Educating Students about Programming Plagiarism and Collusion via Formative Feedback

OSCAR KARNALIM*, University of Newcastle, Australia and Universitas Kristen Maranatha, Indonesia
SIMON, Unaffiliated, Australia
WILLIAM CHIVERS, University of Newcastle, Australia
BILLY SUSANTO PANCA, Universitas Kristen Maranatha, Indonesia

To help address programming plagiarism and collusion, students should be informed about acceptable practices and about program similarity, both coincidental and non-coincidental. However, current approaches are usually manual, brief, and delivered well before students are in a situation where they might commit academic misconduct. This paper presents an assessment submission system with automated, personalized, and timely formative feedback that can be used in institutions that apply some leniency in early instances of plagiarism and collusion. If a student's submission shares coincidental or non-coincidental similarity with other submissions, personalized similarity reports are generated for the involved submissions and the students are expected to explain the similarity and resubmit the work. Otherwise, a report simulating similarities is sent just to the author of the submitted program to enhance their knowledge. Results from two quasi-experiments involving two academic semesters suggest that students with our approach are more aware of programming plagiarism and collusion, including the futility of some program disguises. Further, their submitted programs have lower similarity even at the level of program flow, suggesting that they are less likely to have engaged in programming plagiarism and collusion. Student behavior while using the system is also analyzed based on the statistics of the generated reports and student justifications for the reported similarities.

CCS Concepts: • **Social and professional topics → Computing education**; Intellectual property; • **Information systems → Near-duplicate and plagiarism detection**.

Additional Key Words and Phrases: formative feedback, code similarity, programming, plagiarism, collusion, computing education

1 INTRODUCTION

Academic integrity is evidenced when students fully utilize all learning opportunities, apply their best efforts to completing assessment tasks, and engage productively with people at their institution [32]. Breaching academic integrity is a concern, and is widespread among students [1]. Roberts [43] discovered that at his institution, such misconduct occurs most frequently in computing courses. This might be due to the specific nature of programming assessments, in which the requirements of academic integrity are often interpreted differently [48].

To maintain academic integrity in programming, students must at least be educated about acceptable practices [51], and are then liable to be penalized if they perpetrate academic misconduct. Student education can take

Authors' addresses: Oscar Karnalim, oscar.karnalim@uon.edu.au, University of Newcastle, University Drive, Callaghan, New South Wales, Australia, 2308 and Universitas Kristen Maranatha, Surya Sumantri Street No.65, Bandung, West Java, Indonesia, 40164; Simon, simon.unshod@gmail.com, Unaffiliated, Australia; William Chivers, william.chivers@newcastle.edu.au, University of Newcastle, University Dr, Callaghan, New South Wales, Australia, 2308; Billy Susanto Panca, billy.sp@it.maranatha.edu, Universitas Kristen Maranatha, Surya Sumantri Street No.65, Bandung, West Java, Indonesia, 40164.

1946-6226/2022/1-ART1 $15.00
https://doi.org/10.1145/3506717

place at any time, but is typically placed either at the beginning of the course or after a new assessment task is issued. Penalizing students usually happens after all student submissions have been collected, if there is evidence that the cases are not a result of coincidence [21]. This evidence typically consists of undue and/or uncommon similarities shared among suspected programs, which can be conveniently flagged by a similarity detector such as JPlag [42] or MOSS [46].

The common approach to educating students about programming plagiarism and collusion has at least three limitations. First, it relies heavily on human effort, which can be demanding for the instructors. Second, its information tends to be unduly general and is presented before students actually undertake the assessment, so some students may not see the connection with their own particular cases. Third, there is no clear way of warning students about plagiarism or collusion when they are potentially committing it, the most crucial time in determining whether misconduct will occur.

In response to these limitations, this paper presents an approach in which personalized similarity feedback is generated automatically and immediately for each submission. If the submitted program is sufficiently similar to the most recent submissions of other students, all involved students will receive similarity reports, each personalized to their own program. Further, they are encouraged to resubmit and explain why the similarity arises. If there are no such similarities, the submitter will get a report that simulates common similarities, with comparable information to enhance their knowledge.

This gives rise to three research questions:

RQ1 Are students subjected to the immediate similarity feedback approach more aware of programming plagiarism and collusion?
RQ2 Are students subjected to the immediate similarity feedback approach more aware of common yet futile program disguises?
RQ3 Are students subjected to the immediate similarity feedback approach less likely to engage in programming plagiarism and collusion?

2 LITERATURE REVIEW

Our approach generates similarity feedback to educate students about programming plagiarism and collusion. Hence we review three streams of the literature: programming plagiarism and collusion, code similarity detectors, and formative feedback generators in programming.

2.1 Programming Plagiarism and Collusion

Programming plagiarism and collusion involve the reuse of code segments without properly acknowledging the source; the difference is that with collusion, the original authors are aware of the copying [13]. Albluwi [1] summarizes recent studies on the topic, grouping them according to which side of the fraud triangle [6] they primarily address. According to the theory, fraud (in this case, plagiarism or collusion) takes place when there are both the opportunity and the pressure to commit fraud, and a rationale can be provided to justify the misbehavior. Misconduct can be prevented by removing at least one of the sides (opportunity, pressure, and rationalization).

