Choosing Code Segments to Exclude from Code Similarity Detection

Simon

ACM

2020-06

Simon , Karnalim , O , Sheard , J , Dema , I , Karkare , A , Leinonen , J , Liut , M & McCauley , R 2020 , Choosing Code Segments to Exclude from Code Similarity Detection . in ITiCSE-WGR '20: Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education . , 1 , ACM , pp. 1-19 , ITiCSE '20: Innovation and Technology in Computer Science Education , Trondheim , Norway , 17/06/2020 . https://doi.org/10.1145/343780

http://hdl.handle.net/10138/324361
https://doi.org/10.1145/3437800.3439201

Choosing Code Segments to Exclude from Code Similarity Detection

Simon[*]
University of Newcastle
Australia
simon@newcastle.edu.au

Oscar Karnalim[*] [†]
University of Newcastle
Australia
oscar.karnalim@uon.edu.au

Judy Sheard[*]
Monash University
Australia
judy.sheard@monash.edu

Ilir Dema
University of Toronto Mississauga
Canada
ilir.dema@utoronto.ca

Amey Karkare
Indian Institute of Technology Kanpur
India
karkare@iitk.ac.in

Juho Leinonen
University of Helsinki
Finland
juho.leinonen@helsinki.fi

Michael Liut
University of Toronto Mississauga
Canada
michael.liut@utoronto.ca

Renée McCauley
College of Charleston
USA
mccauleyr@cofc.edu

ABSTRACT

When student programs are compared for similarity as a step in the detection of academic misconduct, certain segments of code are always sure to be similar but are no cause for suspicion. Some of these segments are boilerplate code (e.g. `public static void main String [] args`) and some will be code that was provided to students as part of the assessment specification. This working group explores these and other types of code that are legitimately common in student assessments and can therefore be excluded from similarity checking.

From their own institutions, working group members collected assessment submissions that together encompass a wide variety of assessment tasks in a wide variety of programming languages. The submissions were analysed to determine what sorts of code segment arose frequently in each assessment task.

The group has found that common code can arise in programming assessment tasks when it is required for compilation purposes; when it reflects an intuitive way to undertake part or all of the task in question; when it can be legitimately copied from external sources; and when it has been suggested by people with whom many of the students have been in contact. A further finding is that the nature and size of the common code fragments vary with course level and with task complexity.

An informal survey of programming educators confirms the group's findings and gives some reasons why various educators include code when setting programming assignments.

[*]Working group leader
[†]Also with Maranatha Christian University.

ITiCSE-WGR '20, June 17–18, 2020, Trondheim, Norway
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8293-9/20/06…$15.00
https://doi.org/10.1145/3437800.3439201

CCS CONCEPTS

• Social and professional topics → Computing education; **• Information systems → Near-duplicate and plagiarism detection**.

KEYWORDS

Plagiarism; collusion; academic integrity; code similarity detection

ACM Reference Format:
Simon, Oscar Karnalim, Judy Sheard, Ilir Dema, Amey Karkare, Juho Leinonen, Michael Liut, and Renée McCauley. 2020. Choosing Code Segments to Exclude from Code Similarity Detection. In *2020 ITiCSE Working Group Reports (ITiCSE-WGR '20), June 17–18, 2020, Trondheim, Norway*. ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3437800.3439201

1 INTRODUCTION

Assessment is an integral aspect of education, an aspect that serves several distinct purposes. Under the umbrella of *formative assessment*, it is used to help students appreciate how much they have learnt and what gaps might remain in their learning. As *summative assessment*, it is used to measure the extent of a student's knowledge or ability by giving the student a mark or grade [59]. That measure can then be used for internal purposes of grading and assessment [51] or for the external purpose of certification, for example to assure prospective employers that the student has attained a certain level of knowledge or skill [57].

For the various purposes of summative assessment it is important for the assessors to be assessing the work of the student they are assessing, not of some other person or people. It follows that they require some assurance that the work is the student's, and that where the student has incorporated the work of others, that fact is clearly and appropriately acknowledged. This is the basis of academic integrity: an assurance that work submitted for assessment is, except where otherwise indicated, the work of the student being assessed.

It is well established that many university students are willing to breach academic integrity – to cheat – if the opportunity arises and the potential gains offset the potential losses [33, 52]. Violations of academic integrity have been widely researched and reported in computing courses, and particularly in programming courses [7, 18, 48, 54, 58]. An area of focus in this work has been plagiarism and collusion[1] in take-home assignments where students work on a programming task outside class and unsupervised by their teachers. This type of assessment has been shown to have the highest rates of cheating, and such cheating is considered by students to be among the most acceptable forms of academic violation [48, 49].

Much of the research in addressing academic integrity in computing courses has explored ways to educate students and discourage them from cheating [10, 50, 54]. The incidence of misconduct can be mitigated by measures such as informing students about acceptable practices [53], individualising the assessment tasks [6], or incorporating post-submission oral presentations [17]. An obvious measure is to monitor task completion by having students work in labs, under direct supervision. Student computers can be disconnected from the internet and from external drives, and monitored using appropriate software. Another measure involves the use of post-submission oral interviews or presentations, where students are selected randomly to explain their solutions to or answer questions about them. This presentation contributes to the mark for the assessment task, and the fear of being randomly chosen encourages students to do their own work.

One prevention measure that does not rely solely on students' final code submissions is looking at process data gathered from students while they are completing assignments [18, 61]. Process data can show evidence of early collusion, evidence that is not visible in students' final code submissions. For example, in some courses, one member of this working group recorded students' programming process at the keystroke level; this allowed easy identification of copy-pasting and the possibility of comparing just the copy-pasted content with other students' programs. Another member used a web-based system that stores intermediate versions of student programs at regular intervals as they type, and marks an intermediate version as 'pasted' if a large amount of code is added in a small interval. These intermediate versions give insight into a student's thought process, but have also been used to detect cases of plagiarism or collusion. A special case of process data is keystroke data, which is sometimes collected for misconduct detection purposes [29]. Such data can be used to uniquely identify students based on their typing patterns [31], and can be used, for example, to identify students in remote exams [29, 55], helping to detect cases where a student has somebody else complete the exam for them.

While this work is valuable in that it works on preventing the problem rather having to deal with the consequences, these approaches will never deter all students in all circumstances, so there is also much work on detection of cheating in its various forms [54].

With take-home programming assignments, detection of cheating focuses largely on the use of code similarity detection tools by programming educators to help ascertain whether different students' programs were written independently. Several studies [5, 28, 42] report that the use of such tools has many positive effects. They can reduce administrative workload and neutralise the lecturers' and tutors' positions regarding academic dishonesty. Furthermore, they can act as a discouragement to cheating.

Code similarity detection tools typically provide the teachers with a listing of similarity measures for pairs or groups of assignments and highlight the areas of similarity. The output from these tools helps teachers to decide whether two or more programs are more similar than one would expect from coincidence alone. However, interpreting the output is not straightforward: when code similarity is detected, it is not a foregone conclusion that the students have cheated. There are many positive ways in which code copying can be used as part of programming education. For example, code can be provided to the students as a template, or be available in course notes or the textbook. But common code segments that are expected to be present contribute to the similarity measure and confuse the interpretation of this measure for purposes of academic integrity. Removing these elements would make it easier to find cases of possible collusion or plagiarism, but it is not necessarily clear how to choose these elements. Despite this difficulty, there has been little work reported about interpreting the output of code similarity detection tools.

The aim of our working group was to explore this area, specifically investigating what code segments could be expected to arise commonly in students' submissions for an assessment task, and could be candidates for removal before a code similarity detector is used or during the investigation of suspected programs. The results of our work will help teachers to use these tools appropriately and effectively. This work is particularly timely in the current climate of covid-19, where more assessment is happening in non-invigilated situations, giving students more opportunities to cheat. Looking beyond the current virus, there are predictions that this shift is long-term and possibly permanent.

Readers should note that the word 'assessment' is used in two main senses in this report. Used as an uncountable noun ('assessment is an integral aspect of education', 'students dislike assessment') it refers to the act of assessing students' work, or the principle of so doing. Used as a countable noun ('an assessment', 'assessments') or in an adjectival sense ('an assessment task', 'an assessment unit') it refers to a task assigned to students as part of the process of assessment. For the latter sense, this report also uses the word 'assignment' somewhat interchangeably with 'assessment'.

In the next section we summarise the literature of similarity detection, first in the context of prose text and then in the context of computer programs. Section 3 explains our research questions and the methods that we used to address them. Section 4 describes the five different reasons for code commonality that emerged from our analysis, and section 5 describes five different categories of assessment task, relating them where possible to the reasons for code commonality. Section 6 presents the findings of our survey of computing educators. Sections 7 and 8 present the results of a more

[1]While some computing educators use the word 'plagiarism' for any use of code not written by the student and not acknowledged, we use 'plagiarism' for code copied from widely available external sources, and 'collusion' for work arising from unauthorised collaboration between students in the same class.

detailed analysis of a small selection of courses. Section 9 presents some reflections on the use of code similarity detection tools, and sections 10 and 11 summarise the report and suggest directions for future work.

2 BACKGROUND

Code similarity detection has a long and venerable history, both in education and in industry, but because its use is far from universal, this section will provide a brief explanation of its purpose and application. To provide some context, first we discuss the automatic detection of text similarity and then we describe the specific tools used for code similarity detection and explain their use.

2.1 Text Similarity Detectors

In recent decades, software has been developed to assist in the detection of academic misconduct. A well-known example is Turnitin[2], which compares a piece of writing submitted by a student with a vast number of written passages collected from the internet, drawing attention to any common passages. Turnitin is often described as plagiarism detection software, but in fact it is text similarity detection software. Once it has detected similar passages of text, it is the responsibility of the educator to determine whether the similarity is due to plagiarism. By the nature both of its algorithms and of its collection of passages, Turnitin is designed to detect similarity in passages of prose text, and not in, say, images or computer programs.

As an aside, while the apparent intention of educators who use text similarity detection tools is to ensure that they are marking the students' own work, it might be more accurate to say they are ensuring that they are marking the students' own expression of what they wish to write. It is unlikely that educators expect undergraduate students to produce original ideas; rather, they expect the students to use their own words in synthesising and summarising the ideas of others. This is an idea that many students have difficulty with, as they know that they will not be able to express the ideas as well as the authors on whose work they are drawing.

Turnitin does not work for computer programs, partly because its database of writing does not appear to include computer programs. This is to be expected, as computer programs differ substantially from prose text. Unlike most prose text, some program parts, such as comments and identifier names, do not contribute to the inherent structure of the program. In addition, program statements have far less variation than human sentences due to the tighter syntactic constraints of programming languages. Furthermore, some program tokens do not need to be delimited by white space or punctuation; for example, "x=x+1;".

Even if it were possible to co-opt text matching software to deal with computer programs, it would probably not be useful. For example, simple changes to comments and variable names could be used to reduce the levels of similarity detected by Turnitin. Finally, programming students often collude rather than plagiarising. Therefore the educator's goal is typically to compare students' programs with the programs submitted by other students for the same assessment item, rather than with large numbers of code samples scraped from various web pages.

