases that were chosen are as follows: *SCOPUS*, *ACM digital Library* (The ACM Full Text collection), *IEEE Xplore* (metadata only), *Science Direct* (SD), and *Web of Science* (WOS).

To search the databases, various combinations of queries were tried out. Some combinations of keywords that were tried out just gave too many results, for example: *program AND similarity* gave over 13,000 papers only in SCOPUS and *application AND similarity* gave 358,527 papers only in SCOPUS. To limit the amount of articles, so that they can be processed in a reasonable time, and at same time have a sufficient and representative number of articles to analyse, the chosen

Table 1. Overview of the Number of Papers Returned by Databases

	ACM	IEEE	SD	SCOPUS	WOS	Sum
28.2.2015	613	585	137	1,649	809	3,793
20.8.2015	629	558	201	1,613	829	3,830
29.8.2016	518[a]	623	221	1,887	1,354	4,603
Total after importing everything into medley						3,419
Total after using "check for duplicates" option						3,183
Total after removing remaining duplicates manually						3,176
Total after removing of covers						3,069
Total after removing all unrelated papers (biochemistry, electromagnetism, ...) found manually by reading only titles						1,214
After second reading of abstracts, conclusion and introduction						314
Total after the complete reading of all papers						***150***

[a]Number of results in ACM has lowered instead of going up, because the search engine of ACM has changed between the searches and the way of writing queries (Appendix A) so there is an inconsistency in the results in ACM.

(although not perfect) keyword combinations were *source AND code AND plagiarism; source AND code AND similarity; application AND plagiarism; program AND similarity.* The final search query was as follows:

((source AND code) AND (plagiarism OR similarity)) OR
((application OR program) AND plagiarism)

Each database was queried three times on three different dates. The numbers of papers found by the queries are shown in Table 1. Authors could not obtain full text for 12 papers, so they were excluded from the review. Excluded papers are References [14, 43, 51, 65, 77, 137, 142, 144, 145, 149, 176, 183].

All the queries in databases where stated to search *title, abstract, and keywords* if it was possible to define that. Full text was intentionally excluded, because if a paper is in the field, it should have those combinations of keywords in the title, abstract, or keywords. To help with managing the papers during the review the reference management tool Mendeley [161] was used.

2.4 Data Extraction

Once the final list of papers was obtained, a content analysis and review was performed while considering [17] (1) *General paper information*: title, authors, publication year, source databases, and Google Scholar citation and (2) *Content-related information* about tools (used, created, or compared), comparison measures, algorithms, datasets, programming languages, definitions of plagiarism, similarity measures, and obfuscation methods.

The information was stored in Microsoft Excel, and in total there were 190 attributes that were used to track each article. Most information were binary, for example, regarding the tools information there were 17 columns that represented single tools and the information was simply whether the article does or does not use the tool.

3 ANALYSIS AND DISCUSSION OF SELECTED PAPERS

There were 150 papers extracted, and in this section various analysis are presented based on these papers. The reference list is given in Appendix B.1.

All graphs and quantitative analyses were performed using the statistical tool R. To ensure that all newly created functions for processing the quantitative data (like creating frequency tables) are

correct unit tests were written for each such function, and a test-driven development discipline was applied while programming such functions. Graphs were examined manually and checked for correctness. The qualitative analyses were performed manually, since there was no way to automate them and the results were validated by all authors of the article to ensure the validity of the qualitative analyses.

3.1 Definition of Plagiarism

It has already been stated that plagiarism is a serious problem in academia, especially in distance learning enabled with in e-learning environments, as stated, for example, in Reference [106]. But what is plagiarism exactly, and how is it defined? The definitions of plagiarism have put in three categories: the general definition of plagiarism, the definition of source-code (program) plagiarism, and the definition of source-code plagiarism in academia.

A commonly used general definition of plagiarism can be found in the Merriam-Webster Online Dictionary [37]. The dictionary actually gives four definitions for the word plagiarize (or plagiarized or plagiarizing): *"To steal and pass off (the ideas or words of another) as one's own"; "use (another's production) without crediting the source"; "To commit literary theft"; "present as new and original an idea or product derived from an existing source."* Articles that are using some variation of these definitions are References [9–11, 20, 27, 30, 41, 56, 78, 79, 81, 85–87, 91, 107, 110, 116, 122, 131, 154, 160, 166, 168, 180].

Variations of these definitions that exist in the various sources include the following: *"Plagiarism is the act of imitating or copying or using somebody else's creation or idea without permission and presenting it as one's own"* [91]; *"Plagiarism is the act of copying (and sometimes superficially modifying) work of others and submitting this as one's own"* [56]; *"Reproducing someone's work without acknowledging the source is known as plagiarism"* [79]; *"Plagiarism —usage of some web content as one's own"* [87]; *"Plagiarism is the unethical practice of taking someone else's ideas, data, findings, language, illustrative material, images, or writing, and presenting them as if they were your own"* [107].

In Reference [56], authors give a slightly different definition of plagiarism from the Department of Computer and Information Science of Utrecht University, which states: *"Fraud and plagiarism are defined as actions, or failure to act, on the part of a student, as a result of which proper assessment of his/her knowledge, insight and skills, in full or in part, becomes impossible."*

Keeping in mind the general definition of plagiarism, program (or source-code) plagiarism can be defined as follows: "*Source code plagiarism can be defined as trying to pass off (parts of) source code written by someone else as one's own (i.e., without indicating which parts are copied from which author)"* [55] or *"Code plagiarism as the unauthorized reuse of program structure and programming language syntax"* [20]. Articles using variations of such definitions are References [7, 20, 55, 75, 107, 120, 124].

But the most used definition in our sample is the one from Parker and Hamblen [131]: *"A plagiarized program can defined as a program which has been produced from another program with a small number of routine transformations. Routine transformations, typically text substitutions, do not require a detailed understanding of the program"* [131]. Articles using this definition or some variation of it are References [5, 25, 63, 79, 98, 112, 131, 160, 166, 167].

Some simplified versions of Parker and Hamblen definition are as follows: *"A program which has been produced from another program with a small number of routine transformations"* [160] and *"A plagiarized program can be defined as a program that has been produced from another program without a detailed understanding of the source code"* [98].

There are also other definitions that are very similar to the one from Parker and Hamblen: *"Plagiarism: a plagiarized program is a program that can be obtained from the original one by means*

of one or more of the actions listed above" [116]. Here the "listed above" are meant the obfuscation methods and the list can be found in Reference [116]. *"Plagiarism as the application of successive transformations applied on an original document. A transformation preserves the program function but not it's appearance"* [99]. *"A plagiarized program is either an exact copy of the original, or a variant obtained by applying various textual transformations such as verbatim copying, changing comments, changing white space and formatting, renaming identifiers, etc."* [184].

There is also one interesting and different definition from Ohno and Murao [127], but the definition itself is given in a previous version of the article [126]: *"We call it a plagiarism when we find "a source code written in a coding style which does not fit to the coding model of who wrote the source codes."*

The last category of definition that is analysed is the definition of source-code plagiarism among students in academia. There are articles that deal with students perspectives about plagiarism like the one from Joy et al. [74], Aasheim et al. [1], or Zhang et al. [180], but most deal with it from the academic point of view like [32]. One interesting observation about plagiarism among students was done using social network analysis by Luquini and Omar [104]: "*Students committed to plagiarize will rely on closer classmates to obtain a copy of the needed artefact."*

By reviewing the selected articles the most used definition here is the one from Cosma and Joy [32]: *"Source-code plagiarism in programming assignments can occur when a student reuses … source-code authored by someone else and, intentionally or unintentionally, fails to acknowledge it adequately …, thus submitting it as his/her own work. This involves obtaining … the source-code, either with or without the permission of the original author, and reusing … source-code produced as part of another assessment (in which academic credit was gained) without adequate acknowledgement …. The latter practice, self-plagiarism, may constitute another academic offense."*

A shorter version of this definition is given in Reference [33]: *"Source-code plagiarism occurs when students reuse source-code authored by someone else, either intentionally or unintentionally, and fail to adequately acknowledge the fact that the particular source-code is not their own."* Articles that use this definition or a similar variation of it are Reference [32, 33, 44, 114, 125, 138, 156, 156, 164].

Different variations of the definitions are the following: *"Our definition of source code plagiarism in education includes acts of copying source code written in any programming language and submitting the copied code for evaluation as if it were one's own work with no, minor, or even major modifications aimed at concealing plagiarism and without properly acknowledging the original source"* [138]. Plagiarism is *"to purchase the work of others and submit it as one's own …; to incorporate the work of another student without their permission … or to include the work of others without referencing the source … borrowing and using another student's work … giving completed work to another student and asking them to improve it …"* [156].

A somewhat different definition that differs between mono language and cross language plagiarism is given in Reference [150]: *"Mono language plagiarism: It is the act of producing source code file from another source code file of same language just by doing text edit operation and not understanding the granularities of the program. … Cross language plagiarism comes into picture when students want a source code for particular functionality in language A and while surfing the internet students come across the exact source code for the functionality but in language B so student decides to plagiarize by translating syntax of commands on A to syntax of B without understanding the working of the code."*

Based on the reviewed articles, one can conclude that the general definition given in dictionaries is quite a suitable one and highly accepted in the field. Regarding the definition of plagiarism for source-code (program) the Parker and Hamblen definition is an appropriate one. Finally, the

definition of plagiarism for source-code in academia the one form Cosma and Joy appears to be appropriate.

But there is one more definition that is not about plagiarism but about the suspects of plagiarism that can be useful in academia, given by Brixtel et al. *"We consider two or more documents to be suspects if they are much more similar than the average similarity between documents"* [19]. This is not specifically for source-code, but there are again many similarities between soruce-code and written text. A useful comparison of scenarios that could be considered plagiarism between essay and source-code is given by Simon et al. [156].

Previous text explains what source-code plagiarism in academia is, so the next question is how do we detect source-code plagiarism in academia? But before this can be answered, it is necessary to know about different obfuscation methods that the students are using to hide plagiarism.

3.2 Obfuscation Methods

There are various obfuscation methods used by students to hide plagiarism. Usually, one can agree with the state from Faidhi and Robinson [42] that "*novice, student plagiarism mainly utilizes certain stylistic and syntactic changes, while expert programmers may introduce semantic changes (e.g. changing the data structures used, changing an iterative process to a recursive process, etc.) as well.*"

There are 72 articles that mention some kind of obfuscation methods (OM), and in these articles around 25 obfuscation methods were identified. Some of these 25 OM were grouped as one method, so 16 distinct obfuscation methods were specified (more details at the end of this subsection). For example, Đurić and Gašević [166] mention the following three obfuscation methods: (1) changing the order of variables in statements, (2) changing the order of statements within code blocks, and (3) reordering of code blocks. Such methods have been grouped under a unified name "Reordering independent lines of code." Different authors give different names to the same things, for example, Donaldson et al. [39] mention the method "transposing statements," and Whale [170] mentions "changing the order of independent statements," which are basically describing the same thing.

While 25 obfuscation methods were identified (grouped into 16 methods), one could find more methods. For example, the technical report from Cosma and Joy [31] identifies around 50 ways students can modify code. But the 50 modifications can mostly be organized into 1 of the 16 categories mentioned here. For example, "Reordering functions," "Reordering code within functions," and "Modifying the order of the selection statement" can be seen as variants of "Reordering independent lines of code." Also, they have four different comment modifications, whereas we treat comments modification as a single modification. However, some modifications mentioned in Reference [31] can be seen as doing several of the 16 modifications, such as "Fixing bugs." In this analysis, technical reports were excluded, but they are worth reading, since they can include issues that were not analysed here, for example, some modifications that are mentioned in Reference [31] such as modifications in user manuals and other supporting documents or modifications in the interface like colours and fonts.

Most of the 16 obfuscation methods can be obtained (except for 2) by looking at three articles by Faidhi and Robinson [42], Whale [170], and Đurić and Gašević [166]. The two that are missing are as follows: "Simplifying the code," which is first mentioned by Gier [54] and "Translation of program from other programming language," which is first mentioned by Arwin and Tahaghoghi [9].

In most cases, the obfuscation methods are divided into structural and lexical changes that were first introduced by Joy and Luck [75]. This classification is accepted in the most recent articles so it is safe to regard it as a standard classification. Another common classification are the six levels (plus level zero, which means no changes) from Faidhi and Robinson [42]. Chen et al. [23] introduced the *"Pyramid of Program Modification Levels"* with level zero (*no changes*) up to level

eight (*Replacing the control structure with equivalent one*), which can be seen as an improved version of the levels from Faidhi and Robinson.

These levels from Faidhi and Robinson can be matched to the classification of Joy and Luck as stated in the review form Novak [122] and was originally identified by Lancaster and described in his Ph.D. theses [92, p. 27]. The same mapping can be applied to the levels from Chen et al. [23].

The 16 identified methods are described together with the following: ID of the method, article that mentioned the method first, short name, and short description. A list of which articles have mentioned this method are presented in Appendix B.2. Also each method is classified into one of the four categories: *lexical changes (suffix L), structural changes (suffix S), advanced structural changes (suffix AS),* and *logical changes (suffix LG).* Lexical and structural changes are defined by Joy and Luck [75], while advanced structural changes and logical changes are new categories invented while writing this review. The definitions of the four categories are the following. Lexical changes are as follows [75]: *"changes which could, in principle, be performed by a text editor. They do not require knowledge of the language sufficient to parse a program."* A structural change is a change that [75] "*requires the sort of knowledge of a program that would be necessary to parse it. It is highly language-dependent.*" Advanced structural changes are defined as follows: "*subcategory of structural changes that require more knowledge of program possibilities and relations between equivalent statements in a specific programming language.*" Logical changes are defined as "*changes that except for structural changes also change the logic (flow) of a program and require certain amount of programming skills and knowledge about the application being developed to be performed correctly. They are very unlikely to be performed by total beginners."*

The order of the obfuscation methods should indicate the increasing complexity of the methods and mostly follows the order of Đurić and Gašević [166]. This complexity manifests itself in two ways: First, it is likely to take more effort to perform such a modification, and, second (more importantly), it is in most cases harder to detect. The identified obfuscation methods are (ID – First Article – Name – Description):

- OM_01_L – Faidhi and Robinson [42] – "Visual code formatting" – Changes to "code" so it appears different at first look, these include usually modifying whitespace like indents, spaces, new lines, and so on.
- OM_02_L – Faidhi and Robinson [42] – "Comments modification" – Changes of comments in code that includes changing, adding or removing comments of parts of comments.
- OM_03_L – Đurić and Gašević [166] – "Translation of program parts" – Translating parts of the program form another language like from English to Croatian. The parts are usually variable names, comments, output and so on. These modifications could be seen as special cases of OM_02_L, OM_04_L and OM_05_L.
- OM_04_L – Donaldson et al. [39] – "Modifying program output" – Modifying the output of the program so only LOC that print something are changed or if it is a graphical user interface (GUI) these would be changes to the labelling, reposition, and so on. However, if languages like HTML or CSS are included then reposition belongs to method OM_07_S.
- OM_05_L – Donaldson et al. [39] – "Identifier rename" – Changing the names of identifiers like variable names, constant names, function names, class names, and so on.
- OM_06_L – Prechelt et al. [134] – "Changing constant values" – Changing the values of constants, final variables, enum values, and so on. that do not change anything in the logic or output of the program. These is almost the same as renaming identifiers only on some values and it ca be seen as sub method of OM_05_L but it is mentioned in very few articles and probably requires a little bit more knowledge to change than identifier rename.