There are many reports on attempts to reduce the opportunity to cheat, perhaps because it is the easiest of the three to control. Instructors can make it harder to cheat in their assessments by allowing students to choose their own case studies [5] or by randomly generating different versions of the assessment task [12].

Some studies propose additional grading methods to confirm the authorship of the submitted work. Grunwald et al. [17] introduce one-to-one interviews between instructors and students. Halak and El-Hajjar [18] require students to do oral presentations of their work.

To further reduce the opportunity to cheat, code similarity detectors such as MOSS [46] are often used. These tools automatically identify programs with undue similarity, which are then subject to manual investigation

[29]. If the similarity is confirmed to be a result of academic misconduct, all involved students will be penalized according to the course's policy. Details about how these detectors work can be seen in subsection 2.2.

Pressure, the second side of the fraud triangle, is addressed by only a few studies. Spacco et al. [53], for example, address time pressure by incentivizing early submission. To alleviate pressure caused by the difficulty of the work, Allen et al. [2] propose an approach that involves many small assessments rather than a few large ones.

Rationalization, the third side of the fraud triangle, is typically addressed by informing students about the issues [51]. Instructors are advised to explicitly state their expectations regarding academic integrity and how those expectations apply in each course.

Some authors recommend including education about academic integrity in the ethics components that are taught as part of the computing curriculum [15]. However, since this is not always effective, some automated tools have been developed to enhance students' education about the issue. Tsang et al. [56] developed a mobile application to educate students about ethics via modules and quizzes, while a tool introduced by Le et al. [34] attempts to show the futility of copying and disguising code by reporting program similarity to students before their final submission.

2.2 Code Similarity Detectors

Novak et al. [39] provide a comprehensive review of studies related to similarity detectors, noting which of these detectors are publicly available. Common examples mentioned in the literature are JPlag [42], MOSS [46], and Sherlock from the University of Warwick [21]. JPlag and Sherlock are standalone applications and can be used offline. MOSS, on the other hand, relies on a web service whereby instructors upload student programs to the server and are then shown the similarity reports online. In general, MOSS is perhaps more practical to use since it covers the greatest number of programming languages and it requires no local installation. However, instructors should be aware that their data is uploaded to the server and stored temporarily, which can raise data privacy concerns in some countries [49].

Novak et al. [39] classify code similarity detection techniques into 16 categories; but as their classification is very fine-grained, and some types can overlap, we adapt a classification from Karnalim et al. [29], who group the techniques into four categories: attribute-counting-based, structure-based, hybrid, and creation-process-based.

Attribute-counting-based techniques measure the similarity by comparing frequencies of occurrence of program attributes. An early technique by Ottenstein [40], for example, considers two programs as similar if they share the same numbers of operators and operands, while Flores et al. [11] detect similar programs using latent semantic analysis, a technique from the field of information retrieval.

Structure-based techniques rely on program structure to detect similar programs. JPlag [42] converts programs to token strings and compares them with a string-matching algorithm. A similar technique is used by MOSS [46]. Aiming for higher effectiveness, a number of techniques use more complex representations such as syntax trees [38] and program dependency graphs [35].

Hybrid techniques combine an attribute-counting-based technique with another technique that is structure-based, seeking to improve performance by combining the benefits of both techniques. Rosales et al. [44] use the result of a structure-based technique as an attribute for an attribute-counting-based technique, arguing that the combination can result in higher effectiveness. To reduce the processing time required by structure-based techniques, Mozgovoy et al. [36] first filter the input programs with an attribute-counting-based technique.

As the name indicates, creation-process-based techniques compare aspects of how programs are developed. Tahaei and Noelle [54] argue that perpetrators often change a substantial amount of code at one time, and that can be detected via their pattern of resubmission. Yan et al. [59] capture snapshots of student programs and use them as an additional consideration when detecting program similarity.

2.3 Formative Feedback Generators in Programming

Feedback can be either formative or summative [31]. Formative feedback, given while the student is still learning, aims to improve the learning process. Summative feedback, given after the due date of the assessment, is typically an assessment score and perhaps a description of how well the student has addressed the assessment criteria. Formative feedback tends to be less superficial, and can therefore be more effective for learning.

Keuning et al. [31] review formative feedback generators in programming and classify the feedback messages based on the feedback content categories of Narciss [37]: knowledge about task constraints, concepts, mistakes, how to proceed, and metacognition.

Task constraints are perhaps the first thing that students should know about when undertaking an assessment task, and there are feedback generators that provide knowledge of that type. For example, Le et al. [33] emphasize the task requirements by highlighting some keywords in the task statement when students implement a method header incorrectly, and Fischer and von Gudenberg [10] alert students when they use methods that have been explicitly barred.

Knowledge about concepts is typically represented either as an explanation of the subject matter or as an example illustrating a concept. The former can be found in Ask-Elle [14], an educational tool that suggests relevant web pages when a particular programming language construct is encountered; the latter can be found in FIT Java Tutor [16], which suggests an example of a correct solution in response to an erroneous one written by the student.