[2]www.turnitin.com

2.2 Program Code Similarity Detectors

Software developers have been writing code similarity detection tools for more than five decades [38]. Although they vary substantially in their detection techniques [25], the tools generally work in two consecutive phases [23]. First, student submissions are preprocessed to an intermediate format, such as feature vector [15], token string [45], syntax tree [24], or program dependency graph [30]. This preprocessing typically generalises aspects of the code that are subject to trivial modifications, thus nullifying the effect of many simple disguises of copied code. Second, the preprocessed submissions are compared with one another. The comparison itself can take many forms, including string-matching algorithms [45] and graph isomorphism checks [30].

Some tools are publicly available, and according to Novak et al. [38], JPlag [45] is the tool most often mentioned in the literature, followed by MOSS [47] and Sherlock [22].

JPlag [45] was designed in 2001 as a web service, but was subsequently redesigned as a standalone offline tool. It converts student submissions to token strings and then pairwise compares the strings with a greedy string-tiling algorithm [60]. It can currently handle six programming languages: Java, Python, C, C++, C#, and Scheme.

MOSS [47] was introduced in 1997 with winnowing as the similarity algorithm. The tool works as a web service: student submissions are uploaded, and the similarity report is accessible online via a unique link for up to 14 days. MOSS covers the widest array of programming languages (25), including all of those supported by JPlag. A complete list of the languages can be seen on its home page[3].

Sherlock [22] was introduced in 1999, and updated in 2005 [37] to enhance its efficiency. It is a standalone GUI application, covering two programming languages: Java and C++. For detecting similarity, it tokenises student submissions and compares them using a modified version of the longest common subsequence algorithm. This tool is from the University of Warwick, and is not to be confused with another tool of the same name, from the University of Sydney [4, 46].

2.3 Effectiveness of Code Similarity Tools

Several comparative studies have been performed to try to determine the 'best' code similarity detection tools. Qualitatively speaking, there is no 'silver bullet' in code similarity detection [16, 36]; each tool has its own benefits and drawbacks. In terms of accuracy, some studies found MOSS to be the most effective [2, 19], while another found JPlag to have comparable accuracy [16]. The difference is not surprising: the optimal configuration of a code similarity detection tools is very sensitive to the data set on which it is used [46].

Mann and Frew [32] point out that, particularly in early programming courses, most programs to achieve the same task will have a great deal in common; that program similarity is not necessarily an indicator of program copying or unauthorised assistance. Nevertheless, particularly in large classes, some sort of automatic filtering of programs is almost essential to distinguish those programs that are clearly independent from those that merit further examination by a human to determine whether they might have a common source.

[3]https://theory.stanford.edu/~aiken/moss/

Culwin and Lancaster [9] propose four stages for detecting plagiarism and collusion with a similarity detection tool. First, student submissions are collected either manually or with the help of learning management systems such as Moodle [8], Google Classroom [62], or BOSS [21]. Second, the submissions are passed to the detection tool for similarity analysis. Third, the tool's similarity report is confirmed by the educator, checking whether the submissions flagged as suspicious really do have a high degree of similarity. Fourth, the suspected submissions are manually investigated by the educator, and all submissions whose similarity appears to indicate academic misconduct are singled out. These stages were initially formulated in the context of text-based assessments, but Mišić et al. [35] argue that they are also applicable for programming assessments.

The last two stages (similarity confirmation and investigation) require human intervention, and their accuracy and completion time are heavily affected by the number of false positives generated by the tool; that is, student submissions that are marked as suspicious when they have not actually been copied. This number can be high if the tool's preprocessing nullifies too much program variation while the assessments are too simple, as in first-year courses [14, 20]; when they use a restrictive programming language, such as SQL [27] or Verilog [44]; or when they incorporate common 'starter code' provided for them to use [3].

Besides appreciating that surface variation is sometimes all that can distinguish between independent programs [14, 20, 22, 56], the number of false positives can also be reduced by excluding irrelevant code segments, those that should not contribute to suspicion as they occur with good reason in most student submissions; for example, simple code, code needed for compilation, and starter code. Exclusion of code segments is supported by several tools [3, 12, 26, 43], including JPlag [45] and MOSS [47]. Typically, the tools are given the relevant code segments as starter code, and the similarity algorithm excludes them from being marked as matched segments.

Such code segments can also be selected automatically on the basis of their distribution across the whole body of student assignments [13]: segments whose frequency of occurrence exceeds a specified threshold can be deemed as irrelevant. This technique is applied in MOSS with the *m* option, which specifies how many times a code segment may appear before it is ignored.

Though automated selection of irrelevant segments can be more practical than manual selection, determination of the frequency threshold for those segments can be somewhat arbitrary, and depends heavily on the assessment design. This is exacerbated by the fact that not all student submissions are compilable and comply with the assessment specifications, such as the use of starter code or specified libraries. Furthermore, not all code similarity detection software offers automated detection of common segments.

3 RESEARCH QUESTIONS AND METHOD

In this section we first present our research questions and then describe the four-phase method that we took to answer them.

3.1 Research Questions

In this work we investigate students' software solutions in a range of programming languages to identify common code segments that are unlikely to indicate collusion or plagiarism, and to find generic descriptors for these segments that will help others to identify them. We also investigate the impact of eliminating these segments from students' solutions to determine whether that elimination can improve the detection of suspicious code through the use of code similarity tools.

Our research questions are as follows:

- RQ1 What kinds of common code segment should be excluded from code similarity detection?
- RQ2 How does removing common code segments affect the reports generated by code similarity detection tools?

Our aim for the first research question is to expedite the process of code similarity detection for educators. If we can identify the kinds of common code segment that can be excluded from similarity detection, those segments can be more easily excluded from code similarity detection, which will therefore return fewer false positives. It can also reduce the risk of human error by narrowing the educator's focus to the relevant code segments.

With the second research question we aim to validate our findings for RQ1. Specifically, we will remove innocuous common code segments from programs and investigate how much difference that makes to results from two of the most frequently mentioned code similarity detection tools in the literature [38], JPlag and MOSS.

Having answered our research questions, we will outline some factors that instructors might usefully consider when applying our findings in their own courses.

3.2 Phase 1, Exploratory: Search for Common Code Segments

In the first phase of the research we searched student submissions for common code segments and attempted to classify the reasons why we would expect these to be common. We were unable to find any literature addressing these reasons, so we devised our own, using the process described below.

3.2.1 Data set. Each member of the working group collected programming assignments, labs, and tests submitted by students in courses at their own institution and attained the appropriate institutional ethics approval to use these student submissions in this research. Table 1 shows details of the courses from which assignments have been drawn in this research. The courses span all years of an undergraduate degree and cover nine programming/specification languages.

3.2.2 Reasons for similarity. With an eye to identifying code segments that might be naturally common (RQ1), the working group leaders proposed ten reasons why code segments could be expected to be common without indicating collusion or plagiarism (see table 2). This classification was based on the leaders' experience with setting and assessing programming assignments. The ten reasons were considered a starting point; it was expected that the list might be refined or extended based on findings.

In examining the students' submissions, we found the original ten reasons for expected commonality in student submissions to be

Table 1: Courses in the data set, with each ID incorporating both the initials of the course name and the year level of the course; the rightmost columns give the number of assessments that were set in the course offerings being considered and the (sometimes approximate) number of submissions received for those assessments; the courses are sorted by programming language within year level

ID	Name	Year	Language(s)	Offerings	Assessment type(s)	Total ass'ts	Total sub'ns
IP1a	Introduction to Programming	1	C	11	Lab, lab exam	36	198,000
IP1b	Introduction to Programming	1	C++	2	Lab	20	20,000
DS1	Data Structures	1	C++	1	Lab	12	577
ACP1	Advanced Course in Programming	1	Java	2	Assignment, exam	80	24,062
FP1	Foundations of Programming	1	Java	2	Assignment	2	1,140
IP1c	Introduction to Programming	1	Java	2	Assignment, exam	152	48,187
OOP12	Object-Oriented Programming	1,2	Java	2	Assignment	8	57
IP1d	Introductory Programming	1	Java, Python	3	Assignment, lab	253	8,587
ADS1	Algorithms and Data Structures 1	1	Python	1	Lab	22	1,169
IP1e	Introduction to Programming	1	Python	3	Assignment	5	693
IP1f	Introduction to Programming	1	Python	2	Assignment	8	8,712
ICS1	Introduction to Computer Science	1	Python	2	Assignment, lab	24	13,996
SP2	Systems Programming	2	C	1	Assignment, lab	14	3,308
AP2	Application Programming	2	C#	4	Assignment, lab exam	7	402
ADS2	Algorithms and Data Structures 2	2	Java	1	Lab	25	651
OOP2	Object-Oriented Programming	2	Java	1	Lab	29	916
SD2	Software Design	2	Java	1	Assignment	2	545
TC2	Theory of Computation	2	Python	1	Assignment	5	666
N3	Networking	3	C, Python	1	Assignment	3	258
PPL3	Principles of Programming Languages	3	Haskell, Prolog, Scheme	1	Assignment, lab	11	1,417
LSD3	Large System Design	3	Java	1	Assignment	5	438
IAI3	Introduction to Artificial Intelligence	3	Python	1	Assignment	12	140
NM3	Numerical Methods	3	Python	1	Assignment	9	448
IDB3	Introduction to Databases	3	SQL	5	Assignment	5	199
CG3	Computer Graphics	3	C++	1	Assignment	3	1022
IR34	Information Retrieval	3,4	Java	1	Lab	6	44
GP34	Game Programming	3,4	Java	2	Lab	21	200
ML34	Machine Learning	3,4	Java	1	Lab	9	58
PL34	Programming Languages	3,4	Scheme	2	Assignment	3	35

too fine-grained and too code-specific, and despite having taught the courses ourselves, we were often unable to choose just one of them. Discussion led to a grouping of these ten into four broader reasons as shown in table 2 and discussed in section 4. We also found one further reason that was not covered by the original ten, so we expanded the set of detailed reasons to eleven.

3.2.3 Analysis method. The working group members worked individually to identify, classify, and record examples of code similarity in submissions for the assessment items in their own courses. This means that they were familiar with the specification, assessment guidelines and rules, course content, and resources. All this knowledge was necessary to properly assess the likely reasons why code segments might be common.

Each member examined the code submissions, searching for code segments that were similar across the submissions, and identified those that appeared to fit one of the reasons for commonality as shown in table 2. The code segments searched for were at a relatively fine level of granularity, from an expression or a statement to a method or a function. The investigation was exploratory, and three main approaches were taken. One approach was to select code segments that were possible candidates for similarity from knowledge of the assignment task and the resources available to students. The second approach was to search for possible candidates by examining the reports from a code similarity detector (either MOSS [47], JPlag [45], or both) to find common segments. Some members chose the tool that they were already accustomed to using; some chose JPlag because they wanted a locally installed option so as to avoid network delays; some were unable to use MOSS for legal or ethical reasons, as that system stores personal data on servers in the United States. The third approach was to skim through the solutions to identify any code segments that seemed to be occurring frequently. Once candidate code segments were selected, their approximate frequency of occurrence was determined with a search through a larger subset of the solutions. With these three approaches the investigation was both targeted and serendipitous.

The results of this analysis are reported in section 4.