- OM_07_S – Donaldson et al. [39] – "Reordering independent lines of code" – Reordering lines of code for which the order does not make any difference. These include changing the order of variable declarations, changing the order of statements within code blocks like functions, reordering of code blocks or functions, reordering inner classes, and so on.
- OM_08_S – Grier [54] – "Adding redundant lines of code" – Adding code that is not used or adding code that is doing "nothing" (like a=a;) or is unreachable, for example: adding unnecessary variables, adding unnecessary function calls, adding unused initializations, adding unused functions, and so on. In Reference [163, p. 137], a list of various redundant operations that can be done is given. Function tunnelling and grouping methods mentioned in Reference [103] are included. The same principle can be used for any kind of statement, such as when adding redundant statement to some extent where OM_07_S is also performed.
- OM_09_S – Donaldson et al. [39] – "Splitting up lines of code" – This method includes splitting a single line of code into multiple lines, splitting one function into multiple ones, splitting the code into multiple classes, or splitting one class into multiple classes, or just moving inner classes into separate files, and so on. Every splitting is in fact adding some redundant code, but primary intent was to split.
- OM_10_S – Faidhi and Robinson [42] – "Merging lines of code" – There are two sub methods that are often divided in the articles but both can be seen as merging:
  - OM_10a_S – Faidhi and Robinson [42] – "Merging lines of code" – Merging of variable initialization and declaration, merging different classes or multiple functions into one, and so on.
  - OM_10b_S – Whale [170] – "Replacing the procedure call by the procedure body" – Special case of merging is replacing the procedure call by the procedure body.
- OM_11_AS – Whale [170] – "Changing of statement specification" – Here are three sub methods given because they describe different operation but all of them are quite similar and can be grouped together under the unified name:
  - OM_11a_AS – Whale [170] – "Changing the operations and operand" – Changing the order of operands, or changing the logical operations with equivalents (like x!=y to !(x==y) or x<y to x>=y), changing variable scope from local to global and vice versa, and so on. But this method also includes changing the modifiers and datatypes, which most authors specify as different methods, so a special ID was given to this method.
  - OM_11b_AS – Prechelt et al. in [134] – "Altering modifiers" – Channing the modifiers for variables, functions, and classes that do not change the behaviour of the code (like private to public).
  - OM_11c_AS – Whale [170] – "Datatype changes" – Change of datatypes like float to double or vice versa.
- OM_12_AS – Faidhi and Robinson [42] – "Replacing control structures with equivalents" - Replacing the program control structures with equivalent control structures (for instance while instead of for, switch instead of if, function instead of procedure) without changing the logic.
- OM_13_LG – Grier [54] – "Simplifying the code" – Removing lines of code that are not needed or that will remove functionality that is not the core functionality. This is called simplifying, because it can remove complex parts of the program that the student does not want to submit.
- OM_14_LG – Arwin and Tahaghoghi [9] – "Translation of program from other programming language" – This methods converts a program written in one programming language into another, like C to Java. It appears that in most cases this code would be found on the

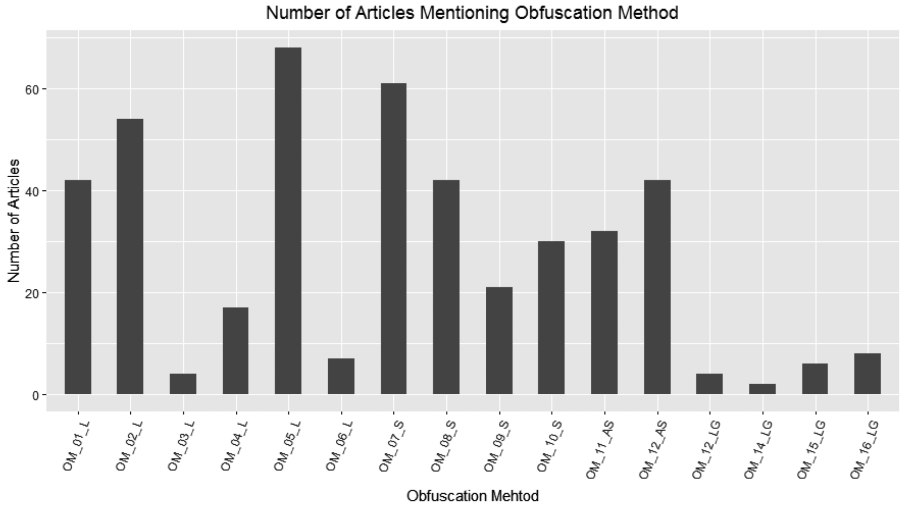

Fig. 1. Number of articles mentioning obfuscation method.

Internet but written in another language. Since often the assignment will not exactly be the same probably some logic changes are necessary so this is not only a structural change.

- OM_15_LG – Faidhi and Robinson [42] – "Changing the logic" – The most complicated of all methods where the student has certain knowledge, but does not know how to implement it all, so makes some changes in the logic and probably has some parts that are original. Also into this category falls the situation where someone is paid to build an application and the person has multiple requests for the same assignments (or has itself done the assignment) and makes small variations of the same assignment solution.
- OM_16_LG – Whale [170] – "Combining copied and original code" – The student can copy parts and combine it with their original part, and so is in some way changing the logic and this cannot be simply considered just a structural modification. At the same time, the logical change can be simple. This category is left as the last obfuscation method, since it can (but must not) be the most complex one.

Figure 1 shows the 16 obfuscation methods in graph form. It shows more clearly that the obfuscation methods that include logic change are rarely mentioned (fewer than 10 articles mention them). This is probably because the detection of such cases is much harder and the articles do not deal with them rather than that the students do not use them.

Also, it is interesting that some lexical changes have only a small number of mentions like the method "Translation of program parts" and "Changing constant values." For the "Translation of program parts," one can conclude that it is mostly covered by other obfuscation methods so it is not very important to be analysed separately. The same goes for "Changing constant values," because only a small percentage of lines of code are constants, so they probably do not have much impact on the result and the authors do not bother with them.

One can see what the most-mentioned methods are, which are mentioned by more than 60 different articles. These are the "Identifier rename" and "Reordering independent lines of code." The two methods are 57 times mentioned together in articles. Also, from these 57 articles, 53 of them are using some kind of a plagiarism detection tool. These could be explained as follows. Probably the two methods are the most used tricks by students and because of this most plagiarism detection tools deal with them and try to eliminate that obfuscation.

From the graph, one can further see which methods are next mentioned the most, and from this one can conclude which methods are more used by the students and were identified by the different authors. Note that, while frequent mention of a method suggests high usage from the students, infrequent mention does not necessarily mean that is infrequently used by the students; rather it means that it was not the focus of many articles. One could further analyse the relationships of the obfuscation methods, but these are beyond the scope of this article.

Except for the 16 mentioned obfuscation methods, there is a special method of plagiarism attempt when students do not perform any obfuscation and just copy whole or parts of a solution, and this is then usually marked as a level zero obfuscation method.

When looking at the aforementioned modifications to the different obfuscation methods, one can say that many of them are actually code refactoring methods. Experienced programmers could use the knowledge about refactoring to help them hide plagiarism. More troublesome, a modern integrated development environment (IDE) makes refactoring easy in some cases. For example, in Java programming language, an IDE makes it easy to switch between a loop iterating through a list to a lambda expression, which is an easy way to obfuscate plagiarism.

3.3 Source-code Plagiarism Detection

Once it is known what source-code plagiarism is and the different obfuscation methods that are used to hide plagiarism are known, then source-code plagiarism detection can be defined as "*a process where someone tries to identify plagiarised source-code regardless the various obfuscation modifications performed on the source-code.* And if the process takes place in academia, then it can be said that *source-code plagiarism detection in academia is a process where a teacher tries to find real plagiarized source-code solutions submitted by students even though they had used various obfuscation methods to hide their plagiarism."*

It is known that plagiarism detection is a tedious process and that with large groups it is impossible to do it manually, and so, as reported in almost every article of the 150 articles that are analysed, some plagiarism detection engines (or short tools) are used to find plagiarism. But one should be careful to remember that tools do not find plagiarism; rather, they find similarities that exist between documents, in our case documents with source-code. "There are many innocuous and suspicious reasons why student code might be similar …" and "with code reuse actively promoted, the distinction to students becomes further blurred" [105]. So even if tools are called plagiarism detection tools, they are actually similarity detection tools. One should always be reminded what Donaldson et al. said [39]: "*It is certainly safe to say that neither the detection system described in this paper nor any other detection system will find all occurrences of plagiarism. There is an inherent tradeoff between a highly discriminatory system, which overlooks some instances of cheating, and a less discriminatory one which flags many dissimilar programs."*

It is thus only natural to ask what tools are there and which ones are the best at finding similarity that are not fooled by various obfuscation methods performed by students. From the 150 articles, 131 articles use, compare, or develop a new tool or a new algorithm for plagiarism detection. From the 131 articles, 120 are developing a new tool. From the 120 articles, only 55 authors compare their new tool or algorithm to others, so 65 articles (more than 50%) are not doing any comparison. This is a problem, since it is really hard to decide what to use and objectively say what is better. This comes partially from there being no standardized way to compare. Until recently, there were no standard datasets that could be used (more detail in Section 3.7). To find out which tool to use one could start by looking at review articles about source-code plagiarism detection.

There are 12 review articles from the 150 that are analysed, and here are short descriptions of what each review article is doing. The articles are listed in chronological order of publication. Parker and Hamblen [131] in 1989 compare plagiarism detection algorithms based on software

metrics they are using and only compare the early attribute counting systems. It is a good comparison if one is interested in such early algorithms and the beginnings of plagiarism detection. Whale [170] explains different kinds of comparison measures (precision, recall, sensitivity, selectivity, excess detections, and performance indices) and gives comparisons of five tools using different metrics. These tools are old and are not used anymore, but the comparison technique described in Reference [170] is still of value. Texler [163] gives an overview of the issue of plagiarism and copying from a wide variety of angles and gives some factors that encourage or discourage students from cheating, why they cheat, and so on. Verco and Wise [167] in 1996 compare three plagiarism detection tools (Plague, SIM, and YAP) based on Whale's 1990 [170] metrics. Lancaster and Tetlow [94] in 2005 compare three metrics (word pairs, tokenised longest common substrings, and compression) for finding collusion in source-code. They performed six different obfuscation methods and compared the results. Mozgovoy [116] has a good description of various algorithms used in the plagiarism detection tools and principles for comparisons. Chen et al.'s [23] review article talks about four different types of detections algorithms. Hage et al. [55] compare five tools for plagiarism detection (JPlag, Marble, MOSS, Plaggie, and SIM), giving detailed qualitative comparison of features in 10 categories. They also performed two experiments: first, where they refactored Java classes (tried to hide plagiarism) in 17 different ways to test sensitivity and, second, another experiment with real cases to test performance for top-10 results for each tool. Martins et al. [108] compare and describe nine plagiarism detection tools (CodeMatch, CPD, jPlag, Marble, MOSS, Plaggie, SIM, Sherlock Sydney, and YAP) with feature comparison in eight categories and performance test using eight levels of plagiarism. Flores et al. [46] present the results of six algorithms performing on the Detection of SOurce COde Reuse (SOCO) PAN track at FIRE competition using the precision, recall, and F1 comparison measures. Misic et al. [114] did feature comparisons on 10 plagiarism detection tools and did two experiments on three tools (MOSS, JPlag, and SPD). For the first experiment they performed different obfuscations and looked which tool found which obfuscated part of the code. The second experiment was on three real courses and showed similarity percentage in comparison to the top-20 percentage of all indicated matches for multiple assignments. Novak [122] performed a qualitative feature comparison of five tools (MOSS, JPlag, SPLaT, SIM, Marble, Plaggie, and Sherlock Warwick) and described various algorithms used in these tools.

As it can be seen from the short description of the review articles, most of them are not very useful to someone strive to find a good tool to use, some because they are old, and some because they review algorithms rather than finished tools that can be used right away. Articles [55], [108], [114], and [122] may therefore be considered the most useful, but one should keep in mind there could be other good articles that did not make it into the analysis. For the first fast elimination the qualitative feature comparisons are most useful. For example, if a tool needs some kind of transfer student assignment from a campus server to other server outside the campus and that is not allowed according to, for instance, university data protection act, then such tool that works as an online services is of no use even if it is the best.

If interested in implementing one's own tool, then the newer papers describing the algorithms can be of use, but then one should not only use review articles, although they can help to begin with. If interested in comparison metrics, then the old review articles should not be discarded, because the comparison metrics introduced by Whale in [170] are still used today.

3.4 Source-code Plagiarism Detection Tools

As already stated, 120 articles (because of the large number the articles will not be listed) from the 150 report the development of some new tool or algorithm for plagiarism detection. Some authors have maybe two versions of the same tool, but this is rare. Figure 2 illustrates how many tools are

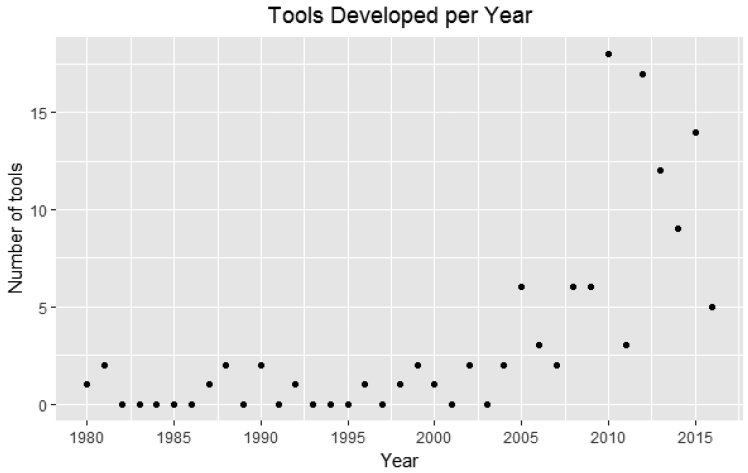

Fig. 2. Number of tools developed per year.

Table 2. Overview of "Top-Five" Tools Found in the Articles

Tools	Last Year	First Year	compared	used	developed
SIM-Grune	2014	2010 in [27]	**4**	2	NA
Plaggie	2016	2006 in [3]	**6**	0	1
Sherlock-Warwick	2016	1999 in [75]	**4**	4	1
MOSS	2016	1999 in [16]	**29**	9	NA
JPLAG	2016	2002 in [134]	**37**	5	1

developed each year. It can be observed that until 2005 up to 2 tools were developed per year, and then the interest starts and from 2010 there is a massive production of tools (in 2010 alone 18 tools were developed).

In spite of the large production of tools in recent years, most of the tools are not available to the public, they are used only by the authors that developed them and are mentioned in only one article. We note that 65 of these tools are not compared in the literature, so the quality is questionable.

Table 2 gives an overview of tools that are considered as "top-five" tools, which means they are compared at least four times. Tools that are compared at least two times are as follows: Ottenstein, Donnaldson, Accuse, Plague, CCFinder, CodeMatch, Sherlock-Sydney, PMD's CPD, SID, Marble, SIM, and YAP3. Table 2 shows the last year each tool was mentioned, the first year it was mentioned in the analysis articles and the article number, and how many times it has been compared with other tools, used, or developed. Table 2 is ordered by year in which the tool was last mentioned and if the year is equal then by the total number of references to it. The entry NA in developed means that an article from an author that describes the developed tool was not included in the final 150 articles.

From Table 2, it is concluded that MOSS and JPlag are the top-two tools (since they are compared the most), followed by Plaggie, Sherlock Warwick, and SIM from Grune. If one compares the list of these five tools, then it can be seen that they and a few others were covered by the four review articles that were suggested to be read first if interested in finding a tool. A list of articles that develop, use, or compare the top-five tools is presented in Appendix B.3.

Table 3. Feature Overview of the Top-Five Most-mentioned Tools

Tools	Number of mentions	Open source	GUI	Available offline	Available from
JPLAG	43	YES	YES	YES	https://jplag.ipd.kit.edu
MOSS	38	NO	YES	NO	https://theory.stanford.edu/~aiken/moss/
Sherlock-Warwick	9	YES	YES	YES	http://warwick.ac.uk/iasgroup/software/sherlock
Plaggie	7	YES	YES	YES	https://www.cs.hut.fi/Software/Plaggie
SIM Grune	6	YES	NO	YES	https://dickgrune.com/Programs/similarity_tester

Table 3 presents a short overview of four features for our "top-five" most-mentioned and compared tools with links to where each can be downloaded or used. Table 3 is mostly based on the feature comparisons from review articles [55], [108], and [122]. All "top-five" tools are free to use and, as it can be seen from Table 3, most are open source, so it is no surprise that they were most compared. There could be a perfect tool, but if it is not available for usage, then it is of limited use to the researchers and has no use for teachers. These tools support the following programming languages: (1) *JPlag*: Java, C#, C, C++, Scheme and natural language text; (2) *MOSS*: C, C++, Java, C, and so on (full list available on link in Table 3); (3) *Sherlock-Warwick*: Mainly build for Java, C and natural language (has no parser so every programming language can be analysed); (4) *Plaggie*: Java; and (5) *SIM from Grune*: C, Java, Pascal, Modula-2, Lisp, Miranda, and for natural language.

