Dolos: language-agnostic plagiarism detection in source code

Rien Maertens https://orcid.org/0000-0002-2927-3032
Charlotte Van Petegem https://orcid.org/0000-0003-0779-4897
Niko Strijbol https://orcid.org/0000-0002-3161-174X
Toon Baeyens https://orcid.org/0000-0002-2069-7824
Arne Jacobs https://orcid.org/0000-0003-0412-9648
Peter Dawyndt https://orcid.org/0000-0002-1623-9070
Bart Mesuere https://orcid.org/0000-0003-0610-3441

Correspondence

Rien Maertens (Rien.Maertens@UGent.be),
Department of Applied Mathematics, Computer Science and Statistics,
Ghent University, Krijgslaan 281 S9, 9000 Ghent, Belgium

Keywords

plagiarism, cheating, academic dishonesty, source code, programming language, data visualisation, online learning, remote assessment

Abstract

Learning to code is increasingly embedded in secondary and higher education curricula, where solving programming exercises plays an important role in the learning process and in formative and summative assessment. Unfortunately, students admit that copying code from each other is a common practice. We present a case study to demonstrate that switching to remote learning even strengthens this phenomenon and introduce Dolos as a user-friendly plagiarism detection tool that supports many programming languages. Where teachers indicate they rarely use plagiarism detection tools, Dolos aims at lowering the barrier for using such tools in practice. Dolos is powered by state-of-the art similarity detection algorithms and outperforms existing tools on a standardised benchmark. Its interactive visualisations assist teachers in discovering, proving and preventing plagiarism. Dolos is available under the permissive MIT open-source licence at dolos.ugent.be.

Lay description

What is already known about this topic?

- Source code plagiarism is common in programming courses.
- Students mention lack of checking as one reason to commit plagiarism.
- Existing plagiarism detection tools are not broadly used in current practice.
- Current tools support few programming languages.

What this paper adds:

- A user-friendly plagiarism detection tool that supports many programming languages.
- Interactive visualisations to discover, prove and prevent plagiarism.
- Plagiarism detection tools can help to prevent plagiarism for remote assessment.

Implications for practice and/or policy:

- Dolos makes it easier for teachers to detect source code plagiarism.
- Dolos helps to prevent students from committing source code plagiarism.
- Interactive visualisations support plagiarism discovery as an exploratory process.
- Keep in mind that high similarity does not equal proven plagiarism.

1. Introduction

A large majority of students admit they have cheated at some point during their higher education studies, leaving McCabe, Butterfield, & Terviño with the conclusion that "*the prevalence of self-reported cheating is high enough for all of us—students, faculty, and administrators—to be seriously concerned*" (McCabe, Butterfield, & Trevino, 2012, p. 71). This is not different in computing courses, where students usually practice their coding skills by solving coding challenges. Teachers also use programming exercises in formative assessment to improve student attainment or in summative assessment to evaluate student learning. Solutions for coding challenges typically consist of one or more text documents containing code for a specific programming language, generally referred to as source code.

Especially when the stakes are high, students may be tempted to resort to dishonest methods to fulfil their assignments (McCabe, Trevino, & Butterfield, 1999; Ruiperez-Valiente, Alexandron, Chen, & Pritchard, 2016). A common form of cheating on programming exercises is submitting source code authored by someone else and failing to adequately acknowledge that the particular source code is not their own. This practice is defined as source code plagiarism by Cosma and Joy (2008). It involves obtaining source code either with or without the permission of the original author, and submitting the copied code with no, minor, or even major modifications aimed at concealing plagiarism (Rahal & Wielga, 2014).

In a survey by Chuda et al. (2012) at a Slovakian university, 33% of the students admitted having copied and modified source code for one of their classes at least once. Teachers not checking for plagiarism or even openly tolerating it, were found among the reasons why students considered copying code from their peers. Only 8% of the university staff mentioned they used specific software to detect plagiarism. This is even lower than 14 of 53 (26%) representatives of higher education computing schools in the United Kingdom interviewed by Culwin et al. (2001) that indicated using automated tools for plagiarism detection.

We believe that this lack of adoption could be explained by the fact that current plagiarism detection tools are hard to use.. Problems either arise with installing or running the software, privacy issues preventing source code being sent to an external server, or interpretation of the results (Weber-Wulff, 2019). In addition, Novak et al. (2019) conclude in a recent review paper that most state-of-the-art tools only support one or a limited number of programming languages, with a major focus on Java, C and C++. However, the diversity of programming languages used in education is definitely much broader (Simon, Mason, Crick, Davenport, & Murphy, 2018).

To remedy these issues, we developed an open-source tool called Dolos that lowers the barrier for using source code plagiarism detection in education. It allows teachers to explore analysis results by means of highly interactive visualisations. Dolos also supports a broad range of programming languages out-of-the-box and is easily extensible to additional languages. We will discuss the underlying algorithms and prove it to be very accurate in identifying educational source code plagiarism according to the publicly available SOCO benchmark (Flores, Rosso, Moreno, & Villatoro-Tello, 2014).

Plagiarism detection tools are merely fast and intelligent assistants for finding highly similar code snippets, but detection of highly similar source code submitted by students in itself is not a formal proof of plagiarism (Joy & Luck, 1999). We therefore discuss a specific case on how we designed an introductory Python programming course with longstanding online learning support for plagiarism prevention and detection, and how remote learning during the COVID-19 pandemic has challenged the organisation of trustworthy and fair assessments. We describe how Dolos is used in building lines of defence against plagiarism and to find evidence for cataloguing specific cases as plagiarism beyond reasonable doubt. This case study also highlights another asset of the tool: raising student awareness and supporting exploratory analysis by interactive visualisations.

Research Goals and Methodology

This manuscript addresses the following research questions:

- **RQ1:** Can generic parser models be used to build source code plagiarism tools that support a broad range of programming languages and are highly competitive with state-of-the-art tools in the accurate detection of potential cases of plagiarism?
- **RQ2:** Can we design open and online learning and assessment environments that prevent source code plagiarism?

In section Dolos we present a new source code plagiarism detection tool by describing its algorithm and how it can be used. RQ1 is answered in subsection Validation by running a comparative benchmark between Dolos and state-of-the-art plagiarism detection tools using a publicly available dataset. RQ2 is addressed in section Case study, where we present our experiences and observations with organising a programming course over multiple years.

2. Related work

Plagiarism *sensu latu* can be defined as copying the work of someone else and presenting it as your own (Mariani & Micucci, 2012). In education, it prevails in different kinds of cheating behaviours committed by students (Sheard, Dick, Markham, Macdonald, & Walsh, 2002) and presents two obvious problems at the institutional level by threatening accurate evaluation of students and reducing their level of learning (Lupton, Chapman, & Weiss, 2000). Although this type of dishonesty is a fundamental issue, there is considerable divergence in the perception of its seriousness and prevalence between students and staff (Brimble & Stevenson-Clarke, 2005).

Technological advances make it increasingly easy for students to access and misuse resources. In a traditional physics course it was found that 10% of submitted answers to

problems in an online learning environment were copied (Palazzo, Lee, Warnakulasooriya, & Pritchard, 2010). This phenomenon is also encountered in non-traditional education; it is estimated that 10% of certificates earned in Massive Open Online Courses (MOOCs) were earned at least partially by creating a separate account to harvest answers (Alexandron, Ruipérez-Valiente, Chen, Muñoz-Merino, & Pritchard, 2017; Northcutt, Ho, & Chuang, 2016). However, new technologies simultaneously also provide better opportunities for staff to monitor potential cases of plagiarism (Brin, Davis, & García-Molina, 1995; Ercegovac & Richardson, 2004; Shivakumar & Garcia-molina, 1995). Where original resources are available in digital form and students submit digital copies of their work, advanced software tools can assist in the automation of plagiarism detection for large classes in higher education (Batane, 2010). Different computational problems arise if the origins of plagiarised fragments submitted by students can be searched for in a closed document space (e.g. original document was also submitted), in an open document space (e.g. original document is available on the Internet) or can not be accessed (e.g. student hired someone else to write his document). When the documents contain textual data, fast string algorithms exist to match a plagiarised fragment with the original document, either by indexing the closed document space or by preprocessing the plagiarised fragment before scanning the open document space (Gusfield, 1997; Vyverman, De Baets, Fack, & Dawyndt, 2012). If processing the original document is not an option, author attribution techniques might expose that author signatures of plagiarised text substantially differ from those of surrounding text or other documents submitted by the student (Stamatatos, 2009). There are also ways to detect cheating behaviour in retrospect by identifying aberrant response patterns (Alexandron, Valiente, & Pritchard, 2019; Karabatsos, 2003).

Plagiarism detection tools must see through the modifications a plagiarised fragment has undergone to disguise its origin. What kind of modifications are expected, heavily depends on the content type and the context. For example, typical countermeasures to mask obfuscations in written natural language are case folding, stemming (removing prefix/suffix from words), stopping (removing common words) and term parsing (removing whitespace, punctuation and control characters) (Hoad & Zobel, 2003). To obfuscate source code plagiarism, inexperienced programmers mainly use stylistic (e.g. altering comments or renaming variables) and syntactic (e.g. reordering independent lines of code) changes (Đurić & Gašević, 2013; Karnalim, Simon, & Chivers, 2020; Novak et al., 2019; Whale, 1990), whereas expert programmers may as well use semantic changes (e.g. altering data structures or changing an iterative process into a recursive process) (Faidhi & Robinson, 1987) or translate code from one programming language to another (Arwin & Tahaghoghi, 2006).

2.1. Educational source code plagiarism detection

In this manuscript, we focus on plagiarism detection in a closed monolingual collection of source files that students submit for a programming assignment. The collection contains both the source files under investigation for plagiarism and the source files they could have originated from (closed collection). All source files in the collection use the same programming language (monolingual collection). Students typically submit solutions that contain tens or hundreds of lines of code, but we do not expect that all solutions are syntactically correct according to the grammar rules of the programming language. Before reviewing some state-of-the-art software tools for detecting source code plagiarism in an

educational context, we first discuss the general workflow to perform plagiarism detection in practice and touch upon some key concepts along the way.

The process starts with collecting, preparing and managing collections of source files and their metadata as input for plagiarism detection tools. It involves file manipulations like filtering, formatting, arranging and packaging files in the expected structure, concatenating files of multi-file software projects (for plagiarism detection tools that expect a single file per submission), and separating files per programming language (for monolingual plagiarism detection tools). These preliminary data wrangling steps can be quite time-consuming if not properly supported by online learning environments or custom scripts (Sheahen & Joyner, 2016). Unfortunately, none of the leading source code plagiarism detection tools has a strong focus on interoperability with external software platforms, apart from providing a non-standard command line interface.