3.2.4 Quantifying the frequencies of occurrence of common segments. Primarily, group members identified common segments by manually inspecting the tool-generated reports and highlighted segments. This was a laborious and time-consuming process. Some detection tools [13, 47] automate the removal of common segments based on an exclusion threshold; any segments whose frequency of occurrence is higher than the threshold will be excluded from similarity detection. As stated in subsection 2.3, determination of such thresholds can be somewhat arbitrary since the occurrence frequencies of common segments depend heavily on the assignment design.

One of the group leaders wrote a simple script to partly automate this task by measuring the frequencies of specified code segments. This program, called Common Code Counter (CCC), takes a piece of code and a set of submissions, and identifies all submissions that contain the code. CCC covers eight languages: Java, C, C++, C#, Python, SQL, Haskell, and R.

Technically speaking, the program works in three stages. First, the common segment and the student submissions are tokenised with ANTLR [41], a process that includes the removal of both comments and white space. Second, identifiers and string literals are generalised to their corresponding token types, as these can be readily changed without altering the underlying structural similarity between two programs. Third, the student token strings are searched for the common token string with a strict substring search algorithm. A submission is considered to include the common segment if all of the segment's tokens are found in the submission in the same exact order. The strict matching condition aims to limit the number of false positives.

Although it seemed promising, CCC works well only on programs that are simple, short, and restricted. This is unsurprising, as the program's preprocessing is overly simple. For example, it does not generalise `i++;` to `i=i+1;` in Java, and it does not remove in-line statement delimiters (semicolons) in Python. Applying more sophisticated preprocessing would take a considerable amount of time, given our broad coverage of programming languages; development of CCC was therefore discontinued, as it was only ever intended to be a helper program rather than a substantial contribution of the report.

CCC was tested with five courses in our data set that have simple, short, and restricted assignments. The courses, IP1d, ADS1, ADS2, DS1, and OOP2, are first-year and second-year courses from one member's university. IP1d's assignments were generally designed to be finished in 20 minutes each; the assignments from the other courses were typically designed to be finished in less than a hour, and the specifications generally included code to be copied and/or suggestions for implementation. Section 7 details the results of the analysis.

In its current form, CCC is not a robust tool due to its high sensitivity to small modifications and its limited applicability. Nevertheless, for readers who might be interested in expanding the program, the code is available for download[4], and users are asked to acknowledge this report as the original source of the program.

3.2.5 Problems encountered. Identification of common code segments was carried out using the code similarity detection tools MOSS [47] and JPlag [45]. While we were analysing student solutions in May and June 2020, MOSS experienced a substantial increase in usage from all over the world, as more universities were going online in response to covid-19 restrictions. Consequently, we had issues with the reliability and response time of MOSS because its servers were being overwhelmed. This slowed down the progress of analysis as the members were able to use MOSS only when the server was not busy. To help overcome this problem, one author acquired a MOSS research license to analyse student submissions locally, and set up the software on a virtual machine which had the capacity to run MOSS without lag or downtime.

3.3 Phase 2, Exploratory: Assignment Styles and their Effects

In the second phase of the research we explored different styles of assignment and examined the data set to assess the influence of these styles on the likelihood of finding common code segments. In this phase we used the same data set as described in phase 1.

3.3.1 Assignment styles. While searching for common code segments and classifying them according to the reasons in table 2 we found that the frequency of code similarity appeared to be influenced by certain characteristics of the assignment itself. For example, the instructor might have provided starter code, which would therefore be expected to appear in all student submissions. Additionally, the complexity of the assignment can affect similarity detection, as can any constraints specified by the instructor. For example, variation between students' programs will naturally be lower for extremely short trivial assignments. Similarly, if an assignment requires students to read a file and output the content, there are only a limited number of possible ways for students to program that task. Based on these characteristics, we proposed five different styles of assignment that might lead to different expectations of common code segments: trivial assignments, bottlenecked assignments, assignments with starter code, broadly specified assignments, and open-ended assignments. These styles are described in table 3. Given the limited time span of an ITiCSE working group, we have not had the opportunity to validate this classification of assignment types; in its current form, it is simply the result of discussion and agreement within the group.

3.3.2 Analysis. Using the classification in table 3, each course in the data set was classified according to which styles its assignments displayed. The characteristics of each assignment style were then generalised into a form that might be helpful for other computing educators, and that might enable future research to better focus on particular styles of assignment when examining the incidence and nature of common code segments. The results of this analysis are reported in section 5.

[4]https://github.com/oscarkarnalim/ccc

Table 2: General and specific reasons why code segments might be naturally similar

General reason	The eleven detailed reasons
Code needed for compilation	Compulsory use for compiling the program
Intuitive implementation	The only likely way to accomplish the required task
	The only way taught to accomplish the required task
	Easier to memorise than other code alternatives
	Shorter than other code alternatives
	A habit from previous experience
	Suggested by the nature of the task
Legitimately copiable	The code can be copied from permitted external resources
Suggested implementation	Explicitly taught in the course
	Suggested by the teacher, tutor, or teaching assistant
[null]	Other

Table 3: Styles of assessment task; a task is not necessarily restricted to one style

Style	Description
Trivial assessment	Assessments so simple that there is very little room for variation (e.g. a program to display 'Hello, world!'). This kind of task often has only one solution alternative with limited possible variation. The common code segments are typically those for compilation and intuitive approach, perhaps along with suggested implementation, and will often comprise the whole program.
Bottlenecked assessment	Assessments so highly constrained, at least in parts, as to limit the possibility of varying solutions (e.g. a function to compute the volume of a rectangular solid). The constraints can result in common code segments for compilation, intuitive approach, and possibly suggested implementation.
Assessment with starter code	Assessments that require students to start from and adhere to the design of a given code base. The starter code is clearly a case of suggested implementation, and common code segments will also arise from code needed for compilation and possibly the intuitive approach.
Broadly specified assessment	Assessments that are given a generalised design (e.g. a program that simulates the board game snakes and ladders). Common code segments for this type of assessment will include code needed for compilation, and possibly the intuitive approach if the specification is sufficiently concrete, but are unlikely to include suggested implementation.
Open-ended assessment	Assessments with few constraints beyond the basics of what it must involve or achieve (e.g. a game with at least two moving objects, or an application that takes at least one external stimulus and has at least two functional operations). This might result in common code segments for compilation, but the breadth of specification is such that other sources of common code are likely to be rare – unless copying from external sources is permitted, in which case some of the submissions might have legitimately copiable common code segments.

3.4 Phase 3, Investigatory: Effect of Code Segment Exclusion

In phase 3 we investigated the effect that removing common code segments had on the reports produced by code similarity tools. The hypothesis was that excluding common code segments would expedite the similarity confirmation and investigation process [9], given that fewer matching segments would be identified by the similarity detection tools.

To determine the impact of removing common segments from students' solutions (RQ2), we wrote scripts to measure the differences between the tool-generated similarity reports before and after exclusion of the common segments. This was done for both JPlag and MOSS reports. The scripts were applied to submissions from several courses in the data set. The findings are reported in section 8.

Two evaluation metrics were introduced: reduced number of matched lines and reduced number of retrieved program pairs. When common code segments are excluded, the similarity report is expected to show fewer matched segments, resulting in a reduction of matched lines. This might also reduce the number of retrieved program pairs since in some program pairs, the matched segments are all common and excluded, so their removal will leave no remaining similarity between the programs. To deal with variation in code length, both metrics are normalised as in equation 1.

$$normalised = \frac{before - after}{before} \tag{1}$$

Given the large size of our data set, only a few courses were considered in this analysis. IP1a is a first-year C introduction to programming course with trivial assignments, bottlenecked assignments, and assignments with starter code. DS1 is a first-year C++ data structures course with trivial assignments in the early weeks and bottlenecked assignments with starter code in the remaining weeks. OOP2 is a second-year Java object-oriented programming course with trivial and bottlenecked assignments. GP34 is an elective Java game programming course in which the assignments either have starter code or are bottlenecked. The course is usually taken by students in the third or fourth year of their degrees.

While selecting common segments for this test, we introduced two constraints. If a segment occurs in multiple assignments (e.g. code for compilation or intuitive code), it is tested only on one assignment because the impact of removing it would be similar in the other assignments. This also applies if a program has several segments in the same category, such as class constructors in Java.

For the purpose of comparison, MOSS was set to report all matched segments per program pair except those that are excluded with code segment removal; this was done by giving the *m* argument, which defines how many times a code segment may appear before it is ignored, a value larger than the number of student submissions. Further, MOSS was set to retrieve all program pairs with at least one matched segment by giving the *n* argument a value larger than the total number of possible pair combinations, calculated as in equation 2, where *nsubs* is the number of submissions.

$$npairs = \frac{nsubs \times (nsubs - 1)}{2} \tag{2}$$

JPlag was set to retrieve all program pairs with at least one matched segment by giving the *m* argument (maximum number of retrieved pairs) a value larger than the total number of possible pair combinations. For short irrelevant segments (those that do not change the metrics on standard settings), higher sensitivity was used by assigning the *t* argument a value of 2. In addition, all Java common segments were submitted as complete class code, as that appears to be the only form in which segments can be flagged for removal.

It should be noted that the configuration of both tools was based on manual investigation of the tools' behaviour. During that investigation, we also encountered a few irregularities in the tools' removal mechanisms. However, the irregularities seem to be minor, and not to warrant the in-depth examination of the tools' implementation that would be required to explain or eliminate them. Additionally, many of the irregularities are similar to those reported by Đurić and Gašević [12], who report findings related to JPlag's code matching algorithm – the same algorithm that is used in the removal of common segments, which helps to explain why similar irregularities were observed in both studies.

3.5 Phase 4, Exploratory: Reasons for Providing Code when Setting an Assignment

In the fourth phase we conducted an informal survey of computing educators to collect data on how and why they provide students with code when setting assignments. Specifically, we asked those who did provide code with assignments to explain what sort of code they provided, and, in particular, why they chose to provide that code. The survey was sent to the *sigcse-members* mailing list, and the response from the list administration indicated that there were 1471 recipients. The results of this survey are discussed in section 6.

4 DIFFERENT TYPES OF CODE SEGMENT CONSIDERED FOR EXCLUSION

To address research question 1, we report the findings of an exploratory analysis of assignments from the courses listed in table 1. In this analysis we sought examples of common code segments for each of the general reasons listed in table 2, with a view to identifying possible candidates for code exclusion. The code segments we considered ranged in granularity from a single statement to a method or function. A thorough knowledge of the assessment task and the course in which it was used was essential for this analysis.

As explained in section 3.2.2, the eleven detailed reasons first postulated were combined into four broader reasons: code necessary for compilation, code reflecting an intuitive approach, code that can be legitimately copied from another source, and code that is suggested by somebody with whom the student has come into contact. These groups are shown in table 2, and the reporting of the findings is organised according to these groups.

4.1 Code Needed for Compilation

Some code segments are common because they are required for compilation purposes. These code segments are often language-dependent, so we will discuss in turn each of the nine languages that are covered in table 1.

In C and C++, import statements, which inform the compiler that the program uses external libraries, are common for compilation purposes; the language's built-in functions are stored as external libraries, even for simple tasks such as console input and output. The main method header is also common to all assignments that are designed to have main programs.