Regarding the tools, some have the same name, like SIM from Githcel and Tran from 1999 and SIM from Grune 1989, Sherlock from Warwick and Sherlock from Sydney, or PMD's CPD and CPD from Kuo et al. [90]. The authors use one and reference other. Sometimes this can be noticed based on the description, for example, authors say that Sherlock has no graphical interface, and that it is Sherlock Sydney. Sometimes this is not possible, and one just has to trust that the authors referenced the right one.

One notable article from Cebrian et al. [22] reports a benchmarking tool for plagiarism detection engines. They have built a tool for generating test cases for the APL2 programming language but only tested one tool. While the idea is interesting the benchmarking tool is still not very useful because of the chosen programming language. As one of the analyses shows, C and Java are the most used programming languages for which plagiarism detection tools are built. But the authors of Reference [22] claim that they plan to support Java and C programming languages in the future.

3.5 Algorithms Used for Source-code Plagiarism Detection

It has been shown which are the most-researched tools, but if one is interested in building a tool or comparing it, knowledge about algorithms used can be useful. The analysis of the algorithms is not performed in depth, because it goes out of scope of this article. What is presented in this subsection is what kinds of algorithms are there and an attempt to classify the algorithms. In Table 4 algorithm types that were used in the articles are given.

The analysis was done on the 120 articles that develop a new tool or algorithm. Table 4 has five columns: name of the algorithm (type), last year and first year this algorithm (type) was used, number of articles that have used this algorithm (type), and how many of these have used tokenization. Table 4 is divided in two parts: The first part is the algorithm types that are commonly known, and the second part adds some new categories that were not identified in review articles like Reference [116]. The number of articles using an algorithm type should not be taken for granted,

Table 4. Overview of Algorithm Types

Algorithm Type	Last year	First year	Number of articles	Tokenized
Based on style	2016	2011	5	2
Based on semantics	2013	2010	7	5
Text based	2016	1996	9	5
Fingerprint based	2015	2005	11	4
Based on attribute counting	2015	1980	25	6
Structure based	2016	1980	25	13
Based on string matching	2016	1981	26	17
New identified categories				
Watermarking	2013	2005	2	0
History based	2016	2013	2	0
XML based	2012	2010	3	2
Compiled code based	2015	2006	5	2
Compression based	2010	2004	6	5
Graph based	2015	2005	10	2
Clustering based	2015	2005	11	7
Based on nGrams	2016	2006	15	9
Based on Trees	2015	1988	24	8

because in some articles it is not clear in which category to put an algorithm. But the numbers give some indication what are the most researched algorithms.

Identified algorithms or algorithm types are as follows: *Attribute Counting*, *Fingerprint*, *String Matching*, *Text Based*, *Structure Based*, *Stylistic*, *Semantic*, *nGram*, *Trees*, *Graphs*, *History Based*, *XML based*, *Compiled Code Based*, *Compression Based*, *Only GST*, *Only RKR*, *RKR with GST*, *Winnowing*, *Latent Semantic*, and *Watermarking*. A list of articles that were identified as using a particular algorithm or algorithm type is presented in Appendix B.4.

From Table 4, it can be seen that all algorithm types have been used recently, and some of them were invented in the 1980s. Techniques like attribute counting are considered old and are mostly not used alone. Whale [170] showed that structure-based approaches are much better than attribute counting, but there is still research dealing only with attribute counting [153]. Most of today's plagiarism detection tools combine more types of algorithms. For example, Hage et al. [56] state "*the structural comparison on token streams would not be sufficient to detect plagiarism well enough. Therefore, we looked at a number of comparisons that are closer to the computational structure of the submission: the call graph.*" Also Table 4 indicates that most algorithms use tokenization. To be more precise, 53 articles from the 120 also use tokenization. So it can be said that tokenization forms one very important part of a large proportion of algorithms.

The biggest difference between the older algorithm types and the new types is that the older ones focused only on the source-code in pure form. They tried to extract metrics from the raw source-code and compare them. In the new types, additional information is analysed. To explain what this means, a short description of how each of the new types of algorithms works is given below. Note this is not an exact specification, and some algorithms are quite complicated, so a separate review dealing only with algorithms could be performed.

Watermarking algorithms add secret information (called a watermark) that is not visible through the standard view of the source-code in an IDE, but it can be recognized in a binary or hexadecimal view of source-code. In this way the detection is done based on the added watermarks.

Table 5. Top Used Comparison Metrics

Comparison Metric	Last Year	First Year	Comparing	Developing
Selectivity	1996	1990	2	NA
Excess Detections	1996	1990	2	NA
Performance Index	1996	1990	2	NA
Sensitivity	2011	1990	5	3
Speed	2015	1992	5	11
F1	2016	2012	10	10
Fbeta	2016	2012	12	13
Precision	2016	1990	23	26
Recall	2016	1990	24	26
Qualitatively	2016	1980	47	84

History-based algorithms track the data of the user from some repository, or track their progress from IDE activity logs, to find out whether there is suspicious activity. Compiled code-based algorithms try to use the compiled code and not the original source-code. Compression-based algorithms try to compress the code and try to find some similarities between the compressed codes. Clustering-based algorithms add another step after the similarities have been calculated to create clusters of documents based on the similarities. Graph-based, nGrams-based, Trees-based, and XML-based algorithms try to transform the source-code into some new form such as a dependency graph or an XML representation to perform the detection.

3.6 Comparison Measures

When searching for a tool, knowing the implemented algorithm is not very useful information, since one is more interested in (experimental) comparisons. From the 120 articles, only 55 authors compared their new tool or algorithm and only 8 articles actually performed any comparisons. A list of the articles is given in Appendix B.5.

To be able to understand or perform such experimental comparisons, it is necessary to know about the different comparison measurements (metrics) that are used. In Table 5, the metrics used for comparison are listed. Table 5 has comparison metric name, last and first year the metric was used for comparison (column "Comparing"), how many times the metric was used, and how many times it was used by an article that is developing a new tool and used to evaluate the tool (column "Developing"). Note that Comparing is not subset of Developing and vice versa. Table 5 is sorted by the last year and then how many times the metric was used for comparison.

From Table 5, it can be seen that the most used comparison metric is no numeric metric at all as one would expect, which means that authors are doing the comparison qualitatively. Basically they perform the detection with their tool and another tool (if they do comparison) and then state what is found by one or the other and give explanations as to why this happens. Often in such cases, the dataset used is their own, which has specific plagiarized pair(s) that were found by their tool and were not found by the other compared tools. This kind of analysis may be very biased, especially when there is no comparison with other tools, although qualitative analysis is very helpful to answer the "why" question of why some tool find or do not find specific plagiarized matches. Note that we do not want to say that all qualitative comparisons are biased or that they are not necessary. On the contrary, we think that qualitative comparisons are necessary but are more likely to be biased, but a more detailed analysis of the qualitative comparisons is out of the scope of this article.

Thus quantitative analysis is recommended to enable objective comparison, which also means that using a standardized dataset (more details in the next subsection) is advisable.

Excluding qualitative comparisons, the most used metrics to measure system qualities and used for comparisons are the oldest ones, namely *precision* and *recall*. The next in line is F_β (mostly written *F-beta)*, which is a metric calculated based on precision and recall, followed by *F1*, which is a special case of F-beta where equal importance is given to precision and recall. Number of articles using the top-three comparison metrics or their combinations are as follows: Precision (23), Recall (24), F-beta (12), Precision and Recall (23), Precision and No Recall (0), ecall and No Precision (1), Recall and Precision No F-beta (11), Recall and Precision and F-beta (12).

The previous list shows how many articles use which metrics of the three, although logically all that use F-beta use also precision and recall. After that, there is the measurement of speed, meaning how fast a tool performs the detection. Speed will not be further explained, since it is not really the measure of system quality, although it is important and often one does not want to wait multiple days to get results. Next is Sensitivity, which can mean two different measures as it is described below. The last three are Selectivity, Excess Detections, and Performance Index. A list of articles that do comparisons grouped by the used comparison metric (these metrics are discussed in more detail below) is presented in Appendix B.6.

Before going into describing the above-mentioned metrics, some terminology needs to be established (based on articles mentioned in previous list). Similarity tells us how similar two programs are from 0% to 100%. In academia similarity is normally similarity between assignment solutions form two students. Matches (also called pairs) represent two solutions with their similarity. There are four categories of matches and usually it is counted how many of each matches were put into which category by the tool. To ensure each match is "real," it must be confirmed by someone expert (usually the teacher). The four match categories are as follows: *True positive* (tp) or number of *found plagiarized matches* (match that is marked by a tool as plagiarized and it is plagiarized), *False positive* (fp) or number of *found non-plagiarized matches* (match that is marked by a tool as plagiarized and it is in fact not plagiarized), *True negative* (tn) or number of *not found non-plagiarized matches* (match that is marked by a tool as not plagiarized and it is not plagiarized), and *False negative* (fn) or number of *not found plagiarized matches* (match that is marked by a tool as not plagiarized but it is in fact plagiarized). From the four mentioned match categories, other match categories can be created most interesting combinations are as follows: tp plus fp is the number of *indicated matches*, tp plus fn is number of *plagiarized matches*, fp plus tn is number of *non-plagiarized matches*, and tp plus fp plus tn plus fn is number of *matches*.

The similarity that the tool uses to tell what is or is not plagiarism is called the *cut-off threshold* or just *threshold* and can almost always be set by the teacher. Since there can easily be thousands of matches, the expert usually checks only the true positives and the false positives that are identified by the tool at a defined threshold. It is believed that all the others are true negatives, which introduces certain problems that are discussed below.

3.6.1 Metric Description. Whale [170] first used precision and recall to measure performance of plagiarism detection tools. Although he calculated performance index, only precision and recall were accepted by future researchers.

Precision (P) is the number of found plagiarized matches divided by number of indicated matches or tp / (tp + tn). *Recall* (R) is the number of found plagiarized matches divided by number of plagiarized matches or tp / (tp + fn). Precision and recall have values between 0 and 1, where 0 is the worst and 1 the best. High Precision ensures that time is not wasted on looking at false positives (not plagiarized cases). High recall ensures that not many plagiarized matches go unnoticed.

As can be seen, high recall generates false negatives that, as stated before, are often not confirmed so it is impossible to calculate if the number of exact plagiarised matches is not known. One solution is to use a standardized dataset where someone has already identified all real plagiarized matches so it is known upfront which are which. A second solution is to take source code from one or more original programs that are intentionally modified (thus simulating plagiarism) in different ways and so generate a dataset that is then used for comparison with known plagiarised pairs. But if research is performed on real data, if the above-mentioned options are not suitable, then the only option is to estimated true positives. To improve the estimate, usually the detection is performed by different tools and then what is detected by all tools and confirmed as true positive is considered the whole set of true positives and everything else is true negative. It is the idea that multiple tools with different algorithms will not miss many plagiarized matches as used by Verco and Wise [167]. In a perfect system precision and recall would be 1 so the number of indicated matches is in fact the number of plagiarized matches.

When recall and precision are known, F-beta can be calculated, as explained by Acompora and Cosma [2]: $F_\beta = (\beta^2 + 1) * (\text{Precision x Recall}) / (\beta^2 \text{ x (Precision + Recall)})$.

From this F1 can be calculated, which gives equal importance to precision and recall as F1 = (P x R) / (P + R). If β smaller than 1, then it gives more importance to precision, and if β larger than 1, then it gives more importance to recall. There are two other measures mentioned in Reference [2], which are *specificity* and *accuracy* both rarely used. "*Accuracy is the proportion of files labelled as true (both true positives and true negatives), and Specificity is the true negative rate (i.e. files correctly identified as negative)*" [2]. In other words *accuracy* is number of indicated matches divided by number of matches and *specificity* is number of found not plagiarized matches divided by number of not plagiarized matches. Specificity is kind of inversion of recall, in recall interest is in plagiarized matches and in specificity at non plagiarized matches.

3.6.2 Graphs. Except for the mentioned metrics, there are three graphs based on which interesting analysis can be done. The first graph is like the one used, for example, by Prechlet et al. [134], who plotted the distribution of similarity values found among plagiarized matches and among non-plagiarized matches with box and whiskers. Such a graph shows nicely how the different similarities are distributed over the dataset, and from this one can see what similarity most pairs have. Similarly, one can plot other match categories (not just plagiarised and non-plagiarised matches), such as the distribution of indicated matches over the similarity values, and one could also limit the graph to the top-*n* percentage if only interested in the similarities in the high ranking matches, as performed in Reference [114].

Another graph done by Perchlet et al. [134] is precision or recall threshold tradeoff. Such a graph shows how recall or precision changes when the cut-off threshold is gradually increased. Such a graph can also be done for the F measure, as done, for example, by Đurić and Gašević [166].

The third graph is the one used, for example, by Arwin and Tahaghoghi [9] or Cosma and Joy [33], the interpolated precision-recall graph. The graph shows "the interpolated precision at a particular recall level is the highest precision observed at that or any higher recall level. Interpolated precision-recall is usually shown at the eleven 10% steps of recall from 0% to 100%" [9]. To create such graph, recall is set on the x axis, and on the y axis the corresponding precision that was observed at this recall level. Such a graph shows how precision lowers as the recall increases.

3.6.3 Derived Measures. In Reference [9] are the two measures derived from precision and recall: R-precision and precision@n. "R-precision is the precision at the R-th program on the list, where R is the number of correct answers for the query; Precision@n is the precision at the *n*th program on the list." It is sometimes advisable to do comparisons on top-*n* detections. For example (as explained by Hage et al. [55]), when one has a large collection of real submissions and cannot

Table 6. Dataset Category Usage by Different Review Articles Included

Dataset Type	All	Using	Developing	Comparing
Personal Student dataset	96	87	80	41
Personal Generated dataset	45	43	39	27
PersonalGenOrStud dataset	3	3	3	0
SOCO dataset	5	5	4	4
ICPC dataset	2	2	2	1
OtherDataset	6	4	4	3

examine all manually to confirm plagiarized cases, or when different tools are compared and each has different way of computing then the similarity.

The last "metric" that derives from precision and recall that is mentioned is the optimal cut-off criterion defined [115]: "...as the cutoff criterion value at which TPDP is maximized. The total plagiarism detection performance (TPDP) measure involves both precision and recall; it uses a parameter m, which indicates the importance of recall against precision for a particular task of plagiarism detection and it is defined as follows: TPDP = P+m∗R, m ≥ 1."

3.7 Dataset Analysis

To have comparable results with other research (except for the metrics), the right dataset is also important. Standardized datasets, like ACM International Collegiate Programming Contest (ICPC) and SOurce COde re-use (SOCO), used for plagiarism detection comparison are rare (Table 6).

Most used are the personal datasets from real students followed by personally generated datasets (Table 6). Generated datasets are datasets where usually there is an original program that is put through a series of modifications that should imitate plagiarism. These two categories are mostly used. Sometimes is not clear from the article [118, 123, 152] whether the dataset is a real dataset from students or generated so it is called "personalGenOrStud." When taking into account all 150 articles there are 7 articles [23, 49, 61, 116, 122, 164, 174] not using any dataset and do not perform any kind of evaluation for example review article. There are 6 articles [1, 3, 10, 29, 128, 163] where it is unknown what kind of dataset it is.

Table 6 presents the different datasets' usage over the total 150 articles (All), the 131 articles comparing or using or developing a tool (Using), the 120 articles developing a tool (Developing), and the 64 articles comparing tools (Comparing). Note that the numbers in Table 6 do not add up, since some articles use multiple datasets. The most interesting are the ones that do some comparing, and below a list of articles by the dataset type is given. It should be mentioned that some articles use more than one dataset, so they are mentioned under multiple dataset types in the list. It can be seen from Table 6 that there are three other dataset types: Reference [9] uses a personal student dataset and some files from the web, Reference [119] uses the dataset from Malphol from the University of Karlsruhe,[1] and Reference [44] uses a personal student dataset and the Google Code Jam dataset.[2]

Articles that compare of tools grouped by dataset type are presented in Appendix B.7. The five categories that were identified are as follows: (1) *Personal Student Dataset*, (2) *Personal Generated Dataset*, (3) *SOCO*, (4) *ICPC*, and (5) *Other dataset*.

[1]It was available on http://www.ipd.uka.de:2222.
[2]https://code.google.com/codejam/contest/1460488/scoreboard?c=1460488#.