What plagiarism detection tools then actually provide are fast algorithms to find similar code fragments among the given collection of source files and support for reviewers to discern whether the resemblance of these fragments is accidental or points to actual plagiarism. Although a lot of research has gone into the algorithmic aspects of screening source code for similar fragments (Roy, Cordy, & Koschke, 2009), all leading tools nowadays follow the same two-step approach but their implementation details differ. The first step transforms each source file into a **list of tokens** to mask local obfuscations. A **token** is a structural element in the source code with a specific meaning in the programming language (e.g. a keyword, variable or operator). Tokenization uses software components that are typically used in the front end of a compiler: a **lexer** for lexical analysis, a **parser** for syntax analysis and a semantic analyser (Aho, Lam, Sethi, & Ullman, 2006). The token stream does not represent every detail in the source code, just its structural elements and some content-related details. Constructs like whitespace, delimiters or grouping parentheses are typically stripped away, and literal values, identifier names or comments are denoted as anonymous syntactic constructs. The second step searches for similar code fragments by performing a **pairwise alignment** on the lists of tokens of each pair of source files to accomodate for more global obfuscations like insertions, deletions, substitutions and transpositions (Wise, 1993).

Alternative techniques for source code plagiarism detection that have been explored include tree-based algorithms (Li & Zhong, 2010; Zhao, Xia, Fu, & Cui, 2015), graph-based algorithms (Chae, Ha, Kim, Kang, & Im, 2013; Liu, Chen, Han, & Yu, 2006), Latent Semantic Analysis information retrieval (Cosma & Joy, 2012), fuzzy-based matching (Acampora & Cosma, 2015), program logic analysis (Cheers, Lin, & Smith, 2019), and program behavioural analysis (Cheers, Lin, & Smith, 2021). However, while some yield promising initial results, neither of these approaches have gone beyond the proof-of-concept stage and resulted in tools that are applicable in practice to confirm the authors' own conclusions. Novak et. al. (2019) have put it this way: "*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.*"

There is no standard format for reporting similarity analysis results, but all tools compute a **similarity score** from each pairwise alignment of source files and report a (filtered and/or sorted) list of scores. Pairwise similarities are expressed either as values between 0 and 1 or as percentages, with higher scores for pairs with a higher likelihood of plagiarism. However, different tools use different similarity measures, preventing a direct comparison. Some tools

assist reviewers to inspect high-similarity pairs or cluster source files into larger groups based on similarity, but support for advanced plagiarism exploration is generally poor and some tools do not even output analysis results in a machine-readable format for more elaborate downstream processing.

2.2. State-of-the-art tools

Novak et al. (2019) found 120 tools for source code plagiarism detection in the scientific literature, but most have never been publicly available or no longer are. We only benchmarked Dolos against four tools these authors identified as leading the field today: Moss (Schleimer, Wilkerson, & Aiken, 2003), Sherlock Warwick (Joy & Luck, 1999), JPlag (Prechelt, Malpohl, & Philippsen, 2002), and Plaggie (Ahtiainen, Surakka, & Rahikainen, 2006). Relevant properties of these tools have been summarised in Table 1.

	Dolos	Moss	Sherlock Warwick	Sherlock Sydney	JPlag	Plaggie
initial release	2020	1997	1999	2011	1996	2002
university	Ghent University, Belgium	Stanford University, USA	Warwick University, UK	University of Sydney, Australia	Karlsruhe Institute of Technology, Germany	Helsinki University of Technology, Finland
under active development[b]	Yes	No	No	No	Yes	No
pairwise inspection[c]	Yes	Yes	Unknown[d]	No	Yes	Yes
advanced visualisations[e]	Yes	No	Yes	No	Yes	No
open-source	Yes	No	Yes	Yes	Yes	Yes
local execution[f]	Yes	No	Yes	Yes	Yes	Yes
programming language parser[g]	Yes	Yes	Yes	No	Yes	Yes
supported programming languages[h]	41	25	1 (Java)	0	7	1 (Java)
data export[i]	CSV	HTML	Unknown[d]	TXT	CSV, HTML	HTML

Table 1: Properties of plagiarism detection tools benchmarked in this study.

[a]: The source code of MOSS is not publicly available and thus it is uncertain in which programming language it is developed. However, the CLI program users have to run is written in Perl.

[b]: Whether last year there has been any development.

[c]: Whether visualising similar code fragments between two files is supported.

[d]: Property could not be discerned because we could not run Sherlock Warwick.

[e]: Whether advanced visualisations give deeper insights during plagiarism analysis beyond a similarity score or a pairwise comparison (e.g. through a plagiarism graph or a similarity score distribution).

[f]: Whether analysis can be run locally (offline).

[g]: Whether programming language parsers are used to process files. If not, plagiarism detection is unaware of programming language semantics.

[h]: Number of programming languages (parsers) supported, excluding plain text analysis.

[i]: Supported data export formats. Colour-coded by ease of processing (green=easy, yellow=hard).

Before we explain some of their inner workings, advantages and disadvantages, let us first disentangle one issue. In the field of source code plagiarism detection, two unrelated tools are named Sherlock but only one has an accompanying publication. Although they have been developed in different programming languages and at different universities, we have reasons to believe that some publications used one tool and referenced the other. To avoid such confusion, we included both tools in our discussion and will consistently suffix them with their originating university throughout this manuscript: Sherlock Warwick and Sherlock Sydney.

Moss

Moss[1] (Measure Of Software Similarity) was developed at Stanford University (USA). It is provided as a web service that is free for non-commercial use. Its source code is not publicly available, but the underlying winnowing algorithm was published by Schleimer et al. (2003). Moss supports the following programming languages: C, C++, Java, C#, Python, Visual Basic, JavaScript, FORTRAN, ML, Haskell, Lisp, Scheme, Pascal, Modula2, Ada, Perl, TCL, Matlab, VHDL, Verilog, Spice, MIPS assembly, a8086 assembly, a8086 assembly, and HCL2.

According to modern standards, Moss has a rather archaic user interface. It comes with a Perl script that bundles and uploads a collection of source files to the Moss server. The server tokenizes source files, extracts fingerprints from the token streams, and computes pairwise similarities from shared fingerprints. Results are reported as HTML pages that list pairs of highly similar source files and highlight shared code snippets in a side-by-side view. Moss has an active community of users that contributed submission scripts in other programming languages, a graphical user interface (GUI), integrations with other web

[1] http://theory.stanford.edu/~aiken/Moss/

services, and scripts that extract raw data from HTML reports and convert them into machine-readable formats.

Using Moss requires sending source code of students to an external server. This could raise security concerns and might not be allowed by local privacy regulations. Occasionally, the service is not responsive due to high demand or unscheduled outages. We found that Moss can be especially unreliable during exam periods (January and June).

Sherlock Warwick

Sherlock Warwick[2] was developed in Java at the University of Warwick (UK) and is released open-source under the GPLv2 licence. This command line tool supports most programming languages as plain text, but offers specific optimizations for Java (Joy & Luck, 1999).

Sherlock Warwick incrementally compares three versions of the source code: the original text, a version with stripped comments and whitespace, and a tokenized version. The first two comparisons work for all programming languages, but the last one is only supported for Java. Pairwise similarities are based on longest common matches of characters/tokens with a configurable number of insertions and deletions.

Despite our best efforts, we have not been able to compile and run Sherlock Warwick from source. Its dependencies are no longer available online or require a specific version that we could not pinpoint. There is also a compiled JAR file for Sherlock Warwick that we could execute, but no results were reported in the graphical user interface. We could not extract similarity values from generated reports due to unreadable formats and lack of documentation. Both issues prevented us from including this tool in the validation benchmark.

Sherlock Sydney

Sherlock Sydney[3] was developed in C at the University of Sydney (Australia). It is a command line tool that treats all source code as plain text and has no specific support for programming languages.

Sherlock Sydney parses text files (including source code) into streams of words, hashes *n* subsequent words and discards non-zero hashes after applying a bitmask. This winnowing algorithm differs from the one used by Moss (Schleimer et al., 2003). The remaining hashes are used as digital fingerprints, with pairwise similarity computed as the ratio of shared fingerprints between text files. Similarities are reported in a text file or on the terminal. Processing source code as plain text is simple and fast, but less effective against obfuscation methods typically observed in educational source code plagiarism.

The source code of Sherlock Sydney originally appeared on the university's website in 2011, and was taken offline in 2018. Other developers recovered a snapshot from the Internet Archive[4] for further maintenance on GitHub[5].

---

[2] https://warwick.ac.uk/fac/sci/dcs/research/ias/software/sherlock/
[3] https://github.com/diogocabral/sherlock
[4] https://web.archive.org/web/20170710071658/http://rp-www.cs.usyd.edu.au/~scilect/sherlock/
[5] https://github.com/diogocabral/sherlock

JPlag

JPlag[6] is written in Java and it is still actively developed at the Karlsruhe Institute of Technology (Germany). It is released open-source under the GPLv3 licence (Prechelt et al., 2002). This command line tool mainly focuses on Java but also supports C, C++, C#, Python 3, and Scheme. JPlag was initially provided as a web service and a Moodle plugin, but is now also available for download and executes locally.

JPlag tokenizes source files for supported programming languages, and otherwise processes them as plain text. Pairwise string matching is done by greedy string tiling (Wise, 1993) and similarities are computed as the fraction of characters/tokens covered by tiles. Results are reported in CSV and HTML format.

Plaggie

Plaggie[7] was developed in Java at the Helsinki University of Technology (Finland). It is released open-source under the GPLv3 licence (Ahtiainen et al., 2006). It was developed at a time when JPlag was only available as a web service with no publicly available source code. It mimicked JPlag's functionality and output, but could be downloaded and executed locally.

Plaggie only supports Java and uses the same greedy string tiling technique as JPlag. Results are reported in HTML format.

3. Dolos

Dolos is a plagiarism detection tool written in TypeScript and developed at Ghent University (Belgium). It is released under the permissive MIT open-source licence.

Dolos uses the tree-sitter parsing library (Brunsfeld et al., 2021) in combination with state-of-the-art string-matching algorithms to detect similar code fragments for a wide range of programming languages. It provides an intuitive user interface, raw data exports, an open API for integration into online learning systems, and interactive data visualisations for exploratory analysis. It is published as an npm package to ease local installation and updates. Source code and documentation are available at dolos.ugent.be.

Although Dolos may process plain text files, it was especially designed to exploit the structured grammar of programming languages. Supporting new programming languages does not require any changes to Dolos itself. Instead, it relies on the availability of tree-sitter parsers on the local system. All tree-sitter parsers produce a generic abstract syntax tree representation that Dolos uses to perform its similarity detection. At the time of writing, official tree-sitter parsers are available for Bash, C, C#, C++, CSS, Elm, Eno, ERB / EJS, Fennel, Go, HTML, Java, JavaScript, Lua, Markdown, OCaml, PHP, Python, Ruby, Rust, R, SystemRDL, TOML, TypeScript, Verilog, VHDL, Vue, YAML, and WASM. New parsers are under development for Agda, Haskell, Julia, Nix, Scala, SPARQL, and Swift. Tree-sitter and its official parsers are also released under the MIT licence.