Import statements and main method headers are also common in Java and C# solutions, along with a class declaration due to the object-oriented nature of the language: all programs, even essentially procedural ones, are written in class files. These common segments can be automatically generated by an IDE if an appropriate one is used, in which case they will share even their fine details.

Some Java and C# compilation-purpose segments are common only with particular assignment requirements. Examples are Java's Scanner declaration for solutions requiring console input and a *try-catch* or *throws* statement for assignments requiring exception handling (e.g., reading files).

Assignments in the course ADS2, a data structures course taught in Java, require the students to use template code generated by a particular tool. In such cases, the common template code is considered as required for compilation purposes.

Common segments can also result from the use of framework-based libraries. The library libGDX, used in course GP34, requires the user to implement abstract methods for game functionalities. These methods are often implemented either in the same way, since their main purpose is just to enable a functionality, or as empty methods since Java requires all abstract methods to be implemented regardless of whether they are used.

We found no common segments for compilation purposes in the Python courses in our data set. Python requires no specific syntax for the main program entry point, and the assignments in the courses analysed were console-based and typically required no additional libraries. The same applied to the courses that use Scheme, Haskell, SQL, and Prolog.

It should be noted that segments for compilation purposes are irrelevant for raising suspicion only if they are commonly used segments. A very unusual set of import statements might help to identify collusion, for example if only two student submissions include such an import. This also applies to the listing of import statements in an uncommon order.

4.2 Intuitive Implementation

The general reason 'intuitive implementation' encompasses six of the eleven detailed reasons why code segments might be similar: the only likely way to accomplish a task; the only way taught to accomplish a task; easier to memorise than other code alternatives; shorter than other code alternatives; habit from previous experience; and suggested by the nature of the task.

Some segments are common because they are the only likely way to accomplish a task. Many of these involve simple logic, such as determining the larger of two values, traversing an array, or initialising a common data structure. These segments tend to be more common in first- and second-year assignments such as those in IP1d, IP1e, and ADS1, as students at those levels are generally new to programming.

Common segments will sometimes be strongly linked to the assignment design. In IR34, an information retrieval course, we found a common segment to calculate run-time memory in an assignment dealing with exactly that topic. Common segments can also have different scope; in data structures and object-oriented programming courses, such as FP1, ADS2, and OOP2, the segments are often standalone methods rather than part of the main method.

Some common segments, while not the only likely way to accomplish a task, are the only way that has been taught to the students. Examples include code for file input and output (IR34), data parsing (ML34), adding game resources (GP34), and defining instance structures for classification in machine learning (ML34).

Occasionally, a common segment might reflect the only technique taught to that point in the course. For example, in IP1d, the *while* loop is introduced a week before the *for* loop, and it is expected that the *while* loop will be the only looping option used during that week. This also applies to more advanced algorithms such as searching and sorting.

Some code segments might be common because they are the most obvious implementation, despite being neither the only alternative for their purpose nor the only alternative taught. These segments might be thought of as suggested by the nature of the task. For example, in an IP1d assignment that asks students to swap two variables, most of the solutions involve an additional variable, although an alternative approach not requiring an additional variable has also been taught. Another example, from IP1a, is an assignment task to check whether three lines can form a triangle. Most solutions check each possible combination of sides to determine whether the sum of the lengths of two lines is greater than the length of the third. That approach is more obvious than identifying the maximum value of the three lengths and checking whether it is smaller that the sum of the lengths of the other two sides.

The same phenomenon can be observed at a substantially greater scale in larger assignment. The major take-home assignment in course AP2 asks students to write a program to implement a particular dice game. For the display of the dice, most students extend earlier lab exercises and draw each face of each die as one or more filled circles appropriately located within a square. This leads to many common code segments with a six-way selection statement and sequences of one to six circle-drawing commands.

Obvious common segments can look more alike if the programming language has a particularly repetitive syntax, such as the use in Scheme of s-expressions, parenthesised lists with a prefix operator followed by operands, which give rise to many common segments in course PL34.

Some assignments are completed with the help of code generation from an application wizard, as in courses IR34 and GP34. The generated code segments might have minimal variation across student submissions, and can therefore be flagged as common. Examples of this phenomenon include GUI-creating code generated by Visual Studio (AP2) and getter/setter methods.

Some code segments can be accessible from expected sources and can therefore be common among student submissions if they are readily adapted to the assignment task. The segments can be copied from either the assignment specification (DS1) or the course slides (IR34, GP34). Sometimes they might be provided as images, requiring the students to rewrite the code and practise their typing and debugging skills. Common segments of this sort can also be the result of combining several shorter segments provided in the teaching materials.

Previously-created code files can also be legitimate sources of copying. An offering of the game programming course GP34 taught the students to create a simple game engine, with code that could be reused for each assignment. Likewise, some lab assignments in courses IP1a and IP1b could be completed based on previous assignment solutions, and the reuse of previously written code was permitted in IP1c and ACP1.

In some offerings of IP1c and ACP1, students were permitted at various stages to study the model solution while completing an assignment [39]. The goal was to help students when they were stuck on the assignment. In those versions of the course, parts of the model solution code became common in students' submissions.

Previous habit can also contribute to the commonness of some code segments. For example, in some advanced and elective courses (IR34, GP34, and ML34), students often include getter and setter methods in their code even though these methods are not needed. They may do this due to habits acquired in prior object-oriented programming courses.

Some segments can be common, or at least more common, at structural or surface level if they are either more self-explanatory or shorter than other alternatives. For example, in IP1d, *switch-case* is less preferred than *if-else* in cases where both are applicable; it is simpler for students to remember the latter, as it is more broadly applicable than the former. Another example is the selection of identifier names in an assessment about a warehouse management software in ACP1, with most of the student submissions having warehouse-related class names.

4.3 Suggested Implementation

Code segments can be common if their use is explicitly instructed. Some of the assessment specifications include details such as input and output patterns, guiding algorithms, expected code structure, and expected implemented algorithms.

Input and output patterns are often provided for console-based assessments. Simple input and output patterns, such as accepting four integers, result in similar code segments among student submissions. This is especially the case when the students have been taught few alternatives, as in many first-year courses, including IP1a, IP1b, IP1c, and IP1d. A similar situation can arise when the programming language is fairly simple, as is Python. On the other hand, two-dimensional graphical input and output patterns in game programming (GP34) are less likely to result in common code segments as they have many possible implementations and often involve advanced logic.

Guiding algorithms can result in similar code segments if they are either simple, very clear, or have limited implementation variants. In some assessments, as in courses IP1a, IP1b, IP1c, IP1d, and ACP1, students were given algorithms in which each statement either is very specific or has a direct translation to code.

Expected code structures are typically given in object-oriented assessments, and can tend to lead to similar code implementations. The structure commonly includes class hierarchies, class names, and method definitions with specified signatures. The structures can be delivered as detailed explanations, when the lecturer wishes to model object-oriented design without explicitly teaching it (ADS1, ADS2, DS1). Alternatively, it might be delivered as a class diagram in courses, such as OOP2, that assume the students have been taught how to read class diagrams.

Expected implemented algorithms are often given in advanced courses such as SD2, SP2, and ADS2, and can lead to similar implementations. For example, an assessment asked students to implement the divide and conquer algorithm, and most of the solutions display the same aspects of the algorithm, such as the number of recursive calls, the basis condition, and the ‘merging’ part. The same applies to various graph-processing algorithms such as breadth-first search and depth-first search.

Some common code segments can result from instructions given at the course level. In some first-year courses with Python in IP1d, most students encapsulated their main program statements as functions, as they had been told to do during the course.

Additional code files might be expected to be copied if the lecturers want the students to focus on some main task at hand and so provide other parts of the solution. Two offerings of the introductory programming course IP1d gave students an additional code file containing a function to generate a ‘static’ two-dimensional array. Many lab assignments in courses IP1a and IP1b provide code for accepting the input and displaying the output. In ACP1, the instructor provides starter code for some assignments, leaving students to complete relevant methods to achieve the desired functionality.

Moving beyond explicit instructions and files, some segments are common due to suggestions given by people who have more knowledge than the students: lecturers, tutors or teaching assistants, and senior colleagues. As we have no information about how our students interacted with their seniors, no examples can be given for this last group of people.

In our data set, many common code segments in this category were suggested by the lecturers. For example, in elective courses with object-oriented design (GP34 and ML34), getter and setter methods were suggested while the students were working on the assessments, but were not required or graded. In an offering of game programming (GP34), a wrapper class for a music player was also encouraged even though students could play background music without that class.

Segments can also be suggested by tutors or teaching assistants, and students helped by the same tutors or teaching assistants will tend to have similar segments. In one offering of IP1d, the lecturer noticed that some teaching assistants suggested starting the traversal loop at index 0 rather than 1. Subsequent discussion showed that the suggestion came from their experience in advanced courses (with Java or C#) where such loops were often required.

There is necessarily some overlap between this category and intuitive implementation. The main difference that we see is that with suggested implementation there is at least one other way of solving the task, while with intuitive implementation there is not – at least at the level of programming the students are expected to have attained. To illustrate this, a C programming task to accept four integers can be solved by using either four *scanf* statements with one *%d* or one *scanf* statement with four instances of *%d*. If the lecturer teaches both ways, but focuses on one of them, the focused one becomes a common segment by way of suggested implementation. If the lecturer teaches only one way, that approach becomes a common segment by way of intuitive implementation.

4.4 Legitimately Copiable

In some courses, particularly at the advanced level, students are permitted and perhaps encouraged to find and adapt code that is pertinent to the assessment task. In such cases, it might be expected that several students will find and use the same external source. Depending on the extent of adaptation required, segments of their programs might thus be similar, but for a reason that is acceptable according to the course rules. This does not apply to any of the courses in our data set, but does apply to other courses that we teach; for example, a major open-ended assignment in a mobile app development course, where students are required to decide for themselves what their app will do, and are encouraged to find (and to reference) publicly accessible code on which they can model their own.

4.5 Summary

This section has explained and summarised our qualitative observations about types of code segment that could be marked for exclusion from code similarity detection, based on the programming assignments that we have gathered and analysed. It is expected to help lecturers in determining which common code segments should not be used to raise suspicion of programming plagiarism and collusion and should therefore be removed from code similarity detection.

5 CODE SEGMENTS CONSIDERED FOR EXCLUSION BASED ON ASSIGNMENT CHARACTERISTICS

This section explains the five assignment styles mentioned in section 3.3.1 and suggests what types of common code segment are likely to be found in each. By way of illustration, table 4 shows which styles of assignment are found in each of the courses in our data set. While there are no clear-cut distinctions, there appears to be a tendency for the courses in earlier years to include trivial and bottlenecked assignments, while broadly specified and open-ended assignments seem more likely to be found in more advanced courses.

5.1 Trivial Assessments

Trivial assignments, which are often found in introductory courses, can be implemented by only a severely limited number of reasonable programs. Examples include writing a program that displays 'Hello, world!', writing a function that computes the volume of a sphere, writing a program that counts the vowels in a string – indeed, any programs with an accumulator pattern – and so on.