There is one dataset that is not personal and that is used more than one time for comparison. That is the SOurce COde re-use (SOCO) dataset[3] from the international PAN@FIRE 2014 competition. The SOCO dataset is divided into training and test collection, where training collection consists "of university student assignments that contains source code re-use cases that were manually detected and reported by the professors" [45]. The test corpus [46] "has been extracted from the 2012 edition of Google Code Jam Contest."[4] The training collection has the benefit that it is labelled manually by three experts as shown in Reference [46], where also more details about the datasets can be found. These SOCO dataset is a good step into standardization of collections on which the plagiarism detection can be performed to create comparable objective results. If one is interested in cross-language plagiarism detection, then the CL-SOCO dataset[5] can be of use, and details about the dataset can be found in Reference [47] and was used in References [60, 141].

Another open dataset available used by Ji et al. [68] and Lee et al. [98] was the ACM International Collegiate Programming Contest (ICPC),[6] but this is not a dataset in which it is known what the plagiarized cases are, as is the case for SOCO. It is a very large dataset, as is the Google Code Jam.

Sometimes real datasets are useful, so when talking about real personal student datasets that are used for comparisons, then in most cases one course is used. Specifically, 20 articles use data from one course like [33, 143, 154], 2 articles use data from two courses ([101, 167]), 4 articles use three courses ([25, 73, 114, 172]), 1 articles used five courses [83], and 1 uses six courses [75]. For 13 articles, it is unknown how many courses they used. Sixteen courses are at the undergraduate level, and there were 5 at postgraduate level, but for others the level is not known. This is not surprising, since undergraduate-level courses are more programming oriented than postgraduate-level courses and have more students, so the detection systems are more necessary there.

These articles have from 1 to 150 assignments (when assignments from multiple years are used like in Reference [176]). But the majority use up to 6 assignments. The number of submissions ranges from 2 to over 8,000 as in References [36] and [20], but most range from 50 to 300 submissions. The analysis of institutions where personal student datasets come from is not very interesting, since most every institution is present once. Since most courses are at the undergraduate level, the program submission lengths are mostly short (50–500 lines of code). In 25 articles that do a comparison on personal student datasets, the length is unknown. There are rare cases when there are more than 600 lines of code like in References [172], [73], [20], [30], and [114].

Datasets are always programming language dependent; for example, the SOCO dataset is made for programming languages Java and C/C++. Als, the various tools are not in most cases universal; they are built to detect plagiarism in certain programming languages. In the 131 articles that develop, use, or cooperate, some tools are built for different programming languages. Lists of the different programming languages and in which article where the tool that was tested on it is presented in Appendix B.8. The first item, "All," refers to articles describing tools that were supposed to detect plagiarism in any programming language.

Table 7 shows the usage of the top-10 most used programming languages per year groups. Pascal is in the top 10, but the articles dealing with Pascal were mostly published before the year 2000. On the other side, the majority of usage for the top-3 programming languages is after 2010. One problem that arises when building a tool that is not for Java or C is that there are not many articles

[3] http://users.dsic.upv.es/grupos/nle/soco/.
[4] https://code.google.com/codejam/contest/1460488/dashboard.
[5] http://users.dsic.upv.es/grupos/nle/clsoco/.
[6] https://icpc.baylor.edu/worldfinals/problems.

Table 7. Programming Languages in which the Tools Were Tested per Year Groups

Year	Java	C	C++	Pascal	Python	PHP	C#	VB	Haskel	UnixShell
1981–1998	NA	3	1	8	NA	NA	NA	NA	NA	NA
1999–2009	11	14	10	3	NA	NA	NA	1	NA	1
2010–2016	37	32	22	1	5	4	3	1	3	2
Total	48	49	33	12	5	4	3	2	3	3

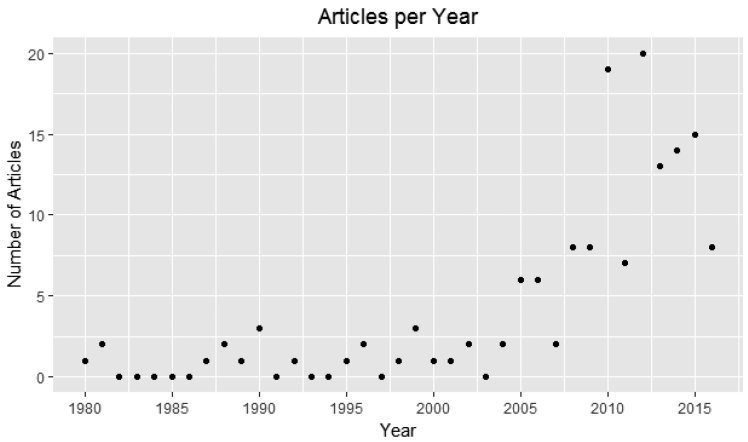

Fig. 3. Number of articles per year.

to compare and maybe no standardized datasets. So more research should be done in providing tools for multiple languages. Table 7 shows only what was tested; the majority of tools support only few programming languages if not one. Currently, MOSS has the best offer of languages, except for the ones that can be used on all languages that do not parse the code for detection.

3.8 Bibliographic Analysis

This section analysis was performed based on bibliographic information. Figure 3 presents the number of articles that were selected per year, and from it can be clearly seen that until 2005 there were no more than three articles per year. This tells us that research interest increased in 2005, and then the number tripled in 2010 when the interest escalated in the field of source-code plagiarism detection in academia. At that point, massive production of plagiarism detection tools started, because the community recognized that automatic plagiarism detection tools are beneficial. Although the most interest has been in the recent years, the most-cited articles are published before 2005.

The question is where to search for articles. As described in the protocol, five databases were used, so an analysis was done on the articles and where they are indexed. Since the analysis is not very interesting until 2005 because of the small number of articles, only the years from 2005 and later are analysed, when interest in the field increased. The first thing that was noticed is that SCOPUS holds most of the selected articles and that Science Direct has the lowest number. But there is also a ratio problem of how many articles were found and how many were actually selected. Table 8 shows how many articles were found by each database when queries were performed, how many were selected, how the ratio was found and selected, and how many unique articles were found by each database that were not found by any other database. From this, it can be seen that

Table 8. Overview of Articles per Database

Database	Found articles on last search	Selected articles	Ratio found/selected	Unique articles	Top cited	Ratio top cited/selected	Ratio top cited/found
SCOPUS	1,887	133	7.0%	27	21	15.8%	1.1%
ACM	518	60	11.5%	10	14	23,3%	2.7%
WOS	1,354	65	4.8%	3	12	18.5%	0.8%
IEEE	623	54	8.7%	2	5	9.2%	0.8%
SD	221	7	3.2%	1	1	14.3%	0.4%

SCOPUS finds the most articles and the most unique articles, and the ratio found against those selected is better than in WOS and SD; on the negative side, SCOPUS has the most number of articles retrieved. What is clear is that Science Direct yields poor results and ACM gives the best ratio overall. Also, Table 8 presents how many top-cited articles are found in a database and the ratios cited over selected and cited over found articles. SCOPUS, WOS, and IEEE all have their advantages, while SCOPUS finds the most articles, so it takes the longest to be processed, as with WOS, but they both, once selected, have a good ratio of top-cited articles. IEEE has the worst ratio of selected and top-cited articles, but it takes the researcher less time to process the retrieved articles. Also, three of the top-cited articles [131, 170, 172] are only available in SCOPUS.

The 12 top-cited articles with more than 100 citations are in Google Scholar (GS), and the citation counts from WOS and SCOPUS are also available. The reason GS was chosen is that some papers are not available in WOS and/or SCOPUS, and even though they are cited by scientific papers and some citation numbers exist, they are not easy to be extracted. The downside of using GS is that it includes technical papers in the citation count, so it is probably not the best measure; nonetheless, it gives a good overview of the major papers to begin with. The 12 articles are as follows: Reference [134] (L. Prechelt, G. Malpohl, and M. Philippsen 2002) with 549 GS, 129 WOS, and 231 SCOPUS citations; Reference [173] (M. J. Wise 1996) with 325 GS, 9 WOS, and 161 SCOPUS citations; Reference [25] (X. Chen, B. Francia, M. Li, B. McKinnon, and A. Seker 2004) with 259 GS, 96 WOS, and 143 SCOPUS citations; Reference [75] (M. Joy and M. Luck 1999) with 198 GS, 63 WOS, and 105 SCOPUS citations; Reference [53] (D. Gitchell and N. Tran 1999) with 196 GS, 36 WOS, and 93 SCOPUS citations; Reference [131] (A. Parker and J. O. Hamblen 1989) with 150 GS, 40 WOS, and 78 SCOPUS citations; Reference [42] (J. A. W. Faidhi and S. K. Robinson 1987) with 138 GS, 41 WOS, and 61 SCOPUS citations; Reference [167] (K. L. Verco and M. J. Wise 1996) with 131 GS, 9 WOS, and 13 SCOPUS citations; Reference [170] (G. Whale 1990) with 124 GS, 23 WOS, and 49 SCOPUS citations; reference [54] (S. Grier 1981) with 119 GS, 11 WOS, and 52 SCOPUS citations; Reference [172] (M. J. Wise 1992) with 114 GS, 3 WOS, and 56 SCOPUS citations; and Reference [39] (J. L. Donaldson, A.-M. Lancaster, and P. H. Sposato 1981) with 105 GS, 8 WOS, and 56 SCOPUS citations.

It can be seen that the most-cited articles were published before 2000. Of course, the older the article the more chances are for it to be cited, so this is not the most accurate number. The number would be also different if the WOS or SCOPUS citations would be used. Because of that, if articles with more than 20 citations would be included, 25 more articles would be added that are in descending citation order as follows: [67], [72], [98], [20], [9], [3], [32], [33], [16], [36], [146], [115], [171], [55], [29], [117], [105], [158], [116], [19], [143], [66], [68], and [22].

To see what the most-used words in the selected articles are, word analysis is performed using a corpus of words from titles, keywords, and abstracts was created. From the corpus of words, a term document matrix was created that has the different words and the occurrence frequency of

each individual word. From the matrix, a ranking was created with words that have more than 30 occurrences. The top 10 words and their occurrences were as follows: plagiarism, 642; detection, 389; code, 367; source, 337; programming, 318; similarity, 200; computer, 179; students, 159; program, 151; and codes, 136. This confirms that the chosen keywords—"plagiarism," "source code," "similarity," and "program"—were fine, but the word "application" (for example) is not well chosen; it would be better to use some other word instead like "computer," "student," or "detection." But, again, the question is how many results would be retrieved. Finding the right balance of retrieved and relevant articles is not easy. The word "analysis" identified keywords for new queries done in the future, if one is interested in the topic presented in this review. It would be interesting to see if the same 150 articles could be retrieved with a different query with fewer excluded articles, but this is currently beyond the scope of this article.

3.9 Missing Articles

To complete the analysis, a few more papers need to be mentioned. The titles of these papers were not precise enough, so they were not found by queries or were simply not present in the searched databases. The best example is the paper about the Winnowing algorithm [148], which is one of the most referenced papers, but the paper was not picked up by the queries. This is simply because it does not have the combination of words that were used for searching. Specifically, the word "source-code" was missing in the article's title, keyword list, and abstract.

Articles with Google Scholar citations obtained on July 20, 2017 are (sorted by publication year) as follows: (1) *1976 - [130] (153 citations)*; this is one of the first articles that describes an algorithmic approach to the problem of source-code plagiarism detection. A program was built for plagiarsim detection in the programing language ANSI-FORTRAN and used in courses at Purdue University. (2) *1995 - [97] (36 citations)*; this is an example of an article that combined three software metrics to detect plagiarism: Halestad metrics, McCabe cyclomatic complexity, and the coupling metric. (3) *2001 - [35] (17 citations)*; to our knowledge, this is the first article that describes a systematic large-scale survey dealing with the question of source-code plagiarism. The survey was conducted in United Kingdom, including 55 high education computing schools. (4) *2003 - [148] (1027 citations)*; this is a description of a fingerprint algorithm called Winnowing that is used in the popular plagiarism detection engine MOSS. As shown in previous analysis, MOSS is in the second-most-used plagiarism detection engine. (5) *2004 - [93] (51 citations)*; the review article gives good classification of source-code plagiarism detection engines. The article describes 11 engines and gives different classifications of the engines: classification how they operate on source code submissions, classification based on programming languages, classification based on techniques used to find similarity, and so on. (6) *2014 - [48] (7 citations)*; this article gives a nice overview of the differences among collaboration, collusion, and plagiarism. The article also discusses the problem from the professor and student viewpoints. (7) *2016 - [136] (4 citations)*; this is an article that performed the largest comparison of 30 tools/techniques for similarity detection against pervasive code modification. They state that "every technique and tool turned out to be extremely sensitive to its own configurations consisting of several parameter settings and a similarity threshold. Moreover, for some tools the optimal configurations turned out to be very different to the default configuration, showing one cannot just reuse (default) configuration."

4 CONCLUSION

Source-code plagiarism detection is a growing research field, as shown in this review. This is not unexpected, since the problem of plagiarism is something that most teachers deal with. The educational system must ensure that the students who gain a diploma also gain the knowledge that is promised with it. Only in this way can society grow and new problems be solved. So every

educational institution must do their best to find those who try to gain credit based on others' results. One way to do that is for the teachers to use mechanisms for preventing plagiarism, but also to use similarity detection systems for identifying potential plagiarism if the prevention mechanism fails. As the saying goes, "trust but verify."

To help teachers, researchers try to find and verify mechanisms for prevention of plagiarism, but also they build tools for automatic similarity detection of potential plagiarism. These tools are constantly compared and improved. The tools are important, since manual plagiarism detection is a tedious job and misses many occurrences of plagiarism, especially in large classes. This review's intent is to help teachers find the right tools for similarity detection as well to help researchers in their future research. The review gives researchers an overview of the field of source-code plagiarism detection in academia and points out some places for improvement and future research.

At the beginning of this review, research questions of interest were presented. Short answers to the questions based on the discussions in this review are as follows:

Q1: *What is meant by source-code plagiarism in academia?* The answer to this question is best given by Cosma and Joy [32]: *"Source-code plagiarism occurs when students reuse source-code authored by someone else, either intentionally or unintentionally, and fail to adequately acknowledge the fact that the particular source-code is not their own."*

Q2: *What is meant by source-code plagiarism detection in academia?* Source-code plagiarism detection in academia is a process where a teacher tries to find real plagiarized source-code solutions submitted by students even though they had used various obfuscation methods to hide their plagiarism, usually with a help of a plagiarism (similarity) detection engine (often called a plagiarism detection tool).

Q3: *What obfuscation methods do students use to hide source-code plagiarism?* In the Section 3.2 about obfuscation methods, there are 16 obfuscation methods listed that are known to be used by students. In Figure 1, one can see which methods were found as the most used. The top three are "Identifier rename," "Reordering independent lines of code," and "Comments modification." Weak mentioning of the obfuscation method in articles must not mean that is not used by the students in many cases, but rather it can mean that it was not the focus of the articles. Such methods must be further researched, especially the category of methods that incorporate logic changes.

Q4: *What detection tools are used and which are the best?* Many tools have been developed, but most are not available for use. The most promising and available tools are JPlag, MOSS, and Sherlock Warwick. Based on comparison counts, these are also the most compared tools. There are other tools that in some comparisons outperform the three mentioned, but these three are always reported to deliver good results. However, it must always be remembered that tools do not find plagiarism regardless of how good they are; they only find similarity, and it is up to a human to decide what is and is not plagiarism.

Q5: *What algorithms are used in these detection tools?* There are various algorithms used in tools, and it is hard to pick out a few, although the top-three tools use RKR-GST and Winnowing algorithms and implement tokenization. The individual algorithms can be put into one or more categories, and in this review some new categories have been identified.

Q6: *What measures are used to compare the tools?* There are multiple measures available, but if one wants comparable results, then precision, recall, and F measure (more precisely F1) are the most commonly used. There are also various graphical versions of tools based on these three metrics that can help the analysis. But one should not forget qualitative

analysis, which is still the most-used "metric" that can give deep understanding of why something is happening, but it has the problem of being biased.

Q7: *What datasets are used to compare the tools?* The majority of research uses real personal student datasets or datasets that contain instances of intentional plagiarism. While standard datasets are good for comparisons, real datasets are better to find new ways of how students try to cheat and find flaws in the plagiarism detection tools and situations in which they will fail. So one should combine both types of datasets. But the problem is that there is a very limited number of standard datasets, although this is improving; for example, the SOCO dataset, introduced in 2014, is currently a good option for tools that support Java or C/C++.

Q8: *Where should one search for articles dealing with source-code plagiarism detection in academia?* The ACM database is the best place to start, since it gives the best ratio of found/selected and top-cited/selected articles. SCOPUS will cover almost all articles but will find many unrelated articles. One can also start from the top-cited articles listed in this review and then look at the most recent or highly cited articles that cite this article.