---

[6] https://jplag.ipd.kit.edu/
[7] https://www.cs.hut.fi/Software/Plaggie/

Where a collection of source files is the only information needed for similarity analysis, Dolos may use additional metadata to enhance its visualisations. For example, submission timestamps are used to render plagiarism graphs as directed graphs, and labels can be used to colour their nodes.

3.1 Algorithm

The similarity detection algorithm of Dolos has four main steps: tokenization, fingerprinting, indexing and reporting. Where we omit implementation details in our discussion, we refer to the source code and its documentation for more information.

Tokenization. Given a programming language and a monolingual collection of source files for that language, Dolos uses the tree-sitter parsing library (Brunsfeld et al., 2021) to convert each source file into an abstract syntax tree (AST): a tree representation of the abstract syntactic structure of the source code where each node corresponds to a construct in the code (Figure 1). Tree-sitter parsers are robust enough to generate ASTs even for source code containing syntax errors, which is common for source code submitted by students.

```
0 function sum(a, b) {
1   return a + b;
2 }
3

program ([0, 0] - [3, 0])
  function ([0, 0] - [2, 1])
    identifier ([0, 9] - [0, 12])
    formal_parameters ([0, 12] - [0, 18])
      identifier ([0, 13] - [0, 14])
      identifier ([0, 16] - [0, 17])
    statement_block ([0, 19] - [2, 1])
      return_statement ([1, 2] - [1, 15])
        binary_expression ([1, 9] - [1, 14])
          identifier ([1, 9] - [1, 10])
          identifier ([1, 13] - [1, 14])
```

Figure 1: Sample JavaScript code (top) and its AST (bottom) as generated by the tree-sitter parser for JavaScript. Each AST token is on a separate line that describes the meaning of the construct, followed by its location in the source code. Locations are given as start and stop positions between brackets, with positions given as row and column indices between square brackets. Indentation reflects AST tree structure.

Each AST is then serialised into a list of tokens, resulting in a condensed representation of the original source code. While preserving the structure of the code, it is deprived from many degrees of freedom available to programmers, like formatting, variable names, delimiters,

comments, literal values, and so on. This makes similarity detection more robust against many types of obfuscations that are commonly found in educational source code plagiarism.

Fingerprinting. Given the repetitive nature of source code, it is not very informative to find individual tokens that are shared between source files. We will therefore search for common sequences of *k* successive tokens, called *k*-grams. Ideally, *k*-grams are long enough to capture local context, but short enough to bypass more structural obfuscations like adding, splitting, merging or reordering lines of code.

In order to speed up the process of finding common *k*-grams, individual tokens and *k*-grams are represented as integers. We first use a fast hashing function to convert the string representation of each token into an integer. This hashing function was specifically designed so that each integer is the unique representation of a token from a serialised AST. We then use a windowed rolling hash function to reduce the integers for each k-gram into a single integer that serves as a fingerprint. Note that the same fingerprint may appear multiple times in the representation of a single source file if similar code constructs are reused.

Because *k*-grams are overlapping, the number of fingerprints roughly corresponds to the number of tokens in the serialised AST for a source file. The Moss winnowing algorithm (Schleimer et al., 2003) is used to reduce the number of fingerprints. This works by selecting the smallest fingerprint from each overlapping window of *w* fingerprints, and only keeping the selected fingerprint if it differs from the one selected from the previous window. As a result, the number of fingerprints roughly decreases by a factor *w*, with no reduction if *w* is set to 1. Less fingerprints yields faster processing with a smaller memory footprint, but lowers the recall of the analysis.

Indexing. Previous steps converted each individual source file into a (filtered) list of fingerprints. To speed up the process of finding common fingerprints, we build an index that maps each fingerprint onto all its occurrences in the source files. For each fingerprint occurrence, the index also stores the location of the corresponding *k*-gram in the source file. As such, the index contains all information needed to compare files and visualise shared fragments.

Reporting. Dolos relies on the fact that source files still have shared fingerprints, even when students tried to obfuscate plagiarism. However, manual inspection of all source files sharing at least one fingerprint would be extremely time consuming and yield many false positives. As an alternative, Dolos computes three plagiarism metrics that represent different views on which shared fingerprints are relevant and how to quantify them. Sorting file pairs according to such a metric defines an order in which files can be inspected and helps to determine a cutoff beyond which further inspection is deemed useless.

The **similarity** between two source files *a* and *b* is computed as

$$sim(a, b) = \frac{S_a + S_b}{T_a + T_b}$$

with T_x the total number of fingerprints in file *x* and S_x the number of fingerprints in file *x* that also occur in the other file. It is a value between 0 and 1 that measures all shared fingerprints relative to the total number of fingerprints in both files. Note that S_a and S_b are not necessarily the same because a shared fingerprint might have a different number of occurrences in files *a* and *b*.

Similarity is the primary metric to compare source files, but as a relative value it is highly dependent on file size. It easily inflates for extremely small files with shared fingerprints. Therefore it might be interesting to only look at the numerator of the similarity ($S_a + S_b$) as an absolute measure for all shared fingerprints. We call this metric the **total overlap**.

The previous plagiarism metrics both treat fingerprints as independent features of source files. However, when *k*-grams are shorter than code snippets that were copied and modified between two source files, they may result in longer sequences of shared fingerprints. We formally define a **shared fragment** (or fragment for short) as a maximal exact match (Vyverman et al., 2012) between the lists of fingerprints of two files, and its length as the number of matched fingerprints. Long fragments might be another signature for plagiarism, that could be disguised by a global similarity measure when the fragments are hidden in large chunks of otherwise dissimilar code. Dolos therefore also reports the **longest fragment** as the length of the longest fragment shared between two files.

3.2. Usage

Dolos can be used as a standalone tool that processes a given collection of source files on the local system. Detailed results are stored as a set of CSV files and can be explored in a browser to find and inspect potential cases of plagiarism. As an alternative, Dolos can also be used as a JavaScript/TypeScript library to embed plagiarism detection functionality into online learning environments and other web applications.

Command line interface

Launching Dolos from the command line requires setting an option with the programming language used by the source files under investigation. The tree-sitter parser for that language needs to be installed on the local system. File paths of the source files must be passed either as separate arguments or in a single CSV file.

Dolos has well-chosen default settings for plagiarism analysis (*k*-gram length 23 and window size *w*=17) and reporting**,** which were determined by experiments we will describe later in this paper. These settings can be overridden by command line options. By default, it reports a summary table of highly similar source files to standard output, sorted by similarity. It also creates a directory with a set of CSV files containing more detailed results from plagiarism analysis. Users can instruct Dolos to start a local HTTP server that launches a graphical user interface for exploring these results in a new browser window.

Web interface

Interpreting detailed results from plagiarism analysis might be tedious and challenging. Dolos therefore provides a web interface to interactively explore the results in a browser window. All pairwise source file comparisons are summarised in a list and a plagiarism graph, accessible from the navigation drawer on the left. Manual inspection can be done by navigating between the overview provided in the list and graph views, and more detailed information visualised in the compare view.

We published the web interface for results obtained with Dolos default settings on the Java[8] and C[9] files from the SOCO benchmark (see validation section), for users that would like to explore the results in their browser. Nodes in the plagiarism graphs are colour coded according to expert plagiarism validation from the SOCO benchmark: orange nodes are involved in at least one case of plagiarism and blue nodes are not involved in any confirmed cases of plagiarism (Figure 2). Screenshots that illustrate the different views of the web interface are taken from the same analyses.

List view

The list view shows a paginated table summarising all pairwise source file comparisons (Figure 3). Each row represents a single pair of source files and displays their file paths and plagiarism metrics. Clicking a row brings up the compare view for the corresponding pair of source files. Table rows are initially sorted by decreasing similarity. Rows can be sorted in ascending or descending order for each of the columns by clicking the corresponding column header. Rows can be filtered by specifying search terms.

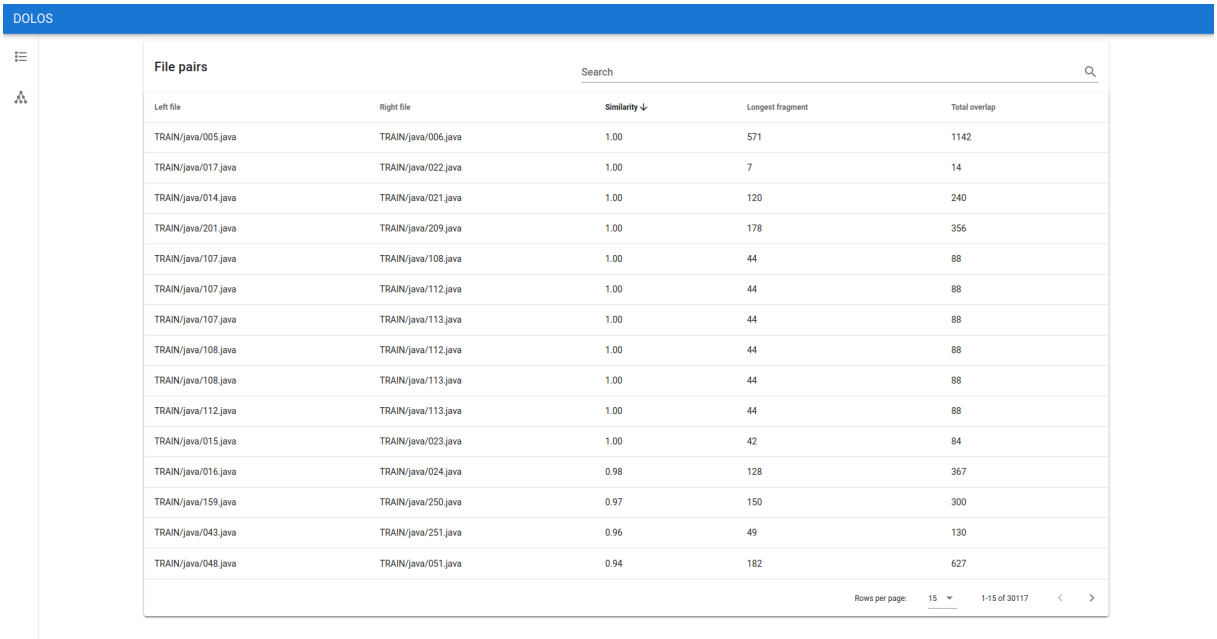

Figure 3: List view showing all pairwise comparisons of Java files in the SOCO benchmark, sorted by decreasing similarity. Apart from pairs that are 100% identical after tokenization and fingerprinting, the first page also shows pairs where only part of the source code may have been copied or where more elaborate obfuscations may have been applied.