Code similarity detection is seldom worthwhile for trivial assessment tasks, because of the expectation that most or all submitted programs will be close to identical. Code would be expected to be common because it is needed for compilation, is an intuitive implementation, and is probably a suggested implementation.

One example from the introductory programming course IP1f asks students to write a function that calculates the distance travelled by a projectile given the elevation and magnitude of its initial velocity. The answer is a simple function that applies the relevant formula and returns the result. In addition, the programs should all import the relevant functions and constants from the appropriate library. Therefore all student programs will be effectively identical.

A potential benefit of code similarity detection with trivial assignments is to draw the marker's attention to submissions that are similar to one another while being markedly different from most other submissions. An assignment in IP1e asked students to determine the number of days between two dates in January or February. Of more than 150 submissions, most subtracted the day of the first date from the day of the second, then added 31 if the first date was in January and the second in February. A somewhat innovative alternative was to create a list of the numbers from the day of the first date to 31, a list of the numbers from 1 to the day of second date, and a list of the numbers from the day of the first date to the day of the second date. If the first date was in January and the second in February, the program displayed the sum of the lengths of the first two lists; otherwise it displayed the length of the third list. This approach drew clear attention to the two submissions that incorporated it.

5.2 Bottlenecked Assessments

Bottlenecked assessments are tasks that are so highly constrained as to limit the possibility of varying solutions. Examples might be to write a program to extract data from a particular file, or to write an SQL query to select all students who have a grade-point average higher than 3. The best that code similarity detection tools can offer for this category of assignments is to identify segments that have been copied and pasted directly.

For example, a second-year course in the theory of computation (TC2) asked students to write a function that solves the Dutch flag problem [11, 34] – given a number *x*, permuting a list into three sections: the numbers less than *x*, the numbers equal to *x*, and the numbers greater than *x*. As it was clear that the instructor required students to implement a divide-and-conquer approach, the function would be expected to include several recursive calls to itself.

Bottlenecking can also arise in specific sections of longer programs. As mentioned in section 4.2, a major take-home assignment in course AP2 requires students to implement a particular dice game. Most solutions involve drawing faces of dice, so it is inevitable that many of them will include six-way selection statements in which each branch contains between one and six instructions to draw circles. These common segments are only a small part of the complete solution, but are clearly a prime target for code similarity detectors. However, they represent a reasonably obvious way to implement the display of dice faces, and so should not be taken as evidence of inappropriate copying.

Expected similarities among bottlenecked submissions would be code needed for compilation, intuitive implementation, and possibly suggested implementation.

5.3 Assessments with Starter Code

Assignments that require students to build solutions from some instructor-provided code base will clearly result in solutions with commonalities. However, students are typically given more flexibility in how they build their solution from the base case than with trivial or bottlenecked assignments. For example, students may be required to follow a certain design pattern, and perhaps even to implement specific methods whose functionality is described; but the manner in which they solve the problem, employ cohesiveness and/or coupling of individual modules, and use helper methods may be entirely up to them. Assignments of this type appear to be well suited to automatic code similarity detection. Setting a low similarity threshold (e.g. less than 25% in MOSS) gives unhelpful results, essentially flagging code segments that the instructor would expect to see as being similar; but higher thresholds (e.g. above 40% in MOSS) often prove to be useful for code similarity detection. For assignments of this type, it is desirable for code similarity detection tools to permit the instructor to provide a solution and to tag the code segments to be excluded from similarity checking.

An assignment from a third-year artificial intelligence course (IAI3) requires various search heuristics to be used in the A*-path search algorithm. The starter code includes complete definitions for two heuristics and documentation specifying two more that the students are to implement. With such an assignment it would be helpful if the similarity detection software could be set to avoid capturing:

Table 4: Courses (table 1) and the styles of assessment task (table 3) that they use; each task can fall into several style categories

ID	Name	Year	Trivial	Bottle-necked	Starter code	Broadly specified	Open-ended
IP1a	Introduction to Programming	1	✓	✓	✓		
IP1b	Introduction to Programming	1	✓	✓	✓		
DS1	Data Structures	1		✓	✓		
ACP1	Advanced Course in Programming	1	✓	✓	✓		
FP1	Foundations of Programming	1				✓	✓
IP1c	Introduction to Programming	1	✓	✓	✓		
OOP12	Object-Oriented Programming	1,2	✓		✓		
IP1d	Introductory Programming	1	✓				
ADS1	Algorithms and Data Structures 1	1		✓			
IP1e	Introduction to Programming	1	✓			✓	
IP1f	Introduction to Programming	1	✓	✓			
ICS1	Introduction to Computer Science	1			✓		
SP2	Systems Programming	2		✓	✓		
AP2	Application Programming	2		✓		✓	
ADS2	Algorithms and Data Structures 2	2		✓			
OOP2	Object-Oriented Programming	2		✓			
SD2	Software Design	2					
TC2	Theory of Computation	2		✓	✓		
N3	Networking	3			✓		
PPL3	Principles of Programming Languages	3		✓	✓		
LSD3	Large System Design	3				✓	
IAI3	Introduction to Artificial Intelligence	3			✓		
NM3	Numerical Methods	3			✓		
IDB3	Introduction to Databases	3		✓			
CG3	Computer Graphics	3			✓	✓	✓
IR34	Information Retrieval	3,4			✓	✓	
GP34	Game Programming	3,4			✓	✓	✓
ML34	Machine Learning	3,4			✓	✓	
PL34	Programming Languages	3,4			✓		

- the completed function definitions
- the documentation for the functions that students need to complete
- any extra code resulting from specific instructor requirements that would lead to situations similar to those described in section 5.2.

The starter code provided in such assignments is clearly a (strongly) suggested implementation, while there remains scope for common code needed for compilation and intuitive implementation.

5.4 Broadly Specified Assessments

A broadly specified assessment might specify a design pattern or a specific set of requirements that focuses the assessment in a particular direction. For example, a design course, LSD3, provides a generalised set of technical requirements, asking students to create an application of at least three sub-systems, which are easily swappable on the fly, are diverse, and can handle stimuli received while respecting the modelled domain. These are imposed design constraints, as also found in a graphics course, CG3, whose students are required to create a game in C++ that includes at least two moving objects. The dice game of AP2 mentioned in sections 4.2 and 5.2 is another example of a broadly specified assignment. Students are given the rules of the game, but no guidance as to how it is to be implemented beyond specifying that it must be playable both by two people and by one person against the program. Assessments of this type are highly suitable for similarity detection tools. Being so broad in nature, they are unlikely to be independently implemented in substantially similar ways. Nevertheless, as explained earlier, they can be subject to common code segments of the intuitive implementation type, as well as the ever-present code needed for compilation.

5.5 Open-Ended Assessments

Open-ended assessments are ones with few or no restrictions, with very little likelihood of converging to a similar solution. In CG3, students are asked to create a game and demonstrate it at the end of the semester, with no additional requirements. In an earlier offering of IP1e, students were asked to 'create a program that does something interesting with pictures and perhaps also with sound ... you are expected to show your programming competency as well as your creativity'. Assignments of this type are highly suitable for similarity detection tools to provide the instructor with possible insights into collusion, as the developed solutions should differ dramatically. Similarity levels above about 25% are often indicative of possible collusion. There is still, of course, scope for common segments of code needed for compilation; if the assignment permits, there is also scope for legitimately copiable common code segments.

6 REASONS FOR PROVIDING CODE WHEN SETTING AN ASSESSMENT

The working group asked a number of computing educators whether they include code when setting programming tasks, and if so, why. We received eleven responses. The many helpful explanations provided in those responses can be condensed into five categories.

6.1 Time/Focus

The time allowed for the task is too short for students to complete it all, or students are not yet at a level where they would be capable of completing it all. In such cases, the instructors might provide the bulk of the required code, leaving students to work on parts that are specific to the lesson at hand. Some instructors provide the easier parts of the code, not wanting the students to waste their time on it; others provide the harder code, which is currently beyond the students' capability. In either case, respondents agreed on the importance of showing students well written code in the expectation that they will write their own code in a similar manner.

6.2 Code Already Seen

Some code provided with assignments has already been seen by the students, either verbatim or in a similar form. This might be code that was provided in the textbook, in the lecture material, or in class exercises. Alternatively, it might be a standard solution to an earlier stage of the same assigned task. When instructors show program code to their students and then ask their students to write code, there is often an expectation that students will model their own code on that provided by the instructor. This will clearly vary with the size and nature of the task. If students are asked to implement a binary tree, they will probably have been shown code to do this, and their own programs will generally be markedly similar to that code. On the other hand, if students are asked to solve a new problem that has not been discussed in class, their programs are likely to show far more variety.

6.3 Facilitating Assessment

Code can be provided to ensure that students' submissions can all be assessed in a uniform manner. This is commonly the case when students' programs are to be automatically tested and/or graded. The submissions will be run through one or more scripts testing them with a variety of inputs, and they must include the methods that the scripts seek to execute. The same situation arises, although less frequently, with manual grading, where the person doing the grading expects to call specified methods in a specified way. In all of these cases, it is common for the instructors to provide students with the skeletons of the methods in question, to try to ensure that the method signatures will remain the same in the students' submissions.

6.4 Modelling/Reading

Some instructors provide code as a model to guide students in good programming practice, or to give them experience in reading and understanding well written code. A number of respondents emphasised this reason for providing code as parts of assignments. It is important for students to be able to read code, and providing code enhances their opportunity to do that. Furthermore, the code provided is well written and thus serves as an exemplar. One respondent also observed that "When asking students to work from a blank screen, we are asking them to do design ... [which] is an advanced exercise." Another pointed out that code can be provided to teach students about language features that might not be worth explicitly teaching in a class.

6.5 Code Maintenance

Many courses deal with issues of code maintenance. In such courses, students will typically be given a large body of code and asked to make relatively minor changes to the program as a whole. A similar situation often arises in, for example, courses on operating systems and compiler design, where students are asked to add specified functionality to large and otherwise complete programs.

6.6 Reasons for Expecting Common Code

While they were asked to explain why they provide code to their students when setting assessment tasks, some respondents also explained why they would expect their students' programs to include common code segments that had not been explicitly provided. These explanations broadly aligned with our own deductions as expressed in section 3.3. Examples include the use of programming environments that create scaffold code and the use of common libraries pertinent to the assessment task.

7 FREQUENCIES OF OCCURRENCE OF COMMON CODE SEGMENTS

This section reports and analyses normalised frequencies of occurrence of common code segments, checking whether they depend heavily on the assignment design. The analysis was conducted on a small sample of courses with fairly varied assignments. Figure 1 shows that the frequencies vary across courses in terms of both the average value and the range. This is expected as each course has its own assignment design.

Assignment design seems to be a bigger factor than the complexity of the covered materials in determining the frequencies of occurrence of common segments. IP1d was expected to have the highest average occurrence frequency because it is an introductory programming course, the covered materials are so simple, and the assignments tend to be trivial. However, it turns out that DS1, a second-semester course that involves many suggested segments, has a higher average frequency of occurrence. OOP2, whose assessments also suggest specific implementations, has a higher average frequency of occurrence than ADS1, with a broader range, even though OOP2 is the more advanced of the two courses.