All the questions were answered successfully, and what was observed while answering them is that there are missing standard datasets, metrics for evaluation, and objective comparisons of plagiarism detection tools. Therefore, many of the selected articles are not very useful. It was also observed that much has been done for Java and C/C++ programming languages but not for other languages, and they should be better supported. One reason for high usage of Java and C/C++ could be that they are currently the most used programming languages in academia and therefore most interesting for teachers. Another observation is the term "plagiarism detection," which in combination with the word "tool" sends a wrong message. It is strongly suggested to try to use the term "similarity detection," which is better and does not suggest that the tools find plagiarism.

While detection is important, the prevention of plagiarism is even more important, and another review about prevention of plagiarism would be useful to academics. Another useful review would be about authorship attribution, because, although plagiarism detection tools find plagiarism, it is sometimes questionable who is copying from whom, and teachers would benefit from that knowledge. One new approach for author identification would be the use of agile methods with small code delivery and a source-code version control system (Git, SVN, etc.) to track assignment progress, authorship, and locations in documents where similarity begins.

To summarize, the main contribution of this review is the detailed overview of the field of source-code plagiarism detection in academia, which has identified (a) definitions of the phrases "source-code plagiarism" and "source-code plagiarism in academia," (b) obfuscation methods used for hiding plagiarism, (c) the most promising and most used similarity detection tools, (d) the most common algorithms used in similarity detection tools, (e) existing categories for classifying the algorithms, (f) methods for comparing similarity detection tools, and (g) potential future research areas.

Except for the overview, the most interesting findings while writing this review are the following. (i) There are many definitions of "source-code plagiarism" both in academia and outside it, but no agreed clear definition exists in the literature, so we have identified the most promising one and presented a new working definition for "source-code plagiarism detection." (ii) We have expanded existing categorisations of obfuscation methods. (iii) We have expanded existing categorisations of detection algorithms, which encompass the approaches taken by recently developed tools. (iv) It was surprising to see how few metrics and datasets are available for quantitative analysis, even with the 120 tools that are reported in the literature. By listing the metrics, we provide a baseline for future comparisons, and by noting the lack of datasets, we highlight the need for more to

become available to provide a solid basis for future comparison exercises. (v) We have established that most tools are made for only one language, typically Java or C, and therefore note that there is a gap in our expertise for detecting plagiarism in other languages. (vi) The set of keywords that was finally adopted for our literature search provide a basis for future reviews that will assist researchers in accessing appropriate papers and provide a basis for a longitudinal study.

A DATABASE SEARCH QUERIES

This appendix lists the concrete search queries for each database used in the study:

SCOPUS (http://www.scopus.com): *TITLE-ABS-KEY((source AND code) AND (plagiarism OR similarity)) OR TITLE-ABS-KEY ((application* OR program*) AND plagiarism)*

IEEE Xplore (http://ieeexplore.ieee.org/): *((source AND code) AND (plagiarism OR similarity)) OR ((program* OR application*) AND plagiarism)*

Web of Science (http://www.isiknowledge.com/WOS): *TI=((source AND code) AND (plagiarism OR similarity)) OR TS=((source AND code) AND (plagiarism OR similarity))OR TI=((program* OR application*) AND (plagiarism)) OR TS=((program* OR application*) AND (plagiarism))*

ScienceDirect (http://www.sciencedirect.com): *TITLE-ABSTR-KEY((source AND code) AND (plagiarism OR similarity)) OR TITLE-ABSTR-KEY((application* OR program*) AND plagiarism)*

ACM Digital Library (http://portal.acm.org) NEW: *(((acmdlTitle:(+source) AND acmdlTitle:(+code)) AND (acmdlTitle:(+plagiarism) OR acmdlTitle:(+similarity))) OR ((recordAbstract:(+source) AND recordAbstract:(+code)) AND (recordAbstract:(+plagiarism) OR recordAbstract:(+similarity)))) OR (((acmdlTitle:(+program) OR acmdlTitle:(+application)) AND acmdlTitle:(+plagiarism)) OR ((recordAbstract:(+application) OR recordAbstract:(+program)) AND recordAbstract:(+plagiarism)))*

ACM Digital Library (http://portal.acm.org) OLD: *(((Title:source and Title:code) AND (Title: plagiarism OR Title:similarity)) OR ((Abstract:source and Abstract:code) AND (Abstract:plagiarism OR Abstract:similarity))) OR (((Title:program* OR Title:application*) AND Title:plagiarism) OR ((Abstract:application* OR Abstract:program*) AND Abstract:plagiarism)).*

B REVIEWED ARTICLE CROSS-REFERENCES

B.1 Reviewed Articles

This appendix presents the list of reviewed articles. Articles are sorted by date of publication. The reviewed articles are as follows: [143], [39], [54], [42], [59], [67], [131], [139], [170], [171], [172], [163], [167], [173], [121], [16], [53], [75], [76], [72], [52], [134], [25], [71], [36], [73], [86], [94], [115], [117], [3], [9], [105], [106], [116], [147], [20], [182], [30], [32], [49], [66], [68], [69], [119], [146], [22], [40], [87], [91], [103], [160], [168], [176], [5], [19], [24], [27], [28], [41], [63], [79], [89], [99], [100], [112], [111], [123], [124], [165], [175], [177], [178], [23], [55], [74], [104], [110], [113], [127], [1], [8], [10], [15], [21], [29], [33], [34], [64], [70], [80], [82], [84], [90], [98], [107], [118], [120], [133], [152], [4], [26], [166], [56], [88], [96], [101], [129], [157], [169], [174], [179], [181], [6], [11], [12], [85], [95], [108], [128], [132], [138], [153], [156], [164], [180], [184], [2], [7], [13], [44], [46], [50], [81], [102], [109], [125], [135], [140], [150], [154], [158], [38], [61], [78], [83], [114], [122], [151], and [155].

B.2 Obfuscation Methods Articles

Articles that mention the listed obfuscation methods, ordered by publication year, are as follows: OM_01_L, References [4, 9, 13, 20, 23, 30, 42, 53, 59, 66, 69, 72, 75, 76, 80, 86, 87, 94, 102, 111, 112, 114, 116, 120, 122, 125, 131, 133–135, 138, 146, 152, 154, 160, 163, 166, 167, 169, 170, 174, 177, 178]. OM_02_L, References [4, 8, 9, 13, 20, 27, 30, 42, 53, 56, 59, 66, 69, 72, 75, 76, 80, 83, 86, 87, 90, 94, 99,

102, 108, 110–112, 114, 116, 120, 122, 123, 125, 129, 131, 133–135, 138, 139, 146, 147, 152, 160, 166, 167, 169, 170, 174, 176, 177, 178, 181]. OM_03_, References [56, 83, 122, 166]. OM_04_L, References [9, 20, 27, 39, 66, 87, 114, 122, 123, 125, 134, 138, 139, 160, 166, 167, 170]. OM_05_L, References [4, 6, 8, 9, 13, 19, 20, 23, 25–27, 30, 39, 42, 53, 56, 59, 66, 69, 71–73, 75, 76, 80, 83, 85, 87, 89, 90, 94, 99, 102, 103, 108, 110–112, 114, 116, 120, 122–125, 129, 131, 133–135, 138, 146, 147, 152, 154, 160, 163, 166, 167, 169, 170, 172, 174–178, 181]. OM_06_L, References [110, 114, 122, 133, 134, 166, 181]. OM_07_S – [8, 9, 13, 19, 20, 23, 25–27, 30, 39, 42, 53, 54, 56, 66, 67, 69, 71–73, 75, 76, 80, 83, 85–87, 89, 90, 94, 99, 102, 103, 108, 110–112, 114, 116, 120, 122, 124, 125, 129, 131, 133, 135, 138, 139, 146, 147, 166, 167, 169, 170, 172, 175, 177, 178, 181]. OM_08_S, References [4, 9, 13, 19, 20, 23, 25–27, 42, 54, 56, 66, 67, 69, 72, 76, 83, 90, 94, 102, 103, 114, 116, 122, 125, 131, 133, 135, 138, 139, 147, 154, 163, 166, 167, 169, 170, 172, 174, 176–178, 181]. OM_09_S, References [25, 26, 39, 42, 67, 76, 83, 94, 103, 108, 114, 122, 131, 133–135, 147, 166, 175, 178, 181]. OM_10_S—OM_10a_S, References [26, 42, 67, 76, 83, 89, 94, 108, 114, 122, 131, 133–135, 147, 166]; OM_10b_S, References [9, 13, 20, 66, 75, 94, 116, 120, 122, 125, 135, 138, 163, 166, 167, 169, 170, 172]. OM_11_AS—OM_11a_AS, References [4, 8, 9, 13, 20, 23, 66, 69, 72, 75, 94, 108, 112, 114, 116, 120, 122, 125, 133, 134, 138, 147, 163, 166, 170, 172, 174, 178, 181]; OM_11b_AS, References [108, 114, 122, 133, 134, 166]; OM_11c_AS, References [4, 9, 13, 20, 23, 27, 66, 69, 72, 87, 94, 112, 114, 116, 122, 125, 133, 138, 166, 170, 172, 175, 178, 181]. OM_12_AS, References [4, 8, 9, 13, 19, 20, 23, 27, 42, 66, 69, 72, 75, 76, 80, 86, 94, 99, 108, 110–112, 114, 116, 120, 122, 125, 131, 133, 135, 138, 147, 152, 166, 167, 169, 170, 172, 176, 178, 181]; OM_13_LG, References [26, 54, 129, 154]; OM_14_LG, References [9, 66]; OM_15_LG, References [42, 76, 94, 122, 131, 166]; and OM_16_LG, References [9, 20, 66, 125, 138, 167, 170, 172].

B.3 Tools Articles

For the top-five tools, the following articles develop, use, or compare that tool: (1) *JPlag*, References [2, 3, 6, 9, 12, 20, 25–28, 30, 33, 34, 38, 41, 44, 46, 55, 64, 66, 83, 89, 90, 98, 100, 101, 107, 108, 114, 115, 117, 119, 120, 122, 134, 140, 154, 158, 164, 166, 169, 176, 182]; (2) *MOSS*, References [8, 12, 13, 16, 20, 25–28, 34, 36, 38, 55, 63, 64, 70, 73, 79, 81–83, 89, 101, 107, 108, 111, 113, 114, 117, 122, 125, 129, 134, 154, 155, 164, 181, 182]; (3) *Sherlock-Warwick*, References [15, 27, 28, 34, 75, 83, 114, 117, 122]; (4) *Plaggie*: [3, 55, 83, 108, 114, 122, 154]; and (5) *SIM from Grune*, References [27, 28, 55, 108, 167, 179].

B.4 Algorithm and Algorithm-type Articles

Articles identified to use a particular algorithm or algorithm type are as follows: *Attribute Counting*, References [4, 5, 8, 39, 41, 42, 52, 54, 59, 67, 72, 76, 80, 89, 90, 94, 115, 118, 139, 143, 150, 153, 157, 177, 178]; *Fingerprint*, References [56, 63, 73, 89, 90, 95, 102, 118, 123, 124, 176]; *String Matching*, References [4, 12, 21, 26, 38, 39, 53, 56, 59, 67, 75, 87, 91, 100, 107, 110, 134, 140, 146, 147, 150, 152, 165, 172, 173, 177]; *Text Based*, References [33, 34, 44, 71, 75, 89, 90, 151, 173]; *Structure Based*, References [39, 41, 56, 63, 68, 89, 90, 98, 103, 117, 119, 120, 124, 128, 135, 140, 143, 146, 151, 154, 158, 171, 172, 175, 176]; *Stylistic*, References [8, 127, 128, 140, 151]; *Semantic*, References [24, 33, 34, 56, 63, 157, 175]; *nGram*, References [9, 20, 38, 44, 56, 63, 64, 66, 89, 90, 125, 133, 151, 154, 176]; *Trees*, References [7, 10, 50, 67, 68, 73, 84–86, 89, 90, 95, 98, 100, 109, 119, 120, 139, 147, 158, 174–176, 181], *Graphs* - [24, 26, 56, 68, 73, 80, 103, 135, 158, 174]; *Clustering*, References [2, 8, 66, 70, 86, 115, 125, 133, 176, 181, 182]; *History Based*, References [88, 155]; *XML based*, References [10, 70, 112]; *Compiled Code Based*, References [9, 69, 110, 129, 135]; *Compression Based*, References [25, 40, 71, 94, 165, 182]; *Only GST*, References [4, 13]; *Only RKR*, References [107, 152]; *RKR with GST*, References [78, 91, 100, 134, 154, 166, 173]; *Winnowing*, References [56, 90, 166]; *Latent Semantic*, References [33, 34]; and *Watermarking*, References [36, 101].

B.5 Tool Comparison Articles

Articles developing a tool and doing tool comparisons, References [2, 3, 6, 8–10, 12, 13, 20, 25–28, 30, 33, 36, 38, 41, 44, 63, 64, 66, 70, 72, 73, 75, 82, 83, 85, 89, 90, 98, 100, 101, 107, 111, 115, 117, 119, 120, 125, 129, 134, 140, 143, 151, 154, 158, 166, 171–173, 175, 176, 181, 182];

Articles only comparing tools, References [46, 55, 94, 108, 114, 122, 167, 170].

B.6 Comparison Measures Articles

Articles that do comparisons are using the following comparison metrics: (a) *F-beta*, References [2, 33, 38, 44, 46, 101, 125, 129, 140, 151, 158, 166]; (b) *F1*, References [33, 38, 44, 46, 101, 125, 140, 151, 158, 166]; (c) *Precision*, References [2, 9, 20, 30, 33, 38, 44, 46, 64, 101, 107, 115, 125, 129, 134, 140, 151, 158, 166, 167, 170, 176, 182]; (d) *Recall*, References [2, 9, 20, 30, 33, 38, 44, 46, 64, 98, 101, 107, 115, 125, 129, 134, 140, 151, 158, 166, 167, 170, 176, 182]; (e) *Sensitivity*, References [55, 89, 115, 167, 170]; (f) *Selectivity*, References [167, 170]; (g) *Excess Detections*, References [167, 170]; (h) *Performance Index*, References [167, 170]; (i) *Speed*, references [26, 82, 119, 125, 172]; and (j) *Qualitative*, References [3, 6, 8–10, 13, 25, 26, 27, 28, 36, 41, 44, 46, 55, 63, 66, 70, 72, 73, 75, 83, 85, 89, 90, 94, 100, 107, 108, 111, 114, 115, 117, 120, 122, 125, 134, 143, 151, 166, 167, 171–173, 175, 176, 181].

B.7 Dataset-type Articles

Articles that compare tools by dataset type are as follows: (1) *Personal Student Dataset*, References [2, 6, 8, 9, 12, 13, 20, 25–28, 30, 33, 36, 44, 55, 64, 66, 70, 73, 75, 82, 83, 85, 101, 107, 114, 115, 117, 125, 129, 134, 143, 154, 158, 167, 170–172, 176, 182]; (2) *Personal Generated Dataset*, References [12, 13, 25, 27, 41, 55, 63, 72, 73, 75, 85, 89, 90, 94, 98, 100, 108, 111, 114, 115, 120, 134, 166, 173, 175, 181, 182]; (3) *SOCO*, References [38, 46, 140, 151]; (4) *ICPC*, References [98]; and (5) *Other dataset*, References [9, 44, 119].