Compare view

The compare view supports teachers in finding evidence whether or not a given pair of source files results from plagiarism. It shows the two files under review side-by-side in a scrollable code listing, with syntax highlighting based on the given programming language (Figure 4). Their plagiarism metrics are displayed at the top. As provenance for the metrics, all shared fragments are highlighted with a yellow background colour in the code listings. A schematic overview of the entire file is shown to the right of each code listing, highlighting

[8] https://dolos.ugent.be/demo/soco/java/
[9] https://dolos.ugent.be/demo/soco/c/

lines of code included in at least one shared fragment. These overviews are especially useful when the source code is much longer than the screen size.

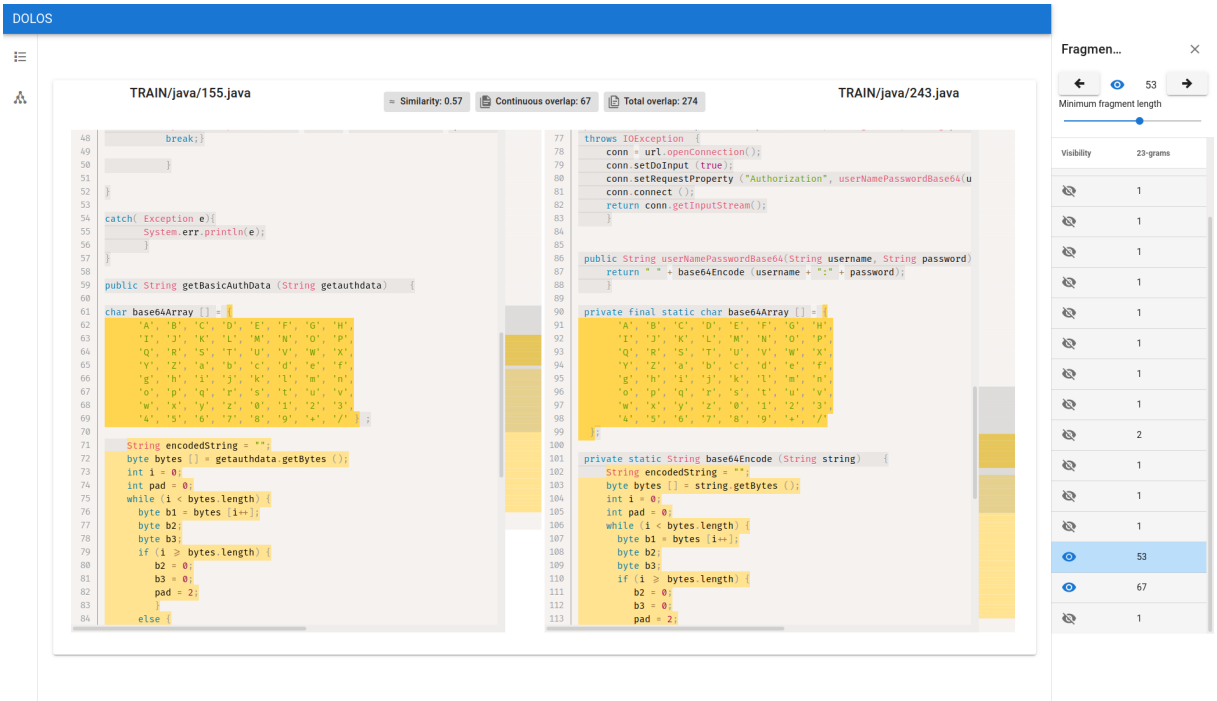

Figure 4: Compare view showing two Java files in the SOCO benchmark side-by-side. Shared fragments panel displayed to the right. High plagiarism metrics and almost identical source code in shared fragments are evidence for potential plagiarism. Expert annotations of the SOCO metadata confirm plagiarism for the two files.

A shared fragment can be selected by clicking one of its highlighted regions in the code listings or in the overview panels next to the listings. Both regions of a selected fragment are highlighted with a darker yellow background colour and are aligned next to each other in the code listings. This eases manual inspection of the source code in both regions, whose differences might have been masked by tokenization and fingerprinting. To the right of the compare view, a collapsible sidebar shows a tabular overview of all shared fragments. If two files have a lot of (overlapping) shared fragments, the list can be filtered by setting the minimum fragment length. In addition, individual fragments can be hidden manually.

Graph view

The graph view visualises all pairwise source file comparisons as an interactive plagiarism graph (Figure 2), rendered as a force-directed graph using the D3 visualisation library (Bostock, Ogievetsky, & Heer, 2011). Nodes represent the given collection of source files and can be assigned colours from metadata provided on the files. Pairs of nodes whose similarity exceeds a given threshold are connected by an edge, with edge thickness corresponding to similarity. If submission timestamps are provided as metadata on the source files, the graph becomes directed with edges pointing from the oldest to the latest submission and the oldest node in each cluster of three or more connected nodes is highlighted with a solid border as the potential source of plagiarism. Clicking an edge brings up the compare view for the corresponding pair of source files. The similarity threshold can be adjusted interactively using a slider. A checkbox controls whether singletons should be displayed, i.e. nodes that are not connected by an edge to any other node in the graph.

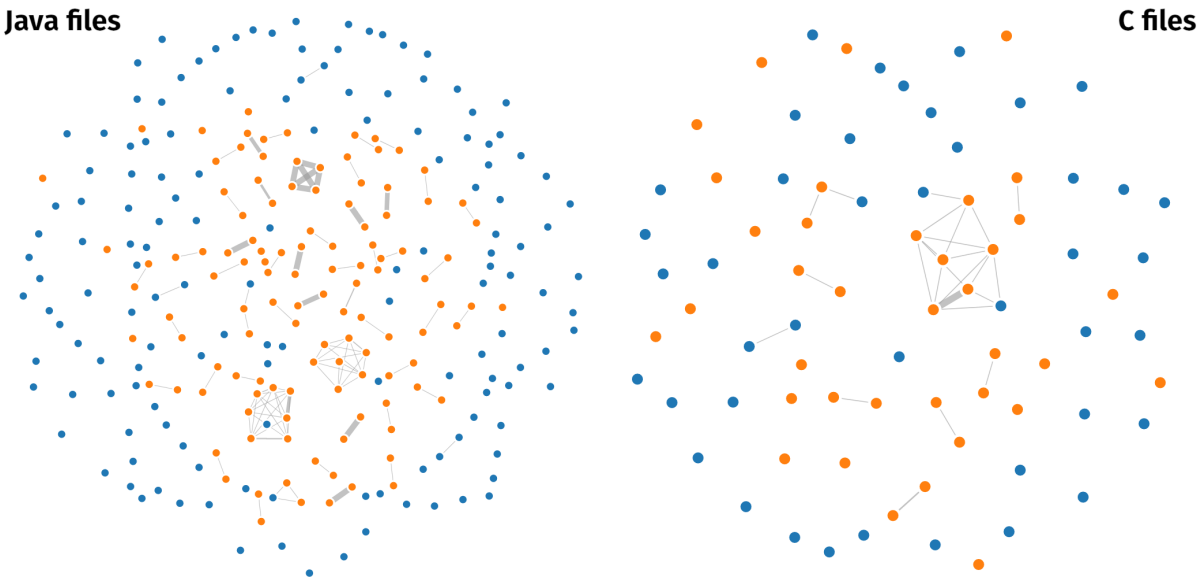

Figure 2: Plagiarism graphs for all Java (left) and C (right) files in the SOCO benchmark, with optimal similarity threshold (0.54 for Java and 0.58 for C) and singletons included in the graph. Orange nodes are involved in at least one case of plagiarism according to the metadata of the SOCO benchmark, whereas blue nodes are not involved in any confirmed cases of plagiarism.

3.3. Validation

We ran a benchmark to evaluate how well Dolos performs in comparison to similar tools. The benchmark quantitatively measures the predictive power of Dolos for plagiarism detection and compares it to four state-of-the-art tools: Moss, Sherlock Sydney, JPlag and Plaggie.

Experimental design

In the field of educational source code plagiarism detection, most tools are either validated using personal datasets of real students or generated datasets where an original program is put through a series of modifications that try to simulate plagiarism (Novak et al., 2019). These datasets are rarely available in the public domain because they are considered to contain personal information, making it hard to replicate and interpret the results or to evaluate different tools against each other.

Instead, we will use the publicly available SOCO benchmark containing 79 C files and 259 Java files (Arwin & Tahaghoghi, 2006; Flores et al., 2014). Its metadata includes expert annotations of file pairs that are deemed to result from plagiarism. 26 of the 3081 C file pairs (0.84%) are labelled as plagiarism, with 37 C files (46.84%) occurring in at least one plagiarism pair. 84 of the 33 411 Java file pairs (0.25%) are labelled as plagiarism, with 115 Java files (41.97%) occurring in at least one plagiarism pair.

All tools under investigation compute a similarity value for each pair of source files, but use different similarity measures. Direct comparison of similarity values is therefore not relevant. Instead we evaluate how well the similarity measure of a tool can separate plagiarised from non-plagiarized code using the optimal similarity threshold for that tool, as a surrogate for its predictive power to detect plagiarism. Similarity values above the threshold are considered

as positive predictions for plagiarism, and similarity values below the threshold as negative predictions. The expert annotations included in the metadata of the SOCO benchmark allow us to determine whether these predictions are true or false (Figure 5).