The most common segments are not always found in all student submissions, even if they are strongly suggested code segments such as those in ADS1 and ADS2. Because of either human error (not paying attention to the assessment specification) or lack of

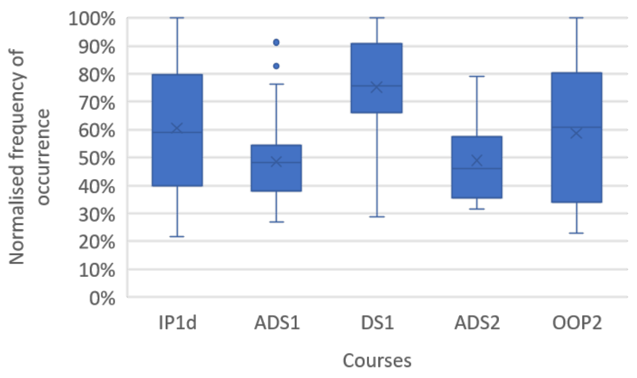

Figure 1: Distribution of normalised frequencies of occurrence for IP1d, ADS1, DS1, ADS2, and OOP2 as a box plot with whiskers representing the maximum and the minimum values excluding outliers

time, not all students copy the code into their solutions, especially when the segments are designed to be used near the end of the assessment task.

8 IMPACT OF REMOVING CODE SEGMENTS FOR EXCLUSION

We have argued in this report that common code segments of certain types should be removed from consideration by code similarity detection software, and that such removal would make it easier for the marker to assess program pairs for evidence of collusion or plagiarism. To address research question 2, we test that assertion by removing these common segments and measuring the impact on the similarity reports generated by similarity detection tools. Section 3.4 explains that the testing was carried out on a selection of four courses from our data set, and also explains the method used to determine the impact of excluding these segments from similarity detection. A number of common segments were identified for testing in the assignments of each course, as shown in table 5.

8.1 Reduced Number of Matched Lines

Fewer than half of the tested segments lead to a reduction in the number of matched lines in MOSS (figure 2). Large reductions typically occur with long common segments such as *java_classes* and *java_methods*. This is expected as longer segments mean more code that can be excluded from student submissions.

Nevertheless, the reduction is not proportional to the segment length. For example, though the *java_gettr_settr* line count is two thirds of *java_methods*'s, the former reduces less than 1% of matched lines while the latter reduces about 36%. This also applies when the length is based on source code tokens; *java_gettr_settr* still results in a far smaller reduction of matched lines than *java_input*, although their numbers of tokens are similar (54 and 55 tokens). The inconsistency might be caused by limitations of the removal mechanism: typically, the common segments are selected and excluded from student submissions via a non-optimal string matching

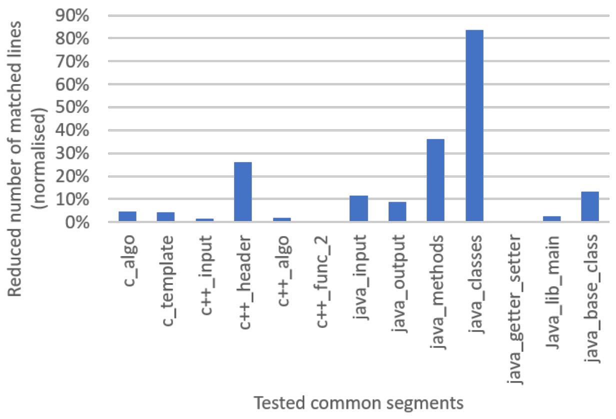

Figure 2: MOSS, reduced number of matched lines on affected code segments

algorithm without fully considering the context (e.g., whether they are within methods).

Some tested segments do not affect the number of matched lines as they are shorter than those that do show a reduction. However, it is interesting that *java_algo* shows no reduction despite having the same number of lines (10) as *java_input*. Again, this might be caused by limitations of the removal mechanism. It is also possible that the MOSS preprocessing removes or generalises specific parts of the code, shortening the segment and subsequently making it undetectable for removal.

A two-tailed paired t-test with 95% confidence rate shows that in MOSS, the reduction of matched lines is not significant across the tested common segments.

Using JPlag, figure 3 shows that most of the segments (26 of 29) lead to a reduction in the number of matched lines. Half of the segments are those that show reductions in MOSS, and in most cases the reduction seems to be more substantial with JPlag. For example, *java_classes* shows a reduction of 12% more matched lines.

Three segments do not affect the number of matched lines with JPlag. The segment *c++_import* is considered too short for starter code even with the highest sensitivity, achieved by assigning the *t* argument a value of 2. The segments *java_class* and *java_package* are both recognised as valid starter code but show no impact. It is possible that JPlag ignores the segments by default as they are expected to be common and are not related to program flow.

JPlag's removal mechanism seems to be more sensitive than that of MOSS for dealing with different kinds of common segment. On most occasions, the removal affects the resulting similarity report and the reduction is statistically significant: the p-value is less than 0.001 when applying a two-tailed paired t-test with 95% confidence rate.

As we did not validate whether all segments removed by JPlag are the common ones, JPlag's performance might be a consequence of the way it implements the removal mechanism, identifying segments as common that we would not see as common, and subsequently removing them from matching. Further investigation is required to validate this.

Table 5: Tested common segments for measuring the impact of code segment removal

ID	Description	Course	Segment type	Lines	Submissions	
					Total	Avg lines
c_algo	Simple expression	IP1a	Intuitive, suggested	14	100	15
c_loop_1	Array traversal	IP1a	Intuitive	4	129	29
c_if	If statement	IP1a	Intuitive	3	166	28
c_loop_2	Loop traversal	IP1a	Intuitive	11	95	29
c_template	Template code for linked list	IP1a	Suggested	34	85	70
c++_input	Input statements	DS1	Suggested	17	54	109
c++_header	Header file (.h)	DS1	Compilation	13	52	65
c++_func_1	List initialisation function	DS1	Suggested	4	54	132
c++_import	Import statement	DS1	Compilation	1	54	132
c++_algo	Algorithm to append item to linked list	DS1	Suggested	11	47	163
c++_func_2	Memory allocation function	DS1	Intuitive	7	47	163
c++_func_3	Memory deallocation function	DS1	Intuitive	4	47	163
java_input	Input statements	OOP2	Suggested	10	28	25
java_output	Output statements	OOP2	Suggested	5	28	25
java_class	Class declaration	OOP2	Compilation	2	28	25
java_main	Main method header	OOP2	Compilation	2	28	25
java_method	Class constructor	OOP2	Intuitive	4	26	78
java_methods	Simple arithmetic methods	OOP2	Suggested	29	25	87
java_algo	Algorithm for counting vowels	OOP2	Intuitive	10	35	37
java_classes	Trivial classes	OOP2	Intuitive	78	34	118
java_constant	Static constant declaration	OOP2	Intuitive	1	35	43
java_gettr_settr	Getters and setters	OOP2	Intuitive	19	36	337
java_try_catch	Try-catch statement	OOP2	Compilation	4	31	33
java_import	Import statement	OOP2	Compilation	1	31	33
java_package	Java package statement	GP34	Intuitive, compilation	1	8	435
java_imports	Java import statements	GP34	Compilation	3	8	435
Java_lib_main	Java main method to run a game library	GP34	Intuitive, compilation	18	8	660
java_impl_mthds	Java library required implemented methods	GP34	Compilation	17	8	660
java_base_class	Java base class given by lecturer	GP34	Intuitive	56	8	568

8.2 Reduced Number of Retrieved Program Pairs

Figure 4 shows that only six common segments reduce the number of retrieved program pairs with MOSS, and these segments are among those that have an impact on the number of matched lines. The greatest reduction occurs with *java_classes* as the assessment focuses mainly on implementing a class diagram with many simple methods, and the common segment is almost the whole of the expected solution. This is followed by *c++_header*, for the same reason.

The reduction, however, is not statistically significant according to a two-tailed paired t-test with 95% confidence rate.

When testing with JPlag, figure 5 shows that more common segments reduce the number of retrieved program pairs. Five of the segments (*c_algo*, *c++_input*, *c++_header*, *java_methods*, and *java_classes*) also reduce the number of retrieved program pairs in MOSS, but in each case except *c++_header*, JPlag leads to the greater reduction.

JPlag's reduction is statistically significant according to a two-tailed paired t-test with 95% confidence rate, with a p-value of 0.04.

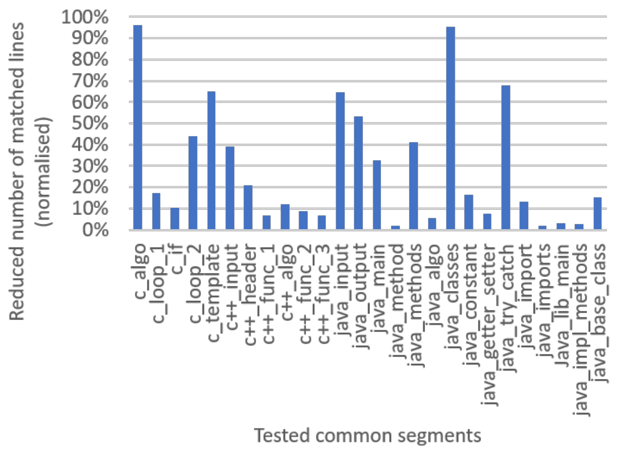

Figure 3: JPlag, reduced number of matched lines on affected code segments

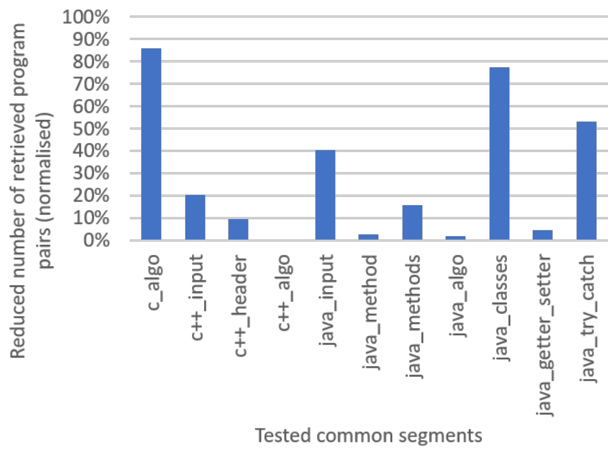

Figure 5: JPlag, reduced number of retrieved program pairs on affected code segments

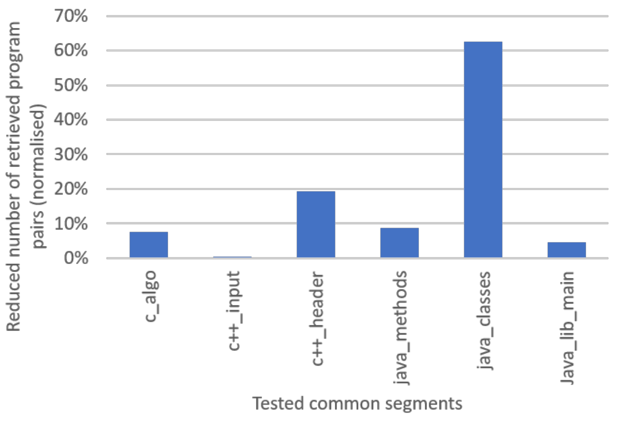

Figure 4: MOSS, reduced number of retrieved program pairs on affected code segments

Strengthening a finding from the previous subsection, JPlag seems to be more robust and/or to use a less strict removal mechanism.