B.8 Programming Language Articles

Articles describing tools that were tested on the programming language are as follows: (1) *All*, References [2, 9, 19, 38, 44, 73, 83, 99, 101, 147]; (2) *C*, References [8, 9, 16, 19–21, 30, 38, 44, 46, 52, 53, 59, 63, 70, 78, 85–87, 91, 95, 98, 99, 102, 108, 111, 112, 114, 118, 119, 128, 129, 132, 134, 140, 146, 147, 153, 154, 157, 165, 172–174, 176–178, 181, 184]; (3) *Java*, References [2–5, 7, 9, 10, 15, 16, 19, 25, 27, 28, 33, 34, 36, 38, 41, 44, 46, 50, 55, 64, 66, 69, 83, 89, 99, 107–109, 117, 119, 120, 127, 128, 133–135, 140, 147, 150, 151, 154, 155, 158, 166, 181]; (4) *C++*, References [6, 8, 13, 16, 25, 27, 28, 44, 46, 68, 73, 75, 76, 82, 84, 85, 87, 88, 90, 98, 100, 111, 113, 114, 115, 118, 121, 129, 132, 134, 150, 154, 182]; (5) *Pascal*, References [16, 42, 54, 67, 75, 79, 91, 139, 167, 170–172]; (6) *Python*, References [19, 44, 99, 154, 155]; (7) *PHP*, References [19, 83, 99, 154]; (8) *C#*, References [83, 110, 118]; (9) *Haskell*, References [19, 56, 99]; (10) *UnixShell*, References [19, 75, 99]; (11) *Fortran*, References [39, 143]; (12) *JavaScript*, References [83, 157]; (13) *COBOL*, References [39, 72]; (14) *VisualBasic*, References [94, 125]; (15) *Matlab*, References [118, 179]; (16) *Lisp*, References [91, 172]; (17) *Assembly*, References [114, 146]; (18) *CSS*, Reference [83]; (19) *HTML*, Reference [83]; (20) *ANSI_C*, Reference [26]; (21) *Scheme-*, Reference [134]; (22) *R*, Reference [12]; (23) *BASIC*, Reference [39]; (24) *SML*, Reference [103]; (25) *Prolog*, Reference [103]; (26) *DLV*, Reference [124]; (27) *Miranda*, Reference [75]; (28) *VHDL*; Reference [71]; and (29) *APL2*, Reference [22].

REFERENCES

[1] C. L. Aasheim, P. S. Rutner, L. Li, and S. R. Williams. 2012. Plagiarism and programming: A survey of student attitudes. *J. Inf. Syst. Educ.* 23, 3 (2012), 297–314.

[2] G. Acampora and G. Cosma. 2015. A Fuzzy-based approach to programming language independent source-code plagiarism detection. In *Proceedings of the IEEE International Conference on Fuzzy Systems*. IEEE, 1–8. DOI: https://doi.org/10.1109/FUZZ-IEEE.2015.7337935

[3] A. Ahtiainen, S. Surakka, and M. Rahikainen. 2006. Plaggie: GNU-licensed source code plagiarism detection engine for Java exercises. In *Proceedings of the 6th Baltic Sea Conference on Computing Education Research Koli Calling*, Vol. 276. ACM Press, New York, NY, 141. DOI: https://doi.org/10.1145/1315803.1315831

[4] O. Ajmal, M. M. Saad Missen, T. Hashmat, M. Moosa, and T. Ali. 2013. EPlag: A two layer source code plagiarism detection system. In *Proceedings of the 8th International Conference on Digital Information Management*. IEEE, 256–261. DOI: https://doi.org/10.1109/ICDIM.2013.6693984

[5] Z. A. Al-Khanjari, J. A. Fiaidhi, R. A. Al-Hinai, and N. S. Kutti. 2010. PlagDetect: A Java programming plagiarsim detection tool. *ACM Inroads* 1, 4 (2010), 66. DOI: https://doi.org/10.1145/1869746.1869766

[6] I. Alsmadi, I. AlHami, and S. Kazakzeh. 2014. Issues related to the detection of source code plagiarism in students assignments. *Int. J. Softw. Eng. Appl.* 8, 4 (2014), 23–34. DOI: https://doi.org/10.14257/ijseia.2014.8.4.03

[7] V. Anjali, T. R. Swapna, and B. Jayaraman. 2015. Plagiarism detection for Java programs without source codes. *Proc. Comput. Sci.* 46 (2015), 749–758. DOI: https://doi.org/10.1016/j.procs.2015.02.143

[8] S. Arabyarmohamady, H. Moradi, and M. Asadpour. 2012. A coding style-based plagiarism detection. In *Proceedings of 2012 International Conference on Interactive Mobile and Computer Aided Learning*. IEEE, 180–186. DOI: https://doi.org/10.1109/IMCL.2012.6396471

[9] C. Arwin and S. M. M. Tahaghoghi. 2006. Plagiarism detection across programming languages. In *Proceedings of the 29th Australasian Computer Science Conference*, Vol. 48. 277–286.

[10] A. Asadullah, M. Basavaraju, I. Stern, and V. D. Bhat. 2012. Design patterns based pre-processing of source code for plagiarism detection. In *Proceedings of the 19th Asia-Pacific Software Engineering Conference*, Vol. 2. IEEE, 128–135. DOI: https://doi.org/10.1109/APSEC.2012.141

[11] J. Baby, Kannan T, Vinod P, and V. Gopal. 2014. Distance indices for the detection of similarity in C programs. In *Proceedings of the International Conference on Computation of Power, Energy, Information and Communication*. IEEE, 462–467. DOI: https://doi.org/10.1109/ICCPEIC.2014.6915408

[12] M. Bartoszuk and M. Gagolewski. 2014. A Fuzzy R code similarity detection algorithm. In *Proceedings of the 15th International Conference on Information Processing and Management of Uncertainty in Knowledge-based Systems*. Vol. 444. CCIS, 21–30. DOI: https://doi.org/10.1007/978-3-319-08852-5_3

[13] A. M. Bejarano, L. E. García, and E. E. Zurek. 2015. Detection of source code similitude in academic environments. *Comput. Appl. Eng. Educ.* 23, 1 (2015), 13–22. DOI: https://doi.org/10.1002/cae.21571

[14] F. W. Blackwell. 1980. Computer assistance in detecting plagiarsim in student programs. California Education Computer Consortium, 28–30.

[15] I. Bosnić, B. Mihaljević, M. Orlić, and M. Žagar. 2012. Source code validation and plagiarism detection: Technology-rich course experiences. In *Proceedings of the 4th International Conference on Computer Supported Education*, Vol. 2. 149–154.

[16] K. W. Bowyer and L. O. Hall. 1999. Experience using "MOSS" to detect cheating on programming assignments. In *Proceedings of the 29th Annual Frontiers in Education Conference: Designing the Future of Science and Engineering Education*, Vol. 3. Stripes Publishing L.L.C, San Juan, Puerto Rico, USA, 13B3/18–13B3/22. DOI: https://doi.org/10.1109/FIE.1999.840376

[17] H. P. Breivold, I. Crnkovic, and M. Larsson. 2012. A systematic review of software architecture evolution research. *Inf. Softw. Technol.* 54, 1 (2012), 16–40. DOI: https://doi.org/10.1016/j.infsof.2011.06.002

[18] P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner, and M. Khalil. 2007. Lessons from applying the systematic literature review process within the software engineering domain. *J. Syst. Softw.* 80, 4 (2007), 571–583. DOI: https://doi.org/10.1016/j.jss.2006.07.009

[19] R. Brixtel, M. Fontaine, B. Lesner, C. Bazin, and R. Robbes. 2010. Language-independent clone detection applied to plagiarism detection. In *Proceedings of the 10th IEEE Working Conference on Source Code Analysis and Manipulation*. IEEE, 77–86. DOI: https://doi.org/10.1109/SCAM.2010.19

[20] S. Burrows, S. M. M. Tahaghoghi, and J. Zobel. 2007. Efficient plagiarism detection for large code repositories. *Softw. Pract. Exper.* 37, 2 (2007), 151–175. DOI: https://doi.org/10.1002/spe.750

[21] R. A. C. Campos and F. J. Z. Martinez. 2012. Batch source-code plagiarism detection using an algorithm for the bounded longest common subsequence problem. In *Proceedings of the 9th International Conference on Electrical Engineering, Computing Science and Automatic Control*. IEEE, 1–4. DOI: https://doi.org/10.1109/ICEEE.2012.6421180

[22] M. Cebrian, M. Alfonseca, and A. Ortega. 2009. Towards the validation of plagiarism detection tools by means of grammar evolution. *IEEE Trans. Evol. Comput.* 13, 3 (2009), 477–485. DOI: https://doi.org/10.1109/TEVC.2008.2008797

[23] G. Chen, Y. Zhang, and X. Wang. 2011. Analysis on identification technologies of program code similarity. In *Proceedings of the International Conference of Information Technology, Computer Engineering and Management Sciences*, Vol. 1. IEEE, 188–191. DOI: https://doi.org/10.1109/ICM.2011.240

[24] R. Chen, L. Hong, C. Lu, and W. Deng. 2010. Author identification of software source code with program dependence graphs. In *Proceedings of the 34th Annual Computer Software and Applications Conference Workshops*. IEEE, 281–286. DOI: https://doi.org/10.1109/COMPSACW.2010.56

[25] X. Chen, B. Francia, M. Li, B. McKinnon, and A. Seker. 2004. Shared information and program plagiarism detection. *IEEE Trans. Inf. Theory* 50, 7 (2004), 1545–1551. DOI: https://doi.org/10.1109/TIT.2004.830793

[26] M. Chilowicz, É. Duris, and G. Roussel. 2013. Viewing functions as token sequences to highlight similarities in source code. *Sci. Comput. Program.* 78, 10 (2013), 1871–1891. DOI: https://doi.org/10.1016/j.scico.2012.11.008

[27] D. Chudá and B. Kováčová. 2010. Checking plagiarism in e-learning. In *Proceedings of the 11th International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing on International Conference on Computer Systems and Technologies*, Vol. 471. ACM Press, New York, NY, 419. DOI: https://doi.org/10.1145/1839379.1839453

[28] D. Chuda and P. Navrat. 2010. Support for checking plagiarism in e-learning. *Proc. Soc. Behav. Sci.* 2, 2 (2010), 3140–3144. DOI: https://doi.org/10.1016/j.sbspro.2010.03.478

[29] D. Chuda, P. Navrat, B. Kovacova, and P. Humay. 2012. The issue of (software) plagiarism: A student view. *IEEE Trans. Educ.* 55, 1 (2012), 22–28. DOI: https://doi.org/10.1109/TE.2011.2112768

[30] V. Ciesielski, N. Wu, and S. Tahaghoghi. 2008. Evolving similarity functions for code plagiarism detection. In *Proceedings of the 10th Annual Genetic and Evolutionary Computation Conference*. Melbourne, Australia, 1453–1460.

[31] Georgina Cosma and Mike Joy. 2006. *Source-Code Plagiarism: A UK Academic Perspective—Research Report No. 422*. Technical Report. Department of Computer Science, The University of Warwick, Coventry, UK.

[32] G. Cosma and M. Joy. 2008. Towards a definition of source-code plagiarism. *IEEE Trans. Educ.* 51, 2 (2008), 195–200. DOI: https://doi.org/10.1109/TE.2007.906776

[33] G. Cosma and M. Joy. 2012. An approach to source-code plagiarism detection and investigation using latent semantic analysis. *IEEE Trans. Comput.* 61, 3 (2012), 379–394. DOI: https://doi.org/10.1109/TC.2011.223

[34] G. Cosma and M. Joy. 2012. Evaluating the performance of LSA for source-code plagiarism detection. *Informatica (Slovenia)* 36, 4 (2012), 409–424.

[35] Fintan Culwin, Anna Macleod, and Thomas Lancaster. 2001. Source code plagiarism in UK HE computing schools. In *Proceedings of the 2nd Annual LTSN-ICS Conference*. LTSN Centre for Information and Computer Sciences, London, United Kingdom.

[36] C. Daly. 2005. A technique for detecting plagiarism in computer code. *Comput. J.* 48, 6 (2005), 662–666. DOI: https://doi.org/10.1093/comjnl/bxh139

[37] Merriam-Webster Online Dictionary. 2017. Plagiarsim. Retrieved from https://www.merriam-webster.com/dictionary/plagiarize.

[38] C. Domin, H. Pohl, and M. Krause. 2016. Improving plagiarism detection in coding assignments by dynamic removal of common ground. In *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems (CHI EA'16)*. ACM Press, New York, NY, 1173–1179. DOI: https://doi.org/10.1145/2851581.2892512

[39] J. L. Donaldson, A. M. Lancaster, and P. H. Sposato. 1981. A plagiarism detection system. *ACM SIGCSE Bull.* 13, 1 (1981), 21–25. DOI: https://doi.org/10.1145/953049.800955

[40] T. E. Doyle, Q. Sheng, and A. Ieta. 2009. Entropy based verification of academic integrity. *Comput. Educ. J.* 19, 1 (2009), 70–76.

[41] M. El Bachir Menai and N. S. Al-Hassoun. 2010. Similarity detection in Java programming assignments. In *Proceedings of the 5th International Conference on Computer Science & Education*. IEEE, 356–361. DOI: https://doi.org/10.1109/ICCSE.2010.5593613

[42] J. A. W. Faidhi and S. K. Robinson. 1987. An empirical approach for detecting program similarity and plagiarism within a university programming environment. *Comput. Educ.* 11, 1 (1987), 11–19. DOI: https://doi.org/10.1016/0360-1315(87)90042-X

[43] J. Fiaidhi, S. Mohammed, and Z. AlKhanjari. 2003. Designing a vortal for detecting Java programs cyberplagiarism. In *Proceedings of the International Conference on Internet Computing*, Vol. 1, 252–256.

[44] E. Flores, A. Barrón-Cedeño, L. Moreno, and P. Rosso. 2015. Uncovering source code reuse in large-scale academic environments. *Comput. Appl. Eng. Educ.* 23, 3 (2015), 383–390. DOI: https://doi.org/10.1002/cae.21608

[45] E. Flores, L. Moreno, and P. Rosso. 2016. Detecting source code re-use with ensemble models. In *Proceedings of the 4th Spanish Conference on Information Retrieval*. ACM Press, New York, NY, 1–7. DOI: https://doi.org/10.1145/2934732.2934738

[46] E. Flores, P. Rosso, L. Moreno, and E. Villatoro-Tello. 2015. On the detection of SOurce COde re-use. In *Proceedings of the Forum for Information Retrieval Evaluation*. ACM Press, New York, NY, 21–30. DOI: https://doi.org/10.1145/2824864.2824878

[47] E. Flores, P. Rosso, L. Moreno, and E. Villatoro-Tello. 2015. Pan@fire2015: Overview of cl-soco track on the detection of cross-language source code re-use. In *Proceedings of the 7th Forum for Inform. Retrieval Eval.* 4–6.

[48] Robert Fraser. 2014. Collaboration, collusion and plagiarism in computer science coursework. *Inf. Educ.* 13, 2 (Sep. 2014), 179–195. DOI: https://doi.org/10.15388/infedu.2014.01

[49] M. Freire. 2008. Visualizing program similarity in the Ac plagiarism detection system. In *Proceedings of the Working Conference on Advanced Visual Interfaces*. ACM Press, New York, NY, 404–407. DOI: https://doi.org/10.1145/1385569.1385644

[50] D. Ganguly and G. J. F. Jones. 2015. DCU@FIRE-2014: An information retrieval approach for source code plagiarism detection. In *Proceedings of the Forum for Information Retrieval Evaluation (FIRE'14)*, Vol. 05-07. ACM Press, New York, NY, 39–42. DOI: https://doi.org/10.1145/2824864.2824887

[51] M. Ghosh and R. Ghosh. 2005. Automatic assessment marking and plagiarism detection using SOM, fuzzy logic, and decision trees. In *Proceedings of the 9th World Multi-conference on Systemics, Cybernetics and Informatics*, Vol. 3. 281–285.

[52] M. Ghosh, B. Verma, and A. Nguyen. 2002. An automatic assessment marking and plagiarism detection. In *Proceedings of the 1st International Conference on Information Technology and Applications*. 489–494.

[53] D. Gitchell and N. Tran. 1999. Sim: A utility for detecting similarity in computer programs. In *Proceedings of the 30th SIGCSE Technical Symposium on Computer Science Education*, Vol. 31. ACM Press, New York, NY, 266–270. DOI: https://doi.org/10.1145/299649.299783

[54] S. Grier. 1981. A tool that detects plagiarism in Pascal programs. In *Proceeedings of the 12th SIGCSE Technology Symposium on Computer Science Education (SIGCSE'81)*. ACM Press, New York, NY, 15–20. DOI: https://doi.org/10.1145/800037.800954

[55] J. Hage, P. Rademaker, and N. van Vugt. 2011. Plagiarism detection for Java: A tool comparison. In *Proceedings of the Computer Science Education Research Conference (CSERC'11)*. 33–46.

[56] J. Hage, B. Vermeer, and G. Verburg. 2013. Research paper: Plagiarism detection for Haskell with Holmes. In *Proceedings of the 3rd Computer Science Education Research Conference on Computer Science Education Research (CSERC'13)*. 19–30.