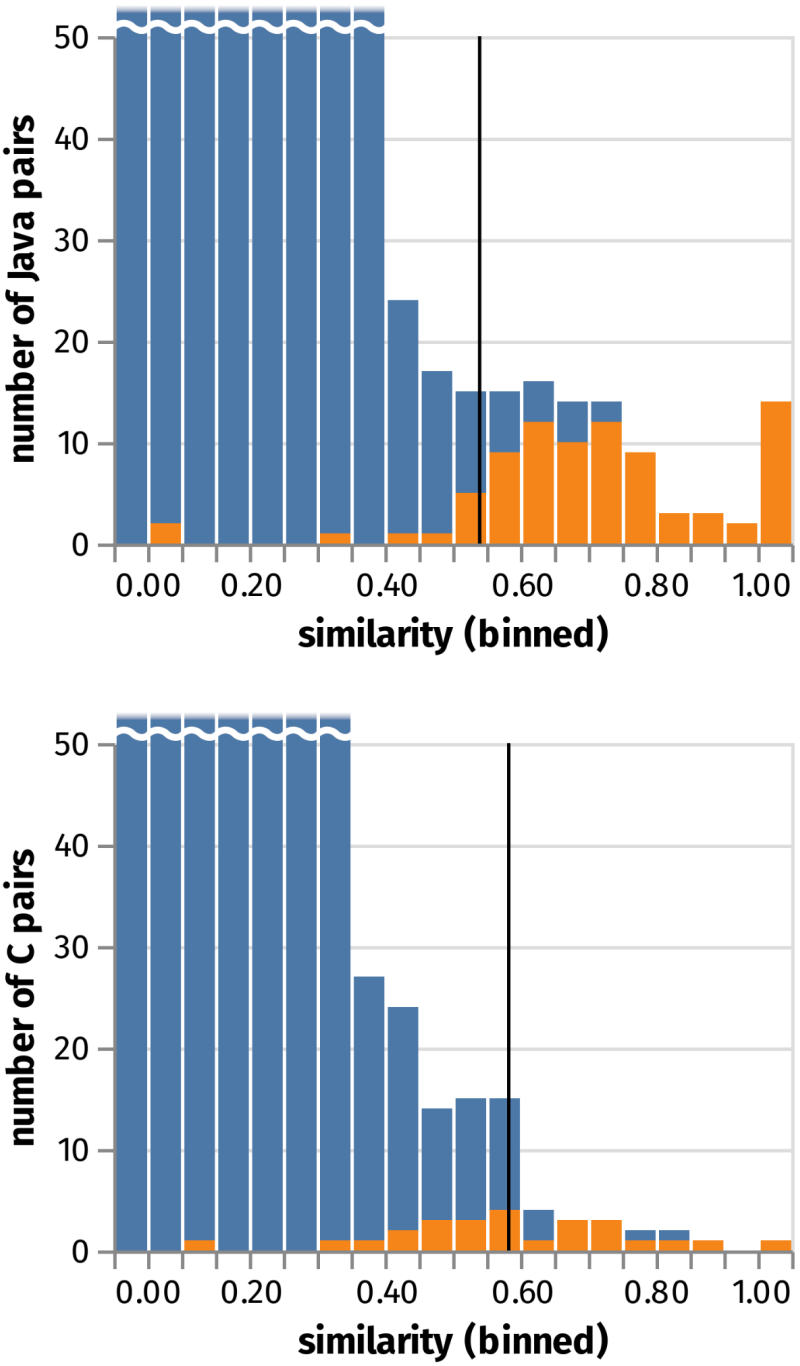

Figure 5: Distribution of pairwise similarities computed with default settings in Dolos for all Java (top) and C (bottom) files in the SOCO benchmark. True cases of plagiarism according to the SOCO metadata are shown in orange and false cases in blue. Black vertical line indicates the similarity threshold (0.54 for Java and 0.58 for C) that corresponds to the maximum F_1-score (0.865 for Java and 0.6 for C). Positive predictions are to the right of this line and negative predictions to its left. Because of the combinatorial explosion of pairs, the number of false cases with similarity values below 0.40 greatly exceeds the maximum value shown on the graph.

We use the F_1-score as a global measure for predictive accuracy of a plagiarism detection tool. It is computed as the harmonic mean of precision and recall. Because the F_1-score

depends on a similarity threshold, we determine the threshold that maximises the F_1-score as the best possible prediction for a given similarity analysis. The threshold that corresponds to the maximum F_1-score simultaneously minimises the number of false positives and true negatives. This mimics how most teachers use plagiarism detection tools: sort file pairs according to descending similarity and screen pairs in that order until the observed sequence of non-plagiarized cases is long enough to expect no further cases of plagiarism to be found.

Each tool under investigation also has parameters that influence how its similarity values are computed. Rather than simply computing the maximum F_1-score for a single configuration of parameter settings, e.g. the default settings of a tool, we repeat this process for a whole range of configurations. Such a parameter sweep gives us additional insights about the impact of parameter settings on the accuracy of predictions, if we observe differences between programming languages, and if default settings are well-chosen. It also prevents us from selecting poor configurations or cherry-picking good results for individual tools. These configurations were evaluated for each tool:

- Dolos: *k*-gram lengths 10, 12, 15, 17, 20, 23 and 25 (option `-k`, default: 23), and window sizes 17, 20, 25, 30, 35 and 40 (option `-w`, default: 17); 42 configurations in total
- Moss: all integers in the range [1, 20] as the maximum number of files in which a code fragment may appear before it is ignored (option `-m`, default: 10); 20 configurations in total
- Sherlock Sydney: all integers in the range [1, 5] as the number of zero bits (option `-z`, default: 3) and the chain length (option `-n`, default: 4); 25 configurations in total
- JPlag: all integers in the range [5, 20] as the minimum number of matching tokens (option `-t`, default: 9 for Java and 12 for C); smaller values increase sensitivity; 16 configurations in total
- Plaggie: all integers in the range [2, 22] as the minimum number of matching tokens (option `-m`, default: 11); equivalent to the option `-t` of JPlag; 21 configurations in total

Most tools only report on pairs whose similarity exceeds a given threshold. To reliably determine the threshold that yields a maximum F_1-score, we have run each tool to report on the maximum number of pairs that was possible.

Results and discussion

When plotting benchmark results for the Java and C datasets, it is clear that all tools perform better at identifying plagiarism for Java than for C (Figure 6). Both the default configuration (vertical black line) and the best configuration (rightmost circle) of each tool has far better predictive power on the Java dataset than on the C dataset. This might be explained by the difference in inter-annotator agreement between the C (κ=0.480) and Java (κ=0.668) files (Flores et al., 2014). Variation in quality of expert annotations therefore seems a more important factor than intrinsic differences between programming languages for the SOCO benchmark.

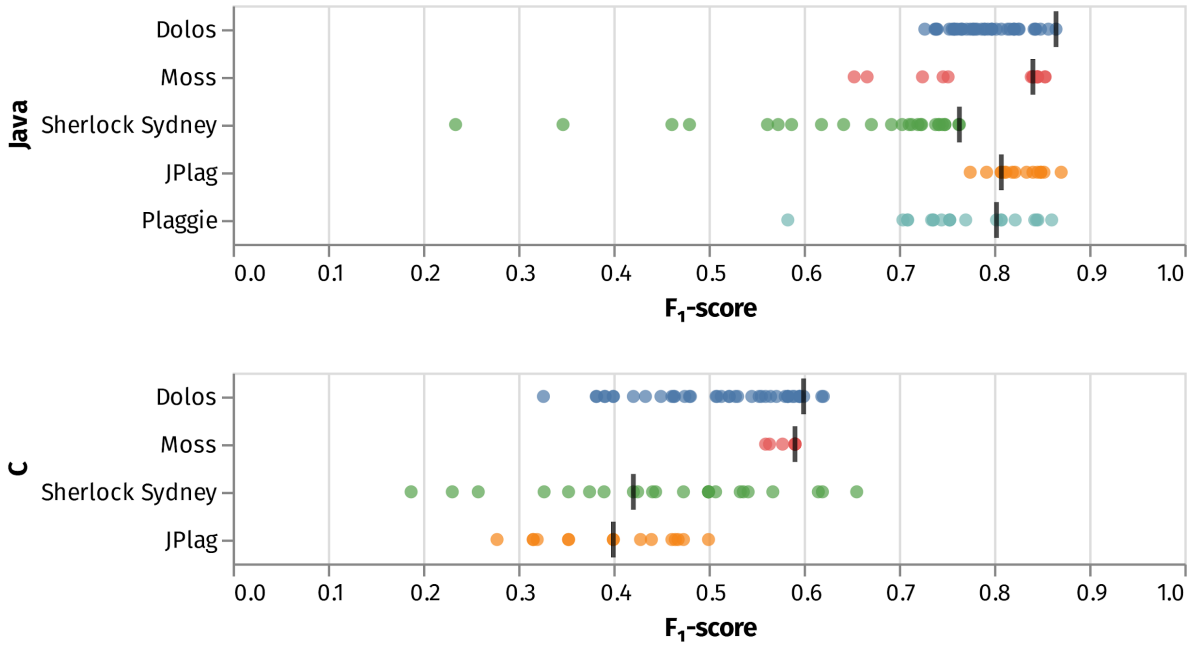

Figure 6: Predictive accuracy of plagiarism detection tools on all Java (top) and C (bottom) files in the SOCO benchmark. Circles indicate maximum F_1-scores obtained with a particular configuration of a tool (parameter sweep). Vertical black lines indicate maximum F_1-scores obtained with the default configuration of a tool. Higher F_1-scores correspond to better predictive accuracy.

Looking at the performance on the Java dataset, we see that Dolos, JPlag, Moss, and Plaggie all score well with a maximum F_1-score between 0.8 and 0.9 for their default configuration. The default configuration of Sherlock Sydney is also its best configuration, and lags slightly behind with a maximum F_1-score of 0.764. JPlag has the best scoring configuration (0.871) when using a minimum number of 19 matching tokens, closely followed by Dolos (0.865) with 23-grams and window size 17 (its default configuration). When we look at the default configurations, however, JPlag drops to third place. This is because its default configuration is one of the worse performing configurations. The performance of Moss is very consistent and seems to be less dependent on the chosen configuration. Sherlock Sydney on the other hand has a very wide range of scores between 0.234 and 0.764. Although JPlag and Plaggie use a similar algorithm, there is a significant difference in the performance of their default and best configurations.

Looking at the same graph for the C dataset, we see that the maximum F_1-scores are noticeably lower. The default configurations for Dolos and Moss have a similar score, just under 0.6. The default configurations for JPlag and Sherlock Sydney score a lot lower with 0.4 and 0.421 respectively. Once again, the scores for the different Moss configurations are very close to each other, while those of Sherlock Sydney span a very broad range. Overall, Sherlock Sydney has the best performing configuration, which is somewhat surprising since it treats all source code as plain text. The poor performance of JPlag can be explained by its main focus on the Java programming language. Plaggie was excluded from this part of the benchmark, because it does not support the C programming language.

Except for Moss, all tools show a large variation in maximum F_1-scores and it is larger for C than for Java. This underscores the importance of well-chosen parameter settings for similarity analysis. In real life usage, it is not easy to validate which parameter settings score best and we can not expect that users try various configurations. It is therefore important that

the default configuration consistently yields good results. The consistent results for Moss can be explained by the nature of its configurable parameter that only changes the maximum number of files in which a code fragment may appear before it is ignored (option `-m`). Changing this option will only have a strong impact on similarity computations if many files are plagiarised from the same source. This is the case for the Java dataset, but not for the C dataset (Figure 2).

Moss and Dolos are the only tools that consistently yield good results for both the Java and C datasets. Dolos has a slightly better overall predictive accuracy across the entire benchmark. Its default configuration scores best for both programming languages and its optimal configurations come second twice. We can thus conclude that Dolos is highly competitive with current state-of-the-art tools across multiple programming languages.