9 REFLECTIONS ON USING CODE SIMILARITY DETECTORS

Code similarity detection tools can provide an efficient and effective way to check student programs for similarity, whereas manual scanning can be labour-intensive and error-prone. However, these tools vary in the features they offer, the way they detect similarity, and how the results are reported to the user. Here we provide some reflections on the tools that we used in our study.

MOSS and JPlag are two common similarity detection tools [38] that are executable via command line instructions. MOSS covers a broader range of programming languages (25), which might be useful for institutions that teach programming in many languages. It also features a number of third-party supports, some including GUI-based client applications to simplify the scripting process. Further, some studies [2, 16, 19] suggested that MOSS might be more effective than JPlag.

However, MOSS has some drawbacks that might tend to make JPlag preferable. First and foremost, in many regions and countries, including Europe and Australia, it is not legal to use MOSS, because this entails uploading personal data to servers in the United States, which has less strict data protection laws than those jurisdictions. Second, uploading student submissions and having the similarity report online might lead to other privacy issues, even though the similarity reports are not available to web crawlers (such as those used by search engines) or to other people who have not been given the link. Third, the MOSS server can be busy and respond slowly to user requests. Some of our members were unable to get a single response from the server over a period of nearly a week. While this can be resolved by adding more resources, even an abundant resource remains limited and there is no guarantee that it will be able to accommodate all requests. Fourth, MOSS requires an internet connection, which might not be available all the time, especially in developing countries.

In terms of removing irrelevant segments, JPlag seems to be more sensitive than MOSS (section 8). Further, JPlag highlights the removed segments, which can be useful in checking whether the exclusion works correctly.

Both MOSS and JPlag generalise all identifiers and remove comments and white space. This means that irregularities in identifiers, comments, and white space will not be flagged as points of similarity although they are sometimes indicative of academic misconduct. For example, suspicion on some student submissions might be enhanced if they share out-of-context variable names such as $myPersonalVariableOne$ or $mickey\_mouse$. Similarity detection tools with simpler preprocessing [14, 20, 22, 56] might be more suitable for this.

Use of third-party libraries in assessment tasks can lead to more false positives with both MOSS and JPlag. Such libraries often introduce additional quasi-keywords, but these are regarded by the tools as identifiers and are therefore replaced with more general forms. Two examples of this in our data set are IP1e, which uses Python with JES[5], and IP1b, which uses Logo [1, 40].

10 DISCUSSION

Similarity detection tools are widely used by educators to detect instances of plagiarism or collusion. The tools typically produce a measure of similarity and a report highlighting sections of code that are similar.

The similarity measure alone is a simplistic measure and should not be used to decide whether plagiarism or collusion is likely to have occurred. While the similar sections might be due to plagiarism or collusion, there are other reasons for code similarity. In our investigation of the assignments of 29 programming courses we found a number of acceptable reasons why code sections could be similar, and many examples to illustrate these reasons.

It is therefore important that the code segments identified as similar should be further investigated to determine how much of the similarity is explainable and how much is suspicious. Culwin and Lancaster [9] propose a process for detecting plagiarism and collusion with a similarity detection tool, where they recommend that educators manually perform similarity checking to check whether the resulting similarity report is logically acceptable, and then perform investigation to determine whether the suspected submissions are the result of plagiarism or collusion.

To provide a more accurate measure of suspicious similarity and to expedite the process of detection, similar code segments that are not cause for suspicion should be excluded prior to manual or automated checking. The segments can be initially selected for exclusion based on types of code segment (section 4), styles of assessment (section 5), and insight gained from computing educators' reasons for providing code when setting an assessment (section 6). A number of similarity detection tools facilitate the removal of these segments by treating them as starter code. MOSS[6] [47] and JPlag[7] [45], for example, enable the removal via *b* and *bc* options respectively. Though the removal is not always accurate, it typically reduces the observable content of the similarity report (section 8).

Some similarity detection tools are able to automatically remove segments whose frequency of occurrence exceeds a particular threshold (e.g. the *m* option in MOSS). Although this might seem more practical, we do not recommend it in general, as determining the threshold is somewhat arbitrary, while the segments' frequencies of occurrence vary across assessments and courses (section 7).

For computing educators who manually check student programs, identifying the expected common segments might still be beneficial in helping them to identify which similarities should give rise to suspicion and can subsequently be used as evidence of academic misconduct.

11 CONCLUSION AND FUTURE WORK

Not all similar code segments should give rise to suspicion of plagiarism or collusion, as some such segments are expected to be found in many student submissions. As a consequence, the results of automated code similarity detection necessarily include indications of similarity between programs that are genuinely independent. To an educator seeking to identify academic misconduct, these similarities constitute false positives. This report identifies steps that can be taken to reduce the number of such false positives.

Our first research question asked what kinds of common code segment should be excluded from code similarity detection. The report summarises the types of code segment that we found to be common but not indicative of plagiarism or collusion (section 4). We explain the reasons for their commonness and suggest in what kinds of assignment they usually occur (section 5). Excluding these segments will help to expedite the process of code similarity detection, potentially increase its accuracy, and even mitigate the risk of human error.

Our second research question asked how removing common code segments affects the reports generated by code similarity detection tools. Specified common segments can be excluded from being considered as 'matched' in many similarity detection tools, and according to our study their removal can result in fewer and shorter matched code segments (section 8). This can increase the effectiveness of code similarity detection as the removed segments do not indicate collusion, and can increase the efficiency of the manual checking as fewer matched segments are processed to be displayed. This is particularly the case with large classes, where the improvement in similarity checking should offset the initial work required to identify the expected common segments and mark them for exclusion.

The frequencies of occurrence of the segments vary according to task-specific characteristics such as complexity and the presence of starter code. We do not recommend blindly removing all segments whose frequency of occurrence is higher than a particular threshold, as determining an appropriate threshold is somewhat arbitrary and prone to human error.

Due to the short time span of an ITiCSE working group (less than six months), our study has a number of limitations that can be addressed as part of future work. First, the impact of removing common segments has not been evaluated from the user's point of view. A proper evaluation would need to consider the impact over at least one academic semester, and further time would be required to gain ethics approval, analyse the data, and write up the results. Second, the report suggests the use of existing tools in removing common segments. Further study into the limitations of those tools for that purpose might be appropriate.

Further work could also be carried out to validate some of the classifications that emerged from our work, such as the types of code segment considered for exclusion (section 4) and the different styles of assessment item (section 5).

One reviewer of this report suggested that the focus on detecting academic misconduct might be misguided, and that it might be possible for instructors to use code similarity detection to identify legitimate common code segments in order to improve their instruction. This is a tantalising thought. While all that we have read about code similarity detection tools concerns their use in investigating possible academic misconduct, it is indeed possible that other uses for the software can be found, and this might be an avenue worth exploring.

[5]http://coweb.cc.gatech.edu/mediaComp-teach
[6]https://theory.stanford.edu/~aiken/moss/
[7]https://github.com/jplag/jplag

ACKNOWLEDGMENTS

We thank Alex Aiken (Stanford University) for granting us a research license for MOSS, speeding up our research progress and allowing us to thoroughly test the large number of assessments collected for this research study. We thank Andrew Wang (University of Toronto Mississauga) for his technical support in building the testing framework. We are also grateful to Lutz Prechelt, co-creator of JPlag, for his helpful and constructive feedback on this report.

REFERENCES

[1] H Abelson and A DiSessa. 1981. *Turtle Geometry: The Computer as a Medium for Exploring Mathematics.* MIT Press, Cambridge, MA.

[2] Alireza Ahadi and Luke Mathieson. 2019. A comparison of three popular source code similarity tools for detecting student plagiarism. In *21st Australasian Computing Education Conference (ACE 2019).* 112–117. https://doi.org/10.1145/3286960.3286974

[3] Aleksi Ahtiainen, Sami Surakka, and Mikko Rahikainen. 2006. Plaggie: GNU-licensed source code plagiarism detection engine for Java exercises. In *Sixth Baltic Sea Conference on Computing Education Research (Koli Calling 2006).* 141–142. https://doi.org/10.1145/1315803.1315831

[4] França B Allyson, Maciel L Danilo, Soares M José, and Barroso C Giovanni. 2018. Sherlock N-Overlap: invasive normalization and overlap coefficient for the similarity analysis between source code. *IEEE Trans. Comput.* (2018). https://doi.org/10.1109/TC.2018.2881449

[5] Kevin W Bowyer and Lawrence O Hall. 1999. Experience using "MOSS" to detect cheating on programming assignments. In *29th Annual Frontiers in Education Conference.* https://doi.org/10.1109/FIE.1999.840376

[6] Steven Bradley. 2020. Creative assessment in programming: diversity and divergence. In *Fourth Conference on Computing Education Practice.* 13:1–13:4. https://doi.org/10.1145/3372356.3372369

[7] Daniela Chuda, Pavol Navrat, Bianka Kovacova, and Pavel Humay. 2012. The issue of (software) plagiarism: a student view. *IEEE Transactions on Education* 55, 1 (2012), 22–28. https://www.learntechlib.org/p/64510

[8] Jason R Cole and Helen Foster. 2008. *Using Moodle: Teaching with the Popular Open Source Course Management System* (2nd ed.). O'Reilly. 266 pages.

[9] Fintan Culwin and Thomas Lancaster. 2001. Visualising intra-corpal plagiarism. In *Fifth International Conference on Information Visualisation.* 289–296. https://doi.org/10.1109/IV.2001.942072

[10] Martin Dick, Judy Sheard, Cathy Bareiss, Janet Carter, Donald Joyce, Trevor Harding, and Cary Laxer. 2002. Addressing student cheating: definitions and solutions. *ACM SIGCSE Bulletin* 35, 2 (2002), 172–184.

[11] Edsger Wybe Dijkstra. 1997. *A Discipline of Programming.* Prentice Hall, USA.

[12] Zoran Đurić and Dragan Gašević. 2013. A source code similarity system for plagiarism detection. *Computer Journal* 56, 1 (2013), 70–86. https://doi.org/10.1093/comjnl/bxs018

[13] Christian Domin, Henning Pohl, and Markus Krause. 2016. Improving plagiarism detection in coding assignments by dynamic removal of common ground. In *2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems.* 1173–1179. https://doi.org/10.1145/2851581.2892512

[14] Mohamed El Bachir Menai and Nailah Salah Al-Hassoun. 2010. Similarity detection in Java programming assignments. In *Fifth International Conference on Computer Science & Education.* 356–361. https://doi.org/10.1109/ICCSE.2010.5593613

[15] Enrique Flores, Alberto Barrón-Cedeño, Lidia Moreno, and Paolo Rosso. 2015. Uncovering source code reuse in large-scale academic environments. *Computer Applications in Engineering Education* 23, 3 (2015), 383–390. https://doi.org/10.1002/cae.21608

[16] Jurriaan Hage, Peter Rademaker, and Niké van Vugt. 2011. Plagiarism detection for Java: a tool comparison. In *Computer Science Education Research Conference.* 33–46.