[57] M. H. Halstead. 1972. Natural laws controlling algorithm structure? *ACM SIGPLAN Not.* 7, 2 (1972), 19–26. DOI: https://doi.org/10.1145/953363.953366

[58] M. H. Halstead. 1977. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., New York, NY.

[59] J. Hamblen, A. Parker, and S. Wachtel. 1988. A new undergraduate computer arithmetic software laboratory. *IEEE Trans. Educ.* 31, 3 (1988), 177–180. DOI: https://doi.org/10.1109/13.2309

[60] S. Heblikar, P. Sharma, M. Munnangi, and C. Bankapur. 2015. Normalization based stop-word approach to source code plagiarism detection. In *Proceedings of the CEUR Workshop*, Vol. 1587. CEUR-WS, 6–9.

[61] M. J. Heron and P. Belford. 2016. Musings on misconduct: A practitioner reflection on the ethical investigation of plagiarism within programming modules. *ACM SIGCAS Comput. Soc.* 45, 3 (2016), 438–444. DOI: https://doi.org/10.1145/2874239.2874304

[62] T. Hodgkinson, H. Curtis, D. MacAlister, and G. Farrell. 2016. Student academic dishonesty: The potential for situational prevention. *J. Crim. Just. Educ.* 27, 1 (2016), 1–18. DOI: https://doi.org/10.1080/10511253.2015.1064982

[63] L. Huang, S. Shi, and H. Huang. 2010. A new method for code similarity detection. In *Proceedings of the IEEE International Conference on Progress in Informatics and Computing*, Vol. 2. IEEE, 1015–1018. DOI: https://doi.org/10.1109/PIC.2010.5687856

[64] U. Inoue and S. Wada. 2012. Detecting plagiarisms in elementary programming courses. In *Proceedings of the 9th International Conference on Fuzzy Systems and Knowledge Discovery*. IEEE, 2308–2312. DOI: https://doi.org/10.1109/FSKD.2012.6234186

[65] J. Iverson, J. Schnepf, and I. Rahal. 2010. Using metrics to quantify similarity in source code: An empirical study using VOCS. In *Proceedings of the 19th International Conference on Software Engineering and Data Engineering*. 269–274.

[66] A. Jadalla and A. Elnagar. 2008. PDE4Java: Plagiarism detection engine for Java source code: A clustering approach. *Int. J. Bus. Intell. Data Min.* 3, 2 (2008), 121. DOI: https://doi.org/10.1504/IJBIDM.2008.020514

[67] H. T. Jankowitz. 1988. Detecting plagiarsim in student pascal programs. *Comput. J.* 31, 1 (1988), 1–8.

[68] J. H. Ji, S. H. Park, G. Woo, and H. G. Cho. 2008. Generating pylogenetic tree of homogeneous source code in a plagiarism detection system. *Int. J. Contr. Autom. Syst.* 6, 6 (2008), 809–817.

[69] J. H. Ji, G. Woo, and H. G. Cho. 2008. A plagiarism detection technique for Java program using bytecode analysis. In *Proceedings of the 3rd International Conference on Convergence and Hybrid Information Technology*, Vol. 1. IEEE, 1092–1098. DOI: https://doi.org/10.1109/ICCIT.2008.267

[70] S. Jia, L. Dongsheng, L. Zhang, and C. Liu. 2012. A research on plagiarism detecting method based on XML similarity and clustering. In *Proceedings of the International Workshop on Internet of Things*. Vol. 312 CCIS, 619–626. DOI: https://doi.org/10.1007/978-3-642-32427-7_88

[71] M. C. Johnson, C. Watson, S. Davidson, and D. Eschbach. 2004. Gene sequence inspired design plagiarism screening. In *Proceedings of the Annual Conference and Exposition: Engineering Researchs New Heights*. 6087–6105.
[72] E. L. Jones. 2001. Metrics based plagarism monitoring. *J. Comput. Sci. Coll.* 16, 4 (2001), 253–261.
[73] I. Jonyer, P. Apiratikul, and J. Thomas. 2005. Source code fingerprinting using graph grammar induction. In *Proceedings of the 18th International Recent Advances in Artifical IntelligenceResearch Society Conference*. 468–473.
[74] M. Joy, G. Cosma, J. Y. K. Yau, and J. Sinclair. 2011. Source code plagiarism - A student perspective. *IEEE Trans. Educ.* 54, 1 (2011), 125–132. DOI: https://doi.org/10.1109/TE.2010.2046664
[75] M. Joy and M. Luck. 1999. Plagiarism in programming assignments. *IEEE Trans. Educ.* 42, 2 (1999), 129–133. DOI: https://doi.org/10.1109/13.762946
[76] D. C. Kar. 2000. Detection of plagiarism in computer programming assignments. *J. Comput. Sci. Coll.* 15, 3 (2000), 266–276.
[77] B. Karthikeyan, V. Vaithiyanathan, and C. V. Lavanya. 2011. Similarity detection in source code using data mining techniques. *Eur. J. Sci. Res.* 62, 4 (2011), 500–505.
[78] P. Karuna and M. Preeti. 2016. Global plagiarism management through intelligence of Hawk eye. *Ind. J. Sci. Technol.* 9, 15 (2016). DOI: https://doi.org/10.17485/ijst/2016/v9i15/92113
[79] B. Kaučič, D. Sraka, M. Ramšak, and M. Krašna. 2010. Observations on plagiarism in programming courses. In *Proceedings of the 2nd International Conference on Computer Supported Education*, Vol. 2. 181–184.
[80] R. Kaushal and A. Singh. 2012. Automated evaluation of programming assignments. In *Proceedings of the IEEE International Conference on Engineering Education: Innovative Practices and Future Trends*. IEEE, 1–5. DOI: https://doi.org/10.1109/AICERA.2012.6306707
[81] M. Kaya and S. A. Özel. 2015. Integrating an online compiler and a plagiarism detection tool into the Moodle distance education system for easy assessment of programming assignments. *Comput. Appl. Eng. Educ.* 23, 3 (2015), 363–373. DOI: https://doi.org/10.1002/cae.21606
[82] W. Kechao, W. Tiantian, Z. Mingkui, W. Zhifei, and R. Xiangmin. 2012. Detection of plagiarism in students' programs using a data mining algorithm. In *Proceedings of the 2012 2nd International Conference on Computer Science and Network Technology*. IEEE, 1318–1321. DOI: https://doi.org/10.1109/ICCSNT.2012.6526164
[83] D. Kermek and M. Novak. 2016. Process model improvement for source code plagiarism detection in student programming assignments. *Inf. Educ.* 15, 1 (2016), 103–126. DOI: https://doi.org/10.15388/infedu.2016.06
[84] P. A. Khaustov. 2012. The NCP algorithm of fuzzy source code comparison. In *Proceedings of the 7th International Forum on Strategic Technology*. IEEE, 1–3. DOI: https://doi.org/10.1109/IFOST.2012.6357644
[85] H. Kikuchi, T. Goto, M. Wakatsuki, and T. Nishino. 2014. A source code plagiarism detecting method using alignment with abstract syntax tree elements. In *Proceedings of the 15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel Computing*. IEEE, 1–6. DOI: https://doi.org/10.1109/SNPD.2014.6888733
[86] Y. C. Kim and J. Choi. 2005. A program plagiarism evaluation system. In *Lecture Notes in Computer Science*, Vol. 3483. 10–19. DOI: https://doi.org/10.1007/11424925_2
[87] M. Konecki, T. Orehovacki, and A. Lovrencic. 2009. Detecting computer code plagiarism in higher education. In *Proceedings of the 31st International Conference on Information Technology Interfaces*. IEEE, 409–414. DOI: https://doi.org/10.1109/ITI.2009.5196118
[88] I. Koss and R. Ford. 2013. Authorship is continuous: Managing code plagiarism. *IEEE Secur. Priv.* 11, 2 (2013), 72–74. DOI: https://doi.org/10.1109/MSP.2013.26
[89] J. Y. Kuo and F. C. Huang. 2010. Code analyzer for an online course management system. *J. Syst. Softw.* 83, 12 (2010), 2478–2486. DOI: https://doi.org/10.1016/j.jss.2010.07.037
[90] J. Y. Kuo, F. C. Huang, C. Hung, and L. H. Z. Yang. 2012. The study of plagiarism detection for object-oriented programming. In *Proceedings of the 6th International Conference on Genetic and Evolutionary Computing*. IEEE, 188–191. DOI: https://doi.org/10.1109/ICGEC.2012.145
[91] C. Kustanto and I. Liem. 2009. Automatic source code plagiarism detection. In *Proceedings of the ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing*. IEEE, 481–486. DOI: https://doi.org/10.1109/SNPD.2009.62
[92] T. Lancaster. 2003. *Effective and Efficient Plagiarism Detection*. Ph.D. Dissertation. South Bank University. Retrieved from https://www.academia.edu/168972/Effective_and_Efficient_Plagiarism_Detection.
[93] Thomas Lancaster and Fintan Culwin. 2004. A comparison of source code plagiarism detection engines. *Comput. Sci. Educ.* 14, 2 (Jun. 2004), 101–112. DOI: https://doi.org/10.1080/08993400412331363843
[94] T. Lancaster and M. Tetlow. 2005. Does automated anti-plagiarism have to be complex? Evaluating more appropriate software metrics for finding collusion. In *Proceedings of the 22nd Annual Conference of the Australian Society for Computers in Tertiary Education*. 361–370.

[95] F. M. Lazar and O. Banias. 2014. Clone detection algorithm based on the abstract syntax tree approach. In *Proceedings of the 9th IEEE International Symposium on Applied Computational Intelligence and Informatics*. IEEE, 73–78. DOI: https://doi.org/10.1109/SACI.2014.6840038

[96] T. Le, A. Carbone, J. Sheard, M. Schuhmacher, M. de Raath, and C. Johnson. 2013. Educating computer programming students about plagiarism through use of a code similarity detection tool. In *Learning and Teaching in Computing and Engineering*. IEEE, 98–105. DOI: https://doi.org/10.1109/LaTiCE.2013.37

[97] Ronald J. Leach. 1995. Using metrics to evaluate student programs. *ACM SIGCSE Bull.* 27, 2 (Jun. 1995), 41–43. DOI: https://doi.org/10.1145/201998.202010

[98] Y. J. Lee, J. S. Lim, J. H. Ji, H. G. Cho, and G. Woo. 2012. Plagiarism detection among source codes using adaptive methods. *KSII Trans. Internet Inf. Syst.* 6, 6 (2012), 1627–1648. DOI: https://doi.org/10.3837/tiis.2012.06.008

[99] B. Lesner, R. Brixtel, C. Bazin, and G. Bagan. 2010. A novel framework to detect source code plagiarism. In *Proceedings of the 2010 ACM Symposium on Applied Computing*. ACM Press, New York, NY, 57. DOI: https://doi.org/10.1145/1774088.1774101

[100] X. Li and X. J. Zhong. 2010. The source code plagiarism detection using AST. In *Proceedings of the International Symposium on Intelligence Information Processing and Trusted Computing*. IEEE, 406–408. DOI: https://doi.org/10.1109/IPTC.2010.90

[101] T. T. Lin and S. H. Tung. 2013. Plagiarism detection in programming exercises using a Markov model approach. *ICIC Expr. Lett.* 7, 9 (2013), 2563–2568.

[102] X. Liu, C. Xu, and B. Ouyang. 2015. Plagiarism detection algorithm for source code in computer science education. *Int. J. Dist. Educ. Technol.* 13, 4 (2015), 29–39. DOI: https://doi.org/10.4018/IJDET.2015100102

[103] G. Lukácsy and P. Szeredi. 2009. Plagiarism detection in source programs using structural similarities. *Acta Cybernet.* 19, 1 (2009), 191–216.

[104] E. Luquini and N. Omar. 2011. Programming plagiarism as a social phenomenon. In *Proceedings of the IEEE Global Engineering Education Conference*. IEEE, 895–902. DOI: https://doi.org/10.1109/EDUCON.2011.5773251

[105] S. Mann and Z. Frew. 2006. Similarity and originality in code: Plagiarism and normal variation in student assignments. In *Proceedings of the Conferences in Research and Practice in Information Technology Series*, Vol. 52. 143–150.

[106] E. Marais, U. Minnaar, and D. Argles. 2006. Plagiarism in e-learning systems: Identifying and solving the problem for practical assignments. In *Proceedings of the 6th IEEE International Conference on Advanced Learning Technologies*. IEEE, 822–824. DOI: https://doi.org/10.1109/ICALT.2006.1652567

[107] L. Mariani and D. Micucci. 2012. AuDeNTES: Automatic detection of teNtative plagiarism according to a rEference Solution. *ACM Trans. Comput. Educ.* 12, 1 (2012), 1–26. DOI: https://doi.org/10.1145/2133797.2133799

[108] V. T. Martins, D. Fonte, P. R. Henriques, and D. Da Cruz. 2014. Plagiarism detection: A tool survey and comparison. In *Proceedings of the 3rd Symposium on Languages, Applications and Technologies*, Vol. 38. Schloss Dagstuhl-Leibniz-Zentrum fur Informatik GmbH, Dagstuhl Publishing, Gualtar, Portugal, 143–158. DOI: https://doi.org/10.4230/OASIcs.SLATE.2014.143

[109] V. T. Martins, P. R. Henriques, and D. da Cruz. 2015. An AST-based tool, spector, for plagiarism detection: The approach, functionality, and implementation. *Commun. Comput. Inf. Sci.* 563 (2015), 153–159. DOI: https://doi.org/10.1007/978-3-319-27653-3_15

[110] V. Mateljan, V. Juričić, and K. Peter. 2011. Analysis of programming code similarity by using intermediate language. In *Proceedings of the 34th International Convention on Information and Communication Technology, Electronics and Microelectronics*. 1235–1240.

[111] Z. Mei and L. Dongsheng. 2010. An XML plagiarism detection algorithm for procedural programming languages. In *Proceedings of the International Conference on Educational and Information Technology*, Vol. 3. IEEE, 427–431. DOI: https://doi.org/10.1109/ICEIT.2010.5608336

[112] Z. Mei and L. Dongsheng. 2010. An XML plagiarism detection model for C program. In *Proceedings of the 3rd International Conference on Advanced Computer Theory and Engineering*, Vol. 1. IEEE, 460–464. DOI: https://doi.org/10.1109/ICACTE.2010.5578975

[113] C. Meyer, C. Heeren, E. Shaffer, and J. Tedesco. 2011. CoMoTo: The collaboration modeling toolkit. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*. ACM Press, New York, NY, 143–147. DOI: https://doi.org/10.1145/1999747.1999789

[114] M. Mišić, Z. Šiuštran, and J. Protić. 2016. A comparison of software tools for plagiarism detection in programming assignments. *Int. J. Eng. Educ.* 32, 2 (2016), 738–748.

[115] L. Moussiades and A. Vakali. 2005. PDetect: A clustering approach for detecting plagiarism in source code datasets. *Comput. J.* 48, 6 (2005), 651–661. DOI: https://doi.org/10.1093/comjnl/bxh119

[116] M. Mozgovoy. 2006. Desktop tools for offline plagiarism detection in computer programs. *Inf. Educ.* 5, 1 (2006), 97–112.