Note that none of the tools achieved an F_1-score above 0.9 for the SOCO benchmark, meaning that each of their predictions yields mismatches with the expert annotations. This underscores that source code plagiarism detection tools (and expert annotations) have their limitations. The ultimate decision to classify cases as plagiarism should therefore never be made automatically but requires human review (Weber-Wulff, 2019).

4. Case study

In this section, we discuss how we designed plagiarism prevention and detection strategies for a programming course with a strong focus on online learning. In particular, we look into the role plagiarism detection tools play in implementing these strategies and illustrate how their features can be used in practice. We used Moss throughout the early editions of the course, but in the last two years we switched to Dolos while the tool was also introduced in other programming courses at Ghent University (Belgium).

We also report how we deal with cases of plagiarism detected at different stages of the course. We applied the same plagiarism policy throughout many editions of this course, but observed an increase of plagiarism in tests and exams taken remotely during the COVID-19 pandemic (2020-2021 edition). Those interested in more details will find a more elaborate report in supplementary material, with coverage on how we found evidence for and followed up specific cases of plagiarism detected during remote assessments.

4.1. Course structure

The introductory programming course at Ghent University runs once per academic year across a 13-week semester (September-December). It is taken by a mix of undergraduate, graduate and postgraduate students enrolled in various study programmes (mainly sciences, but not computer science), with 440 students enrolled for the 2020-2021 edition. Throughout the course, students submit solutions for programming exercises to the online learning environment Dodona[10]. They immediately receive automated feedback upon each submission, even during tests and exams. Based on that feedback, students can track potential errors in their code, remedy them and submit an updated solution.

[10] https://dodona.ugent.be

Each week, we cover one topic of the Python programming language and publish six programming exercises that students must solve before a deadline one week later (Figure 7). These mandatory exercises are automatically graded by unit tests evaluated in Dodona. Students can work on mandatory exercises during weekly computer labs where they can collaborate in small groups and ask help from teaching assistants. They can also submit solutions outside lab sessions. There are two graded tests, one mid-term and one at the end of the semester, where students get two hours to solve two programming exercises. During a final exam after the semester, students get three hours and thirty minutes to solve three programming exercises. Tests and exams are taken on-campus under human surveillance. Students are allowed to complete tests and exams on their personal computers and may use the Internet "read only": they can consult documentation, forum posts, exercise solutions, and so on, but they are not allowed to communicate with someone else.

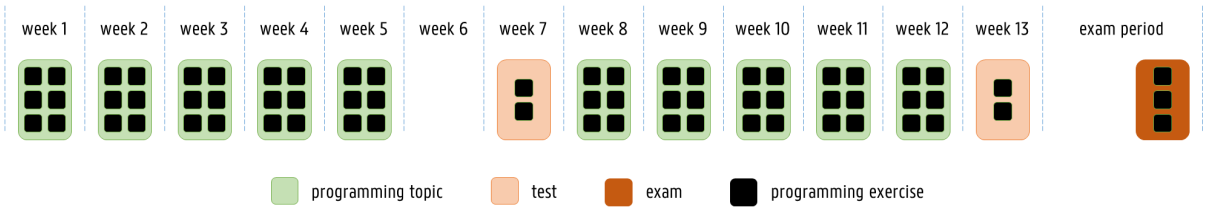

Figure 7: Outline of the Python Programming course that runs once per academic year across a 13-week semester. Students submit solutions to Dodona for ten series with 6 mandatory exercises, two tests with 2 exercises and an exam with 3 exercises. Collaboration among small groups of students is expected for the mandatory exercises, but no collaboration is allowed during tests and exams.

4.2. Plagiarism prevention

As a first line of defence against cheating on Dodona, students can check their solutions for correctness as much as they want without negative repercussions, even after the submission deadline has passed. This requires the number of possible solutions for exercises to be large enough so they cannot be solved with guesswork. Students also have to authenticate on the learning platform using their university account. While not completely flawless, this reduces the chance of students using other accounts to try solutions, a technique known as copying-using-multiple-accounts (Alexandron et al., 2019; Northcutt et al., 2016; Ruiperez-Valiente et al., 2016) that is often applied for cheating in MOOCs.

We have several strategies to discourage students from copying and modifying source code from someone else, and act differently when we suspect plagiarism among solutions for mandatory, test or exam exercises. Ghent University considers committing plagiarism as a form of fraud. As soon as invigilators or evaluators have reason to suspect a student has committed plagiarism, teachers must start a formal procedure where an examination board decides whether disciplinary measures should be imposed. Disciplinary measures may range from an adjustment of the student's score to exclusion for a period of no more than 10 years. We strictly follow these rules for tests and exams, and touch upon this topic during the first lecture. To organise "open book/open Internet" tests and exams that are valid and reliable, we always create new exercises and avoid assignments where solutions or parts thereof are readily available online.

The situation is less clear-cut for mandatory exercises, where we strongly believe that proper collaboration among small groups of students might be beneficial for learning (Prince, 2004). We even encourage students to collaborate. But we recommend them to work together in groups of no more than three students, and to exchange and discuss ideas and strategies for solving the exercises rather than sharing literal code with each other. Each edition of the course uses a new selection of mandatory exercises that we compile from test and exam exercises of the previous edition, newly created exercises, and exercises that were last used four or more editions ago. By not re-using exercises from recent course editions, we reduce the possibility of solutions being exchanged between students from one year to another. If programming courses would not have a large enough collection of exercises, solutions of previous years could be included in plagiarism analysis to detect this form of cheating.

We use Moss/Dolos to monitor submitted solutions for mandatory exercises, both before and at the deadline. The number of possible solutions for the first few mandatory exercises is too small for linking high similarity to plagiarism: submitted solutions only contain a few lines of code and the diversity of implementation strategies is small. But at some point, as the number of possible solutions increases, we start to see highly similar solutions that are reliable signals of code exchange among larger groups of students. Strikingly this usually happens among students enrolled in the same study programme (Figure 8). As soon as this happens — typically in week 3 or 4 of the course — plagiarism is discussed with the students during the next lecture. We use pseudonymized plagiarism graphs as evidence and stress that the learning effect dramatically drops when working in groups of four or more students. Typically, in such a group only one or two students make the effort to learn to code, while the others piggyback by copying solutions. We address the students that share their solutions in general by pointing out that they are probably good at programming and might want to exchange their solutions with other students in a way to help their peers. But instead of really helping them out, they actually take away learning opportunities from their fellow students by giving away the solution. After this lecture, we usually notice a steep drop in the amount of plagiarised solutions for mandatory exercises.

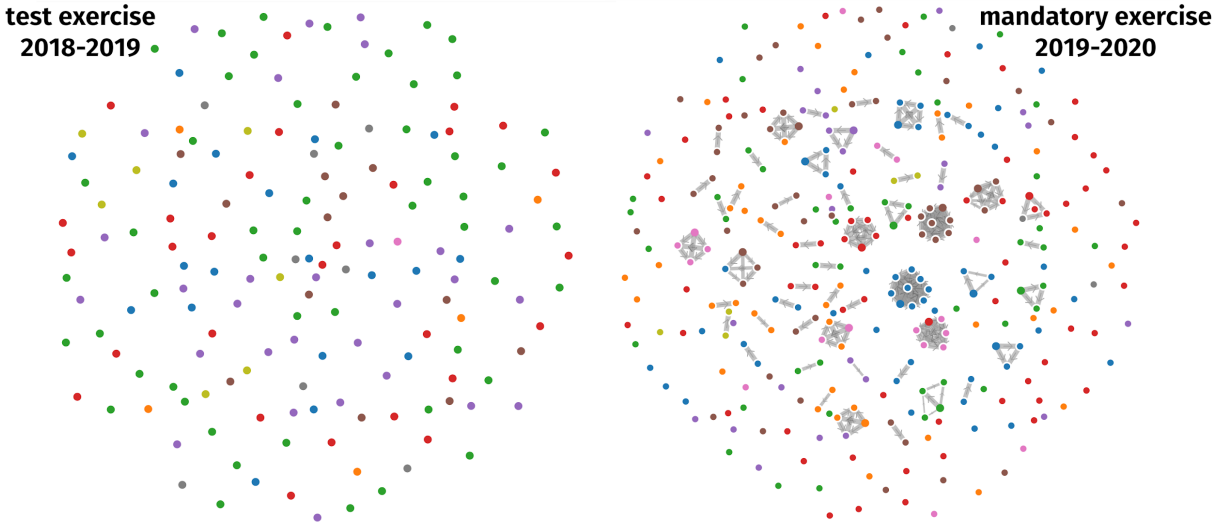

Figure 8: Dolos plagiarism graphs for the same Python programming exercise, created for a test of the 2018-2019 edition of the course (left) and reused as a mandatory exercise in the 2019-2020 edition (right). Graph constructed for the last submission before the deadline of 162 and 311 students respectively. Node colours indicate study programmes of students. Similarity threshold set to 0.5 (left) and 0.87 (right) respectively. Edge directions based on submission timestamps in Dodona. Clusters of three or more connected submissions have

one node with a solid border, indicating the first correct submission among all submissions in that cluster. All students submitted unique solutions during the exam. Submissions for the mandatory exercise show that most students work either individually or in groups of 2 or 3 students, but we also observe some clusters of four or more students that exchanged solutions and submitted them with hardly any modifications.

The goal of plagiarism detection at this stage is prevention rather than penalisation. We want students to take responsibility over their learning. The combination of realising that teachers can easily detect plagiarism and an upcoming test that evaluates if students can solve programming challenges individually, usually has an immediate and persistent effect on reducing cluster sizes in the plagiarism graphs to at most three students. At the same time, the signal is given that plagiarism detection is one of the tools we have to detect fraud during tests and exams. The entire group of students is only addressed once about plagiarism, without going into detail about how plagiarism detection itself works, because we believe that overemphasis on this topic is not very effective and explaining how it works might drive students towards spending time thinking on how they could bypass the detection process. Time they could better spend on learning to code. Every three or four years we see a persistent cluster of students exchanging code for mandatory exercises over multiple weeks. If this is the case, we individually address these students to point them again on their responsibilities, differentiating between students that share their solution and students that receive solutions from others.

Tests and exams have a well-delineated rule that verbal and digital communication is not allowed. Under normal circumstances before the COVID-19 pandemic, students are restricted in their communication because they take tests and exams on-campus under surveillance of human invigilators, but otherwise can use the Internet to consult information resources. After each test and exam, we use Moss/Dolos to detect and inspect highly similar code snippets among submitted solutions and to find convincing evidence they result from exchange of code or other forms of interpersonal communication. If we catalogue a case as plagiarism beyond reasonable doubt, the examination board is informed. When exercises that were initially created for tests or exams are reused as mandatory exercises, we generally observe a clear difference: no high-similarity pairs among solutions submitted during the test or exam and multiple high-similarity pairs found among solutions submitted for the mandatory exercise (Figure 8). This proves that tracing high-similarity pairs is a powerful way to monitor if students have been collaborating or communicating while working on programming exercises.