[17] Basel Halak and Mohammed El-Hajjar. 2016. Plagiarism detection and prevention techniques in engineering education. In *11th European Workshop on Microelectronics Education.* 1–3. https://doi.org/10.1109/EWME.2016.7496465

[18] Arto Hellas, Juho Leinonen, and Petri Ihantola. 2017. Plagiarism in take-home exams: help-seeking, collaboration, and systematic cheating. In *22nd Conference on Innovation and Technology in Computer Science Education (ITiCSE 2017).* 238–243.

[19] Daniël Heres and Jurriaan Hage. 2017. A quantitative comparison of program plagiarism detection tools. In *Sixth Computer Science Education Research Conference.* 73–82. https://doi.org/10.1145/3162087.3162101

[20] Ushio Inoue and Shuhei Wada. 2012. Detecting plagiarisms in elementary programming courses. In *Ninth International Conference on Fuzzy Systems and Knowledge Discovery.* 2308–2312. https://doi.org/10.1109/FSKD.2012.6234186

[21] Mike Joy, Nathan Griffiths, and Russell Boyatt. 2005. The Boss online submission and assessment system. *Journal on Educational Resources in Computing* 5, 3, Article 2 (2005). https://doi.org/10.1145/1163405.1163407

[22] Mike Joy and Michael Luck. 1999. Plagiarism in programming assignments. *IEEE Transactions on Education* 42, 2 (1999), 129–133. https://doi.org/10.1109/13.762946

[23] Oscar Karnalim, Setia Budi, Hapnes Toba, and Mike Joy. 2019. Source code plagiarism detection in academia with information retrieval: dataset and the observation. *Informatics in Education* 18, 2 (2019), 321–344. https://doi.org/10.15388/infedu.2019.15

[24] Oscar Karnalim and Simon. 2020. Syntax trees and information retrieval to improve code similarity detection. In *22nd Australasian Computing Education Conference (ACE 2020).* 48–55. https://doi.org/10.1145/3373165.3373171

[25] Oscar Karnalim, Simon, and William Chivers. 2019. Similarity detection techniques for academic source code plagiarism and collusion: a review. In *International Conference on Engineering, Technology and Education.* https://doi.org/10.1109/TALE48000.2019.9225953

[26] Dragutin Kermek and Matija Novak. 2016. Process model improvement for source code plagiarism detection in student programming assignments. *Informatics in Education* 15, 1 (2016), 103–126. https://doi.org/10.15388/infedu.2016.06

[27] Anthony Kleerekoper and Andrew Schofield. 2019. The false-positive rate of automated plagiarism detection for SQL assessments. In *First UK & Ireland Computing Education Research Conference.* 6:1–6:6. https://doi.org/10.1145/3351287.3351290

[28] Thanh Tri Le Nguyen, Angela Carbone, Judy Sheard, and Margot Schuhmacher. 2013. Integrating source code plagiarism into a virtual learning environment: benefits for students and staff. In *15th Australasian Computing Education Conference (ACE 2013).* 155–164. https://doi.org/10.5555/2667199.2667216

[29] Juho Leinonen, Krista Longi, Arto Klami, Alireza Ahadi, and Arto Vihavainen. 2016. Typing patterns and authentication in practical programming exams. In *21st Conference on Innovation and Technology in Computer Science Education (ITiCSE 2016).* 160–165.

[30] Chao Liu, Chen Chen, Jiawei Han, and Philip S Yu. 2006. Gplag: detection of software plagiarism by program dependence graph analysis. In *12th International Conference on Knowledge Discovery and Data Mining.* 872. https://doi.org/10.1145/1150402.1150522

[31] Krista Longi, Juho Leinonen, Henrik Nygren, Joni Salmi, Arto Klami, and Arto Vihavainen. 2015. Identification of programmers from typing patterns. In *15th Koli Calling Conference on Computing Education Research (Koli Calling 2015).* 60–67.

[32] Samuel Mann and Zelda Frew. 2006. Similarity and originality in code: plagiarism and normal variation in student assignments. In *Eighth Australasian Computing Education Conference (ACE 2006).* 143–150.

[33] Donald L McCabe, Linda Klebe Treviño, and Kenneth D Butterfield. 2001. Cheating in academic institutions: a decade of research. *Ethics & Behavior* 11, 3 (2001), 219–232.

[34] Colin L McMaster. 1978. An analysis of algorithms for the Dutch national flag problem. *Commun. ACM* 21, 10 (1978), 842–846. https://doi.org/10.1145/359619.359629

[35] Marko J Mišić, Jelica Ž Protić, and Milo V Tomašević. 2017. Improving source code plagiarism detection: lessons learned. In *25th Telecommunication Forum.* 1–8. https://doi.org/10.1109/TELFOR.2017.8249481

[36] Phatludi Modiba, Vreda Pieterse, and Bertram Haskins. 2016. Evaluating plagiarism detection software for introductory programming assignments. In *Computer Science Education Research Conference.* 37–46. https://doi.org/10.1145/2998551.2998558

[37] Maxim Mozgovoy, Kimmo Fredriksson, Daniel White, Mike Joy, and Erkki Sutinen. 2005. Fast plagiarism detection system. In *International Symposium on String Processing and Information Retrieval.* 267–270. https://doi.org/10.1007/11575832_30

[38] Matija Novak, Mike Joy, and Dragutin Kermek. 2019. Source-code similarity detection and detection tools used in academia: a systematic review. *ACM Transactions on Computing Education* 19, 3 (2019), 27:1–27:37. https://doi.org/10.1145/3313290

[39] Henrik Nygren, Juho Leinonen, Nea Pirttinen, Antti Leinonen, and Arto Hellas. 2019. Experimenting with model solutions as a support mechanism. In *First UK & Ireland Computing Education Research Conference.* 1–7.

[40] Seymour Papert. 1980. *Mindstorms: Children, Computers, and Powerful Ideas.* Basic Books, Inc., USA.

[41] Terence Parr. 2013. *The Definitive ANTLR 4 Reference.* Pragmatic Bookshelf.

[42] Dieter Pawelczak. 2018. Benefits and drawbacks of source code plagiarism detection in engineering education. In *Global Engineering Education Conference (EDUCON).* 1048–1056. https://doi.org/10.1109/EDUCON.2018.8363346

[43] Jonathan YH Poon, Kazunari Sugiyama, Yee Fan Tan, and Min-Yen Kan. 2012. Instructor-centric source code plagiarism detection and plagiarism corpus. In *17th Conference on Innovation and Technology in Computer Science Education (ITiCSE 2012).* 122–127. https://doi.org/10.1145/2325296.2325328

[44] James F Power and John Waldron. 2020. Calibration and analysis of source code similarity measures for Verilog hardware description language projects. In *51st SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2020).* 420–426. https://doi.org/10.1145/3328778.3366928

[45] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. 2002. Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science* 8, 11 (2002), 1016–1038.

[46] Chaiyong Ragkhitwetsagul, Jens Krinke, and David Clark. 2018. A comparison of code similarity analysers. *Empirical Software Engineering* 23, 4 (2018), 2464–2519. https://doi.org/10.1007/s10664-017-9564-7

[47] Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. 2003. Winnowing: local algorithms for document fingerprinting. In *International Conference on Management of Data*. 76–85. https://doi.org/10.1145/872757.872770

[48] Judy Sheard and Martin Dick. 2011. Computing student practices of cheating and plagiarism: a decade of change. In *16th Conference on Innovation and Technology in Computer Science Education (ITiCSE 2011)*. 233–237. https://doi.org/10.1145/1999747.1999813

[49] Judy Sheard, Selby Markham, and Martin Dick. 2003. Investigating differences in cheating behaviours of IT undergraduate and graduate students: the maturity and motivation factors. *Higher Education Research & Development* 22, 1 (2003), 91–108. https://doi.org/10.1080/0729436032000056526 arXiv:https://doi.org/10.1080/0729436032000056526

[50] Judy Sheard, Simon, Matthew Butler, Katrina Falkner, Michael Morgan, and Amali Weerasinghe. 2017. Strategies for maintaining academic integrity in first-year computing courses. In *22nd Conference on Innovation and Technology in Computer Science Education (ITiCSE 2017)*. 244–249. https://doi.org/10.1145/3059009.3059064

[51] Judy Sheard, Simon, Angela Carbone, Daryl D'Souza, and Margaret Hamilton. 2013. Assessment of programming: pedagogical foundations of exams. In *18th Conference on Innovation and Technology in Computer Science Education (ITiCSE 2013)*. 141–146. https://doi.org/10.1145/2462476.2465586

[52] Simon. 2005. Assessment in online courses: some questions and a novel technique. In *Higher Education in a Changing World: Research and Development in Higher Education (HERDSA 2005)*. 500–506. http://www.herdsa.org.au/publications/conference-proceedings/research-and-development-higher-education-higher-education-106

[53] Simon, Trina Myers, Dianna Hardy, and Raina Mason. 2019. Variations on a theme: academic integrity and program code. In *21st Australasian Computing Education Conference (ACE 2019)*. 56–63. https://doi.org/10.1145/3286960.3286967

[54] Simon, Judy Sheard, Michael Morgan, Andrew Petersen, Amber Settle, Jane Sinclair, Gerry Cross, and Charles Riedesel. 2016. Negotiating the maze of academic integrity in computing education. In *2016 ITiCSE Working Group Reports (ITiCSE-WGR 2016)*. 57–80. https://doi.org/10.1145/3024906.3024910

[55] John C Stewart, John V Monaco, Sung-Hyuk Cha, and Charles C Tappert. 2011. An investigation of keystroke and stylometry traits for authenticating online test takers. In *International Joint Conference on Biometrics (IJCB)*. 1–7.

[56] Lisan Sulistiani and Oscar Karnalim. 2019. ES-Plag: efficient and sensitive source code plagiarism detection tool for academic environment. *Computer Applications in Engineering Education* 27, 1 (2019), 166–182. https://doi.org/10.1002/cae.22066

[57] Maddalena Taras. 2005. Assessment – summative and formative – some theoretical reflections. *British Journal of Educational Studies* 53, 4 (2005), 466–478. https://doi.org/10.1111/j.1467-8527.2005.00307.x

[58] Dieter Vogts. 2009. Plagiarising of source code by novice programmers a "cry for help"?. In *Research Conference of the South African Institute of Computer Scientists and Information Technologists (SAICSIT 2009)*. 141–149. https://doi.org/10.1145/1632149.1632168

[59] Dylan Wiliam and Paul Black. 1996. Meanings and consequences: a basis for distinguishing formative and summative functions of assessment? *British Educational Research Journal* 22, 5 (1996), 537–548. https://doi.org/10.1080/0141192960220502

[60] Michael J Wise. 1996. YAP3: improved detection of similarities in computer program and other texts. In *27th SIGCSE Technical Symposium on Computer Science Education (SIGCSE 1996)*. 130–134. https://doi.org/10.1145/236452.236525

[61] Lisa Yan, Nick McKeown, Mehran Sahami, and Chris Piech. 2018. TMOSS: using intermediate assignment work to understand excessive collaboration in large classes. In *49th SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2018)*. 110–115.

[62] Michael Zhang. 2016. *Teaching with Google Classroom*. Packt Publishing Ltd. 256 pages.