[117] M. Mozgovoy, K. Fredriksson, D. White, M. Joy, and E. Sutinen. 2005. Fast plagiarism detection system. In *Lecture Notes in Computer Science*, Vol. 3772. Springer-Verlag, Berlin, 267–270. DOI: https://doi.org/10.1007/11575832_30

[118] S. Narayanan and S. Simi. 2012. Source code plagiarism detection and performance analysis using fingerprint based distance measure method. In *Proceedings of the 7th International Conference on Computer Science & Education*. IEEE, 1065–1068. DOI: https://doi.org/10.1109/ICCSE.2012.6295247

[119] S. C. Ng, S. O. Choy, and R. Kwan. 2008. *An Intelligent Online Assessment System for Programming Courses*. World Scientific. 217–231 pages. DOI: https://doi.org/10.1142/9789812799456_0014

[120] S. C. Ng, A. K. F. Lui, and L. S. Wong. 2012. Tree-based comparison for plagiarism detection and automatic marking of programming assignments. In *Proceedings of the International Conference on ICT in Teaching and Learning*. Vol. 302 CCIS,, 165–179. DOI: https://doi.org/10.1007/978-3-642-31398-1_15

[121] M. M. Novak. 1998. Correlations in computer programs. *Fractals* 06, 02 (1998), 131–138. DOI: https://doi.org/10.1142/S0218348X9800016X

[122] M. Novak. 2016. Review of source-code plagiarism detection in academia. In *Proceedings of the 39th International Convention on Information and Communication Technology, Electronics and Microelectronics*. IEEE, 796–801. DOI: https://doi.org/10.1109/MIPRO.2016.7522248

[123] A. Nunome, H. Hirata, M. Fukuzawa, and K. Shibayama. 2010. Development of an e-learning back-end system for code assessment in elementary programming practice. In *Proceedings of the 38th Annual Fall Conference on SIGUCCS*. ACM Press, New York, NY, 181. DOI: https://doi.org/10.1145/1878335.1878381

[124] J. Oetsch, J. Pührer, M. Schwengerer, and H. Tompits. 2010. The system Kato: Detecting cases of plagiarism for answer-set programs. *Theor. Pract. Logic Program.* 10, 4–6 (2010), 759–775. DOI: https://doi.org/10.1017/S1471068410000402

[125] T. Ohmann and I. Rahal. 2015. Efficient clustering-based source code plagiarism detection using PIY. *Knowl. Inf. Syst.* 43, 2 (2015), 445–472. DOI: https://doi.org/10.1007/s10115-014-0742-2

[126] A. Ohno and H. Murao. 2008. A quantification of students coding style utilizing HMMBased coding models for in-class source code plagiarism detection. In *Proceedings of the 2008 3rd International Conference on Innovative Computing Information and Control*. IEEE, 553–553. DOI: https://doi.org/10.1109/ICICIC.2008.614

[127] A. Ohno and H. Murao. 2011. A two-step in-class source code plagiarism detection method utilizing improved CM algorithm and SIM. *Int. J. Innov. Comput. Inf. Contr.* 7, 8 (2011), 4729–4739.

[128] A. Ohno, T. Yamasaki, and K. I. Tokiwa. 2014. An online system for scoring and plagiarism detection in university programing class. In *Proceedings of the 22nd International Conference on Computers in Education*. Asia-Pacific Society for Computers in Education, Nara, Japan, 37–39.

[129] C. Oprisa, G. Cabau, and A. Colesa. 2013. From plagiarism to malware detection. In *Proceedings of the 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Comp.* IEEE, 227–234. DOI: https://doi.org/10.1109/SYNASC.2013.37

[130] K. J. Ottenstein. 1976. An algorithmic approach to the detection and prevention of plagiarism. *ACM SIGCSE Bull.* 8, 4 (1976), 30–41. DOI: https://doi.org/10.1145/382222.382462

[131] A. Parker and J. O. Hamblen. 1989. Computer algorithms for plagiarism detection. *IEEE Trans. Educ.* 32, 2 (1989), 94–99. DOI: https://doi.org/10.1109/13.28038

[132] D. Pohuba, T. Dulik, and P. Janku. 2014. Automatic evaluation of correctness and originality of source codes. In *Proceedings of the 10th European Workshop on Microelectronics Education*. IEEE, 49–52. DOI: https://doi.org/10.1109/EWME.2014.6877393

[133] J. Y. H. Poon, K. Sugiyama, Y. F. Tan, and M. Y. Kan. 2012. Instructor-centric source code plagiarism detection and plagiarism corpus. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*. ACM Press, New York, NY, 122–127. DOI: https://doi.org/10.1145/2325296.2325328

[134] L. Prechelt, G. Malpohl, and M. Philippsen. 2002. Finding plagiarisms among a set of programs with JPlag. *J. Univ. Comput. Sci.* 8, 11 (2002), 1016–1038. DOI: https://doi.org/10.3217/jucs-008-11-1016

[135] D. Qiu, J. Sun, and H. Li. 2015. Improving similarity measure for Java programs based on optimal matching of control flow graphs. *Int. J. Softw. Eng. Knowl. Eng.* 25, 07 (2015), 1171–1197. DOI: https://doi.org/10.1142/S0218194015500229

[136] Chaiyong Ragkhitwetsagul, Jens Krinke, and David Clark. 2016. Similarity of source code in the presence of pervasive modifications. In *Proceedings of the 2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM'16)*. IEEE, 117–126. DOI: https://doi.org/10.1109/SCAM.2016.13

[137] I. Rahal and J. Degiovanni. 2008. Towards efficient source code plagiarism detection: An n-gram-based approach. In *Proceedings of the 21st International Conference on Computer Applications in Industry and Engineering*. 174–179.

[138] I. Rahal and C. Wielga. 2014. Source code plagiarism detection using biological string similarity algorithms. *J. Inf. Knowl. Manage.* 13, 03 (2014), 1450028. DOI: https://doi.org/10.1142/S0219649214500282

[139] G. K. Rambally and M. Lesage. 1990. An inductive inference approach to plagiarsim detection in computer-programs. In *Proceedings of the National Educational Computing Conference (INTSOC'90)*, 23–29.

[140] A. Ramírez-de-la Cruz, G. Ramírez-de-la Rosa, C. Sánchez-Sánchez, and H. Jiménez-Salazar. 2015. On the importance of lexicon, structure and style for identifying source code plagiarism. In *Proceedings of the Forum for Information Retrieval Evaluation (FIRE'14)*. ACM Press, New York, NY, 31–38. DOI: https://doi.org/10.1145/2824864.2824879

[141] A. Ramírez-De-La-Cruz, G. Ramírez-De-La-Rosa, C. Sánchez-Sánchez, H. Jiménez-Salazar, C. Rodríguez-Lucatero, and W. A. Luna-Ramírez. 2015. High level features for detecting source code plagiarism across programming languages. In *Proceedings of the CEUR Workshop*, Vol. 1587. CEUR-WS, 10–14.

[142] N. G. Resmi and K. P. Soman. 2014. Multiresolution analysis of source code using discrete wavelet transform. *Int. J. Appl. Eng. Res.* 9, 22 (2014), 13341–13360.

[143] S. S. Robinson and M. L. Soffa. 1980. An instructional aid for student programs. *ACM SIGCSE Bull.* 12, 1 (1980), 118–129. DOI: https://doi.org/10.1145/953032.804623

[144] J. C. Rodriguez, E. Rubio Royo, and Z. Hernandez. 2011. USES OF VPL. In *Proceedings of the 5th International Technology, Education and Development Conference*. IATED-INT, 743–748.

[145] J. C. Rodriguez-del Pino, E. Rubio-Royo, and Z. Hernandez-Figueroa. 2011. Fighting plagiarism: Metrics and methods to measure and find similarities among source code of computer programs in VPL. In *Proceedings of the 3rd International Conference of Education and New Learning Technologies (EDULEARN'11)*. IATED-INT, 4339–4346.

[146] F. Rosales, A. Garcia, S. Rodriguez, J. L. Pedraza, R. Mendez, and M. M. Nieto. 2008. Detection of plagiarism in programming assignments. *IEEE Trans. Educ.* 51, 2 (2008), 174–183. DOI: https://doi.org/10.1109/TE.2007.906778

[147] R. E. Roxas, N. R. Lim, and N. Bautista. 2006. Automatic generation of plagiarism detection among student programs. In *Proceedings of the 7th International Conference on Information Technology Based Higher Education and Training*. IEEE, 226–235. DOI: https://doi.org/10.1109/ITHET.2006.339768

[148] S. Schleimer, D. S. Wilkerson, and A. Aiken. 2003. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on on Management of Data*. ACM Press, New York, NY, 76–85. DOI: https://doi.org/10.1145/872757.872770

[149] A. Semerdzhiev and T. Trifonov. 2013. Practical aspects of plagiarsim detection in coputer science e-learning. In *Proceedings of the 6th International Conference of Education, Research and Inovation*. IATED-INT, 3953–3961.

[150] D. Shah, H. Jethani, and H. Joshi. 2015. (CLSCR) cross language source code reuse detection using intermediate language. In *Proceedings of the CEUR Workshop*, Vol. 1587. CEUR-WS, 15–18.

[151] N. Shah, S. Modha, and D. Dhruv. 2016. Differential weight based hybrid approach to detect software plagiarism. *Adv. Intell. Syst. Comput.* 409 (2016), 645–653. DOI: https://doi.org/10.1007/978-981-10-0135-2_62

[152] S. Shan, F. Guo, and J. Ren. 2012. Similarity detection method based on assembly language and string matching. *Adv. Intell. Soft Comput.* 148, 1 (2012), 363–367. DOI: https://doi.org/10.1007/978-3-642-28655-1_57

[153] S. Q. Shan, Z. G. Tian, F. J. Guo, and J. X. Ren. 2014. Similarity detection's application using Chi-square test in the property of counting method. *Appl. Mech. Mater.* 667 (2014), 32–35. DOI: https://doi.org/10.4028/www.scientific.net/AMM.667.32

[154] S. Sharma, C. S. Sharma, and V. Tyagi. 2015. Plagiarism detection tool "Parikshak." In *Proceedings of the International Conference on Communication, Information & Computing Techology*. IEEE, 1–7. DOI: https://doi.org/10.1109/ICCICT.2015.7045739

[155] D. Sheahen and D. Joyner. 2016. TAPS: A MOSS extension for detecting software plagiarsim at scale. In *Proceeedings of the 3rd ACM Conference on Learning @ Scale (L@S'16)*. ACM Press, New York, NY, 285–288. DOI: https://doi.org/10.1145/2876034.2893435

[156] Simon, B. Cook, J. Sheard, A. Carbone, and C. Johnson. 2014. Academic integrity perceptions regarding computing assessments and essays. In *Proceedings of the 10th Annual Conference on International Computing Education Research (ICER'14)*. ACM Press, New York, NY, 107–114. DOI: https://doi.org/10.1145/2632320.2632342

[157] I. Smeureanu and B. Iancu. 2013. Source code plagiarsim detection method using ontologies. In *Proceedings of the International Conference on Informatics in Economy*. 594–597.

[158] H. J. Song, S. B. Park, and S. Y. Park. 2015. Computation of program source code similarity by composition of parse tree and call graph. *Math. Probl. Eng.* 2015 (2015), 1–12. DOI: https://doi.org/10.1155/2015/429807

[159] N. Spolaôr and F. B. V. Benitti. 2017. Robotics applications grounded in learning theories on tertiary education: A systematic review. *Comput. Educ.* 112 (2017), 97–107. DOI: https://doi.org/10.1016/j.compedu.2017.05.001

[160] D. Sraka and B. Kaucic. 2009. Source code plagiarism. In *Proceedings of the 31st International Conference on Information Technology Interfaces*. IEEE, 461–466. DOI: https://doi.org/10.1109/ITI.2009.5196127

[161] The Mendeley Support Team. 2011. Getting Started with Mendeley. Retrieved from http://www.mendeley.com.

[162] M. F. Tennyson and F. J. Mitropoulos. 2014. Choosing a profile length in the SCAP method of source code authorship attribution. In *Proceedings of the IEEE SoutheastCon 2014*. Institute of Electrical and Electronics Engineers Inc., Bradley University, Peoria, IL. DOI: https://doi.org/10.1109/SECON.2014.6950705

[163] J. Traxler. 1995. Plagiarism in programming: A review and discussion of the factors. In *Proceedings of the 2nd International Conference on Software Engineering in Higher Education II (SEHE'95)*. Computational Mechanics, Inc., Billerica, MA, 131–138.

[164] N. D. Tselikas, M. Samarakou, D. Karolidis, P. Prentakis, and S. Athineos. 2014. Automatic plagiarism detection in programming laboratory courses. In *Proceedings of the 7th IADIS International Conference Information Systems*. IADIS, 232–238.

[165] K. Ueta and H. Tominaga. 2010. A development and application of similarity detection methods for plagiarism of online reports. In *Proceedings of the 9th International Conference on Information Technology Based Higher Education and Training*. IEEE, 363–371. DOI: https://doi.org/10.1109/ITHET.2010.5480091

[166] Z. Đurić and D. Gašević. 2013. A source code similarity system for plagiarism detection. *Comput. J.* 56, 1 (2013), 70–86. DOI: https://doi.org/10.1093/comjnl/bxs018

[167] K. L. Verco and M. J. Wise. 1996. Plagiarism à la mode: A comparison of automated systems for detecting suspected plagiarism. *Comput. J.* 39, 9 (1996), 749–750.

[168] D. Vogts. 2009. Plagiarising of source code by novice programmers a "cry for help"? In *Proceedings of the 2009 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*. ACM Press, 141–149. DOI: https://doi.org/10.1145/1632149.1632168

[169] C. Wang, Z. Liu, and L. Dongsheng. 2013. Preventing and detecting plagiarism in programming course. *Int. J. Secur. Appl.* 7, 5 (2013), 269–278. DOI: https://doi.org/10.14257/ijsia.2013.7.5.25

[170] G. Whale. 1990. Identification of program similarity in large populations. *Comput. J.* 33, 2 (1990), 140–146.

[171] G. Whale. 1990. Software metrics and plagiarism detection. *J. Syst. Softw.* 13, 2 (1990), 131–138. DOI: https://doi.org/10.1016/0164-1212(90)90118-6

[172] M. J. Wise. 1992. Detection of similarities in student programs. In *ACM SIGCSE Bull.* 24, 1 (1992), 268–271. DOI: https://doi.org/10.1145/135250.134564

[173] M. J. Wise. 1996. YAP3: Improved detection of similarities in computer program and other texts. In *Proceedings of the 27th SIGCSE Technical Symposium on Computer Science Education*, Vol. 28. ACM Press, New York, NY, 130–134. DOI: https://doi.org/10.1145/236452.236525

[174] A. Wojdyga. 2013. Towards intellectual property theft prevention: Economic significance of automatic software plagiarism verification. *Act. Probl. Econ.* 142, 4 (2013), 300–306.

[175] S. Wu, Y. Hao, X. Gao, B. Cui, and C. Bian. 2010. Homology detection based on abstract syntax tree combined simple semantics analysis. In *Proceedings of the 2010 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, Vol. 3. IEEE, 410–414. DOI: https://doi.org/10.1109/WI-IAT.2010.100

[176] H. Xiong, H. Yan, Z. Li, and H. Li. 2009. In *Proceedings of the International Conference on Computational Intelligence and Software Engineering*. DOI: https://doi.org/10.1109/CISE.2009.5366790

[177] S. Yang and X. Wang. 2010. A visual domain recognition method based on function mode for source code plagiarism. In *Proceedings of the 3rd International Symposium on Intelligent Information Technology and Security Informatics*. IEEE, 580–584. DOI: https://doi.org/10.1109/IITSI.2010.114

[178] S. Yang, X. Wang, C. Shao, and P. Zhang. 2010. Recognition on source codes similarity with weighted attributes eigenvector. In *Proceedings of the International Conference on Intelligent Control and Information Processing*. IEEE, 539–543. DOI: https://doi.org/10.1109/ICICIP.2010.5565209

[179] K. Zakova, J. Pistej, and P. Bistak. 2013. Online tool for student's source code plagiarism detection. In *Proceedings of the IEEE 11th International Conference on Emerging eLearning Technologies and Applications*. IEEE, 415–419. DOI: https://doi.org/10.1109/ICETA.2013.6674469

[180] D. Zhang, M. Joy, G. Cosma, R. Boyatt, J. Sinclair, and J. Yau. 2014. Source-code plagiarism in universities: A comparative study of student perspectives in China and the UK. *Assess. Eval. High. Educ.* 39, 6 (2014), 743–758. DOI: https://doi.org/10.1080/02602938.2013.870122

[181] L. Zhang and L. Dongsheng. 2013. AST-based multi-language plagiarism detection method. In *Proceedings of the IEEE 4th International Conference on Software Engineering and Service Science*. IEEE, 738–742. DOI: https://doi.org/10.1109/ICSESS.2013.6615411

[182] L. Zhang, Y. T. Zhuang, and Z. M. Yuan. 2007. A program plagiarism detection model based on information distance and clustering. In *Proceedings of the 2007 International Conference on Intelligent Pervasive Computing*. IEEE, 431–436. DOI: https://doi.org/10.1109/IPC.2007.10

[183] M. Zhong, Y. Li, and L. Dongsheng. 2012. A source code and non-source code plagiarism detection research for C program. In *Proceedings of the 3rd International Conference on Computer Technology and Development*. ASME, New York, NY, 2163–2167.

[184] H. M. Zhu, L. Zhang, W. Sun, and Y. X. Sun. 2014. A token oriented measurement method of source code similarity. *Appl. Mech. Mater.* 668 (2014), 899–902. DOI: https://doi.org/10.4028/www.scientific.net/AMM.668-669.899

Received June 2018; revised December 2018; accepted December 2018