4.3. Impact of COVID-19 pandemic

Where we only filed a single case of suspected plagiarism during all previous editions of the course since its first first edition in 2006-2007, four cases of suspected plagiarism have been filed to the examination board during the 2020-2021 edition alone. Two of these cases occurred during tests and two during the exam. The evidence for each case was carefully documented for the examination board and we found Dolos' visualisations extremely useful for this purpose (see supplementary material for more details). The plagiarism graph illustrates that there is a high number of possible solutions and helps to convince students and board members that accidental high similarity is extremely unlikely. The compare view helps to disclose that substantial amounts of source code are identical copies or have been modified after copying as a deliberate act to obfuscate plagiarism.

When looking for explanations for this increase of plagiarism in tests and exams, the only fundamental difference in the organisation of the course is that all students took the 2020-2021 edition remotely due to the COVID-19 pandemic, including tests and exams. We switched to live Zoom sessions for lectures. Students could ask online help from teaching assistants during lab sessions, primarily using the Dodona Q&A module for questions on specific solutions or using MS Teams for general assistance. We also recommended MS Teams as an online collaboration tool, in combination with collaborative coding and pair programming services provided by modern Integrated Development Environments. Throughout the 2020-2021 edition of the course, we did not observe any significant differences in the occurrence of high-similarity pairs among solutions for the mandatory exercises.

Lack of direct human supervision during tests and exams seems the only reason for increased plagiarism. Apart from taking tests and exams remotely, the same rules applied and they had the same online nature as in all previous editions of the course. We deliberately refrained from using an online proctoring tool to remotely monitor the student's behaviour and detect irregularities during tests and exams. Mainly to avoid extra stress that students experience while following proctoring protocols for the first time, and because we believe that current proctoring tools only create a false sense of security and are too invasive on student privacy. We did introduce a sworn declaration that students had to digitally sign at the start of each test and exam (Figure 9), following a best practice recommended by Ghent University when taking remote exams. The document itself is not legally binding, but rather serves as a reminder of regulations in the Education and Examination Code that students accept when enrolling at the university. Having such an institutional honour code, actively informing students of this code, and signing these honour pledges to remember them about the code, has been observed to reduce the amount of cheating (LoSchiavo & Shatz, 2011; McCabe, Trevino, & Butterfield, 2001).

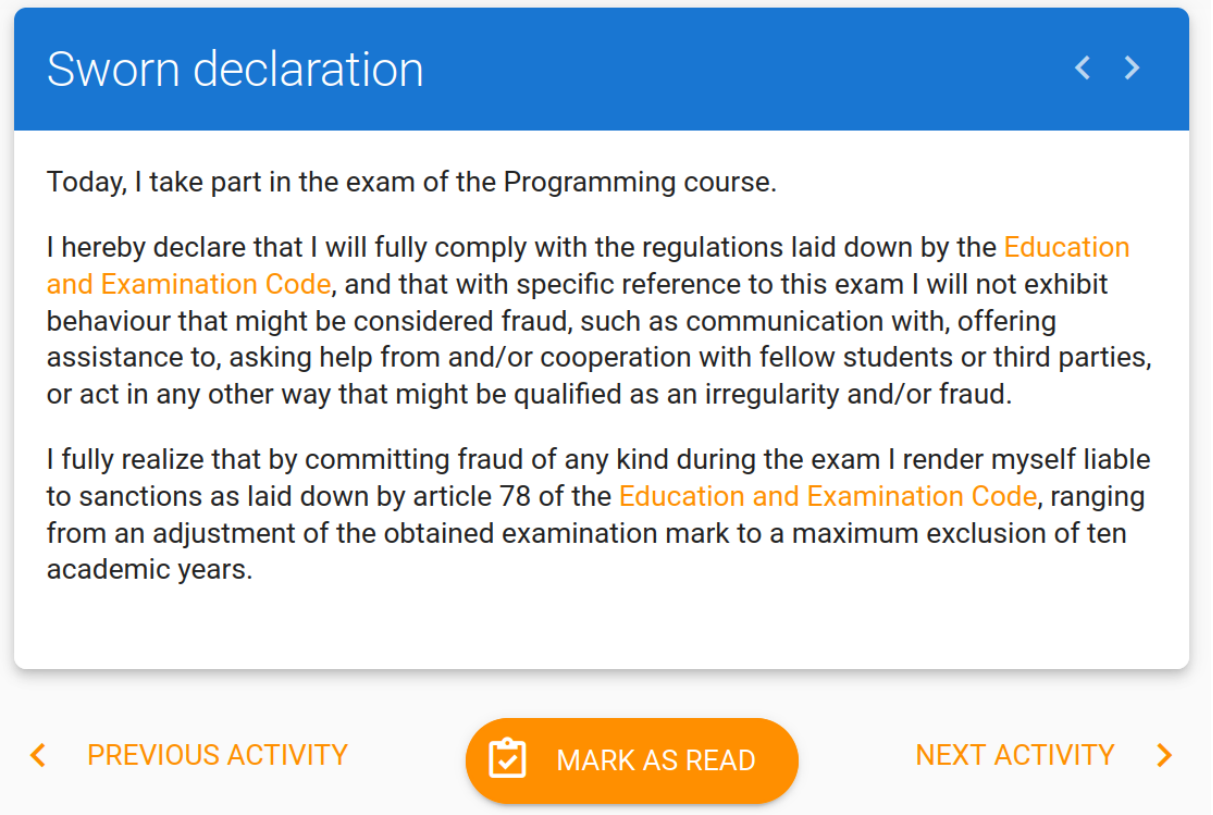

Figure 9: Sworn declaration that students have to digitally sign in Dodona at the start of each test or exam, together with a document describing the agreements for online exams. The contents of both documents are shared with students at the start of the course. The sworn declaration was newly introduced with the organisation of remote tests and exams during the COVID-19 pandemic.

In retrospect, we remain convinced that “trust, but verify” is a viable strategy for organising trustworthy assessments in open and online learning environments. Even if high-stake tests are taken remotely. On the one hand, the strategy builds on educating students about their learning behaviour and making them aware about the importance of academic integrity. We believe that four suspected cases in two tests and an exam for 440 students is acceptable. But only if cheaters get caught and appropriate disciplinary measures are imposed for proven cases of plagiarism beyond reasonable doubt. It definitely creates a ripple effect that helps to prevent plagiarism if students know about disciplinary measures imposed in recent editions of a course and about teachers having powerful tools to detect plagiarism. This brings us to verification as the second pillar of the strategy. Solid plagiarism detection relies on robust and user-friendly tools that support this otherwise tedious and challenging process. Tools also support teachers in collecting strong evidence for suspected cases, especially if it leads to students directly admitting they submitted solutions that are not their own. Otherwise proof beyond reasonable doubt remains a grey zone. Assuming that no cases of plagiarism remained unnoticed in our case study, we also dare to state that many more students were discouraged to plagiarise in a remote examination setting due to the strategy we implemented.

5. Conclusions

We strongly believe that a shift towards open and online learning and assessment environments opens up interesting avenues for more effective learning and better assessment. Risking to lower the barrier for plagiarism and cheating in digital settings does not outweigh its opportunities. Especially because the possibilities for rich data collection in online environments enable better monitoring of learning processes, including the detection of plagiarism. We shared our experiences with designing courses that have a strong focus on online learning. It enabled us to make a smooth transition to remote learning, forced by the COVID-19 pandemic, with hardly any loss in quality for teaching and learning as witnessed by formal university-wide surveys from students that have taken the courses. The reliability of online assessments has definitely been challenged when students were taking high-stake tests and exams remotely, but our “trust, but verify” approach stood well as a line of defence against source code plagiarism. We are critical to adopt current technologies for online proctoring because of privacy concerns and their impact on the assessment process itself. We feel it exposes more trust in students to have plagiarism detection tools that can run in the background, either real-time or after the fact.

With the development and validation of Dolos, we have demonstrated the feasibility of using generic parser models to build source code plagiarism tools that support a broad range of programming languages and are highly competitive with state-of-the-art tools in the accurate detection of potential cases of plagiarism. As teachers of programming courses, we have also experienced that preventing, discovering and proving plagiarism is a highly exploratory process that is well supported by its user-friendly web interface and interactive visualisations.

Dolos can be used as a standalone tool that processes source files on the local system. We plan to bundle Dolos and its dependencies into a Docker container that eases the setup for running it as a local web service. We will also host a public instance of this web service for users that are allowed to send source code to an external server. We also publish Dolos as an npm package with an open API, to ease seamless integration into third party tools. We plan to use this functionality to integrate Dolos into our own online learning system Dodona. This will free teachers that want to perform plagiarism analysis from moving collections of source files and metadata between applications, lowering the barrier even further. The source code of Dolos is made publicly available on GitHub, where we welcome any bug reports and feature requests documented as issues, user experiences shared as discussions, and contributions submitted as pull requests.

Acknowledgement

Part of this work was supported by Research Foundation - Flanders (FWO) for ELIXIR Belgium (I002819N).

References

Acampora, G., & Cosma, G. (2015). A Fuzzy-based approach to programming language independent source-code plagiarism detection. *2015 IEEE International Conference*

on Fuzzy Systems (FUZZ-IEEE), 1–8. https://doi.org/10.1109/FUZZ-IEEE.2015.7337935

Aho, A., Lam, M., Sethi, R., & Ullman, J. (2006). *Compilers: Principles, Techniques, and Tools* (2nd edition). Boston: Addison Wesley.

Ahtiainen, A., Surakka, S., & Rahikainen, M. (2006). Plaggie: GNU-licensed source code plagiarism detection engine for Java exercises. *Proceedings of the 6th Baltic Sea Conference on Computing Education Research: Koli Calling 2006*, 141–142. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/1315803.1315831

Alexandron, G., Ruipérez-Valiente, J. A., Chen, Z., Muñoz-Merino, P. J., & Pritchard, D. E. (2017). Copying@Scale: Using Harvesting Accounts for Collecting Correct Answers in a MOOC. *Computers & Education*, *108*, 96–114. https://doi.org/10.1016/j.compedu.2017.01.015

Alexandron, G., Valiente, J. A. R., & Pritchard, D. E. (2019). Towards a General Purpose Anomaly Detection Method to Identify Cheaters in Massive Open Online Courses. *In Proceedings of The 12th International Conference on Educational Data Mining*, 480–483. https://doi.org/10.35542/osf.io/wuqv5

Arwin, C., & Tahaghoghi, S. M. M. (2006). Plagiarism detection across programming languages. *Proceedings of the 29th Australasian Computer Science Conference*, *48*, 277–286. AUS: Australian Computer Society, Inc.

Batane, T. (2010). Turning to turnitin to fight plagiarism among university students. *Journal of Educational Technology & Society*, *13*(2), 1–12.

Bostock, M., Ogievetsky, V., & Heer, J. (2011). D^3 Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics*, *17*(12), 2301–2309. https://doi.org/10.1109/TVCG.2011.185

Brimble, M., & Stevenson-Clarke, P. (2005). Perceptions of the prevalence and seriousness of academic dishonesty in Australian universities. *The Australian Educational Researcher*, *32*(3), 19–44. https://doi.org/10.1007/BF03216825

Brin, S., Davis, J., & García-Molina, H. (1995). Copy detection mechanisms for digital documents. *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, 398–409. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/223784.223855

Brunsfeld, M., Thomson, P., Vera, J., Hlynskyi, A., Turnbull, P., Clem, T., … Muller, A. A. (2021). *tree-sitter/tree-sitter: V0.20.0*. Zenodo. https://doi.org/10.5281/zenodo.5044536

Chae, D.-K., Ha, J., Kim, S.-W., Kang, B., & Im, E. G. (2013). Software plagiarism detection: A graph-based approach. *Proceedings of the 22nd ACM International Conference on Information & Knowledge Management*, 1577–1580. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/2505515.2507848

Cheers, H., Lin, Y., & Smith, S. P. (2019). A Novel Approach for Detecting Logic Similarity in Plagiarised Source Code. *2019 IEEE 10th International Conference on Software Engineering and Service Science (ICSESS)*, 1–6. https://doi.org/10.1109/ICSESS47205.2019.9040752

Cheers, H., Lin, Y., & Smith, S. P. (2021). Academic Source Code Plagiarism Detection by Measuring Program Behavioral Similarity. *IEEE Access*, *9*, 50391–50412. https://doi.org/10.1109/ACCESS.2021.3069367

Chuda, D., Navrat, P., Kovacova, B., & Humay, P. (2012). The Issue of (Software) Plagiarism: A Student View. *IEEE Transactions on Education*, *55*(1), 22–28. https://doi.org/10.1109/TE.2011.2112768

Cosma, G., & Joy, M. (2012). An Approach to Source-Code Plagiarism Detection and Investigation Using Latent Semantic Analysis. *IEEE Transactions on Computers*, *61*(3), 379–394. https://doi.org/10.1109/TC.2011.223

Culwin, F., MacLeod, A., & Lancaster, T. (2001). Source code plagiarism in UK HE computing schools. *Proceedings of the 2nd Annual LTSN-ICS Conference*, 1–7. London, United Kingdom: LTSN Centre for Information and Computer Sciences.

Đurić, Z., & Gašević, D. (2013). A Source Code Similarity System for Plagiarism Detection.

The Computer Journal, *56*(1), 70–86. https://doi.org/10.1093/comjnl/bxs018

Ercegovac, Z., & Richardson, J. V. (2004). Academic Dishonesty, Plagiarism Included, in the Digital Age: A Literature Review. *College & Research Libraries*, *65*(4), 301–318. https://doi.org/10.5860/crl.65.4.301

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

Flores, E., Rosso, P., Moreno, L., & Villatoro-Tello, E. (2014). On the Detection of SOurce COde Re-use. *Proceedings of the Forum for Information Retrieval Evaluation*, 21–30. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/2824864.2824878

Gusfield, D. (1997). Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. *ACM SIGACT News*, *28*(4), 41–60. https://doi.org/10.1145/270563.571472

Hoad, T. C., & Zobel, J. (2003). Methods for identifying versioned and plagiarized documents. *Journal of the American Society for Information Science and Technology*, *54*(3), 203–215. https://doi.org/10.1002/asi.10170

Joy, M., & Luck, M. (1999). Plagiarism in programming assignments. *IEEE Transactions on Education*, *42*(2), 129–133. https://doi.org/10.1109/13.762946

Karabatsos, G. (2003). Comparing the Aberrant Response Detection Performance of Thirty-Six Person-Fit Statistics. *Applied Measurement in Education*, *16*(4), 277–298. https://doi.org/10.1207/S15324818AME1604_2

Karnalim, O., Simon, & Chivers, W. (2020). Preprocessing for Source Code Similarity Detection in Introductory Programming. *Koli Calling '20: Proceedings of the 20th Koli Calling International Conference on Computing Education Research*, 1–10. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3428029.3428065

Li, X., & Zhong, X. J. (2010). The Source Code Plagiarism Detection Using AST. *2010*

International Symposium on Intelligence Information Processing and Trusted Computing, 406–408. https://doi.org/10.1109/IPTC.2010.90

Liu, C., Chen, C., Han, J., & Yu, P. S. (2006). GPLAG: Detection of software plagiarism by program dependence graph analysis. *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 872–881. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/1150402.1150522

LoSchiavo, F. M., & Shatz, M. A. (2011). *The Impact of an Honor Code on Cheating in Online Courses*. *7*(2), 6.

Lupton, R. A., Chapman, K. J., & Weiss, J. E. (2000). International Perspective: A Cross-National Exploration of Business Students' Attitudes, Perceptions, and Tendencies Toward Academic Dishonesty. *Journal of Education for Business*, *75*(4), 231–235. https://doi.org/10.1080/08832320009599020

Mariani, L., & Micucci, D. (2012). AuDeNTES: Automatic Detection of teNtative plagiarism according to a rEference Solution. *ACM Transactions on Computing Education*, *12*(1), 2:1-2:26. https://doi.org/10.1145/2133797.2133799

McCabe, D. L., Butterfield, K. D., & Trevino, L. K. (2012). *Cheating in College: Why Students Do It and What Educators Can Do about It*. JHU Press.

McCabe, D. L., Trevino, L. K., & Butterfield, K. D. (1999). Academic Integrity in Honor Code and Non-Honor Code Environments: A Qualitative Investigation. *The Journal of Higher Education*, *70*(2), 211–234. https://doi.org/10.2307/2649128

McCabe, D. L., Trevino, L. K., & Butterfield, K. D. (2001). Cheating in Academic Institutions: A Decade of Research. *Ethics & Behavior*, *11*(3), 219–232. https://doi.org/10.1207/S15327019EB1103_2

Northcutt, C. G., Ho, A. D., & Chuang, I. L. (2016). Detecting and preventing "multiple-account" cheating in massive open online courses. *Computers & Education*, *100*, 71–80. https://doi.org/10.1016/j.compedu.2016.04.008

Novak, M., Joy, M., & Kermek, D. (2019). Source-code Similarity Detection and Detection

Tools Used in Academia: A Systematic Review. *ACM Transactions on Computing Education*, *19*(3), 27:1-27:37. https://doi.org/10.1145/3313290

Palazzo, D. J., Lee, Y.-J., Warnakulasooriya, R., & Pritchard, D. E. (2010). Patterns, correlates, and reduction of homework copying. *Physical Review Special Topics - Physics Education Research*, *6*(1), 010104. https://doi.org/10.1103/PhysRevSTPER.6.010104

Prechelt, L., Malpohl, G., & Philippsen, M. (2002). Finding Plagiarisms among a Set of Programs with JPlag. *Journal of Universal Computer Science*, *8*(11), 1016–1038. https://doi.org/10.3217/jucs-008-11-1016

Prince, M. (2004). Does Active Learning Work? A Review of the Research. *Journal of Engineering Education*, *93*(3), 223–231. https://doi.org/10.1002/j.2168-9830.2004.tb00809.x

Rahal, I., & Wielga, C. (2014). Source Code Plagiarism Detection Using Biological String Similarity Algorithms. *Journal of Information & Knowledge Management*, *13*(03), 1450028. https://doi.org/10.1142/S0219649214500282

Roy, C. K., Cordy, J. R., & Koschke, R. (2009). Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, *74*(7), 470–495. https://doi.org/10.1016/j.scico.2009.02.007

Ruiperez-Valiente, J. A., Alexandron, G., Chen, Z., & Pritchard, D. E. (2016). Using Multiple Accounts for Harvesting Solutions in MOOCs. *Proceedings of the Third (2016) ACM Conference on Learning @ Scale*, 63–70. Edinburgh Scotland UK: ACM. https://doi.org/10.1145/2876034.2876037

Schleimer, S., Wilkerson, D. S., & Aiken, A. (2003). Winnowing: Local algorithms for document fingerprinting. *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, 76–85. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/872757.872770

Sheahen, D., & Joyner, D. (2016). TAPS: A MOSS Extension for Detecting Software Plagiarism at Scale. *Proceedings of the Third (2016) ACM Conference on Learning @*

Scale, 285–288. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/2876034.2893435

Sheard, J., Dick, M., Markham, S., Macdonald, I., & Walsh, M. (2002). Cheating and plagiarism: Perceptions and practices of first year IT students. *ACM SIGCSE Bulletin*, *34*(3), 183–187. https://doi.org/10.1145/637610.544468

Shivakumar, N., & Garcia-molina, H. (1995). SCAM: A Copy Detection Mechanism for Digital Documents. *Proceedings of the 2nd International Conference in Theory and Practice of Digital Libraries (DL'95)*, 11–13. Austin, Texas.

Simon, Mason, R., Crick, T., Davenport, J. H., & Murphy, E. (2018). Language Choice in Introductory Programming Courses at Australasian and UK Universities. *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, 852–857. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3159450.3159547

Stamatatos, E. (2009). A survey of modern authorship attribution methods. *Journal of the American Society for Information Science and Technology*, *60*(3), 538–556. https://doi.org/10.1002/asi.21001

Vyverman, M., De Baets, B., Fack, V., & Dawyndt, P. (2012). Prospects and limitations of full-text index structures in genome analysis. *Nucleic Acids Research*, *40*(15), 6993–7015. https://doi.org/10.1093/nar/gks408

Weber-Wulff, D. (2019). Plagiarism detectors are a crutch, and a problem. *Nature*, *567*(7749), 435–435. https://doi.org/10.1038/d41586-019-00893-5

Whale, G. (1990). Identification of program similarity in large populations. *The Computer Journal*, *33*(2), 140–146. https://doi.org/10.1093/comjnl/33.2.140

Wise, M. J. (1993). *String Similarity via Greedy String Tiling and Running Karp−Rabin Matching*. Australia: Department of Computer Science, University of Sydney.

Zhao, J., Xia, K., Fu, Y., & Cui, B. (2015). An AST-based Code Plagiarism Detection Algorithm. *2015 10th International Conference on Broadband and Wireless Computing, Communication and Applications (BWCCA)*, 178–182.

https://doi.org/10.1109/BWCCA.2015.52