Knowledge about mistakes is the most common type of feedback found in programming educational tools. Jurado et al. [22] customize the Eclipse environment to report the output of JUnit tests as feedback following test execution. Truong et al. [55] identify incorrect program constructs by comparing the student's solution with a model solution.

Once a student fully understands why their work is wrong, they need to know how to fix it, and here the feedback must address knowledge about how to proceed. Keuning et al. [30] developed an intelligent tutoring tool for code refactoring. Another tool, presented by Antonucci et al. [4], gradually reveals larger portions of the model solution along with incremental hints.

Knowledge about metacognition is knowledge about why a particular type of thinking is used to complete the work. Keuning et al. [31] found only one programming educational tool supporting such feedback: HabiPro [57], which simulates a student responding to a solution.

3 THE ASSESSMENT SUBMISSION SYSTEM

Our approach is embodied in an assessment submission system that educates students about programming plagiarism and collusion by way of automated, personalized, and timely formative feedback. As the system is fully automated, instructors are not required to spend a great deal of time in educating students about academic integrity. Further, the system's feedback should be easy to digest as it is personalized to each student based on the program they submitted. In addition, the system effectively warns students when they are potentially committing academic misconduct since the feedback is provided immediately after the submission. If a large part of the code is reported as similar, the student might experience guilt or a fear of being caught. In response, some students might be freshly motivated to act with integrity and to write their own program for the next submission.

The assessment submission system deals with the rationalization side of the fraud triangle [6] by informing students about programming plagiarism and collusion and showing them the futility of copying and disguising code. Unlike other tools, it also explains coincidental similarity, which can occur due to compilation requirements, legitimate code reuse, intuitive implementation of an algorithm, and/or implementations suggested by instructors [49]. This can be valuable since in practice, code similarity is unduly emphasized as a sign of academic misconduct [60]. The assessment submission system removes long common code segments from the comparison process, ensuring that the remaining legitimately common code segments are shorter than inappropriately copied code segments. Therefore, coincidental similarity will generally not be a valid explanation for academic misconduct, as the code segments are different in terms of size. The longer the reported segments are, the less likely it is that their similarity is coincidental.

At first glance, our assessment submission system is similar to the tool of Le et al. [34], as both report code similarity before the submission due date; but our system has at least five further benefits. First, while Le et al. anonymize students' programs before displaying them to other students, we believe that some programs might remain recognizable by their programming style, and might therefore betray student privacy. To avoid this issue, our system illustrates similarities not by displaying other students' programs but by disguising the submitter's own program. Second, as Le et al. show anonymized students' programs, some students might be able to see better solutions and use them for their own advantage. Again, this is solved in our system by disguising the submitter's own program instead of showing them another student's work. Third, our system explains code similarity in natural language, as students are often unable to see the underlying similarities in code segments that are different on the surface. Fourth, our system informs students about how to proceed, while providing richer knowledge about task constraints, concepts, and mistakes. Last, our system immediately warns students when they are potentially committing plagiarism or collusion, as the feedback is given right after the submission, whereas the system of Le et al. gives feedback at preset intervals.

Currently, the assessment submission system accepts submissions written in Java or Python, the two most common introductory programming languages in Australasian and UK institutions [50]. Each submission can be a single program or a zip file containing multiple program files. In the latter case, the program files will be concatenated to one large file before being processed, a technique adapted from an existing similarity detector [26]. The report itself is written in English or Indonesian.

We acknowledge that educating students about programming plagiarism and collusion might lead some students to try to trick the detection of such misconduct. Our system attempts to mitigate this issue by showing no information about other students' details and programs, so colluding students cannot be sure that their collusion has been detected even if they are both alerted to similarities. Further, the similarity detection is intentionally designed to be less precise than that of common similarity detectors for detecting plagiarism and collusion (e.g., JPlag [42] or MOSS [46]), which can still be used following the submission deadline. In addition, based on our observations of the similarity detectors listed in a recent literature review [29], differences introduced to disguise the copying, and reported by our system, are often overlooked by existing similarity detectors as they generalize aspects of the code prior to comparison.

Figure 1 shows that our system works in six consecutive steps for each student submission:

(1) A student submits a program to the assessment submission system.
(2) The program is then converted to two intermediate representations: syntax index and token string. The syntax index, which is used to efficiently detect similar programs, is generated via the indexing step of our similarity detector [24]. First, the program's syntax tree is constructed and all directly-connected sets of three adjacent internal nodes are extracted. Occurrence frequencies of those are counted and then combined into a single index. The token string is used to exclude highly common code from being reported as similar and to highlight shared code segments. It is constructed by simply tokenizing the program. Both conversions are conducted with the help of ANTLR [41] and their results are stored in the database along with the original program.
(3) The submitter's program and the most recent versions of all other students' programs are sent to the similarity feedback generator. While the generator is detecting similarities, segments of code of substantial length that are common to many submissions are removed from the list of matched tokens.

(4) If the submitter's program is similar to the most recent submissions of one or more other students, the generator generates similarity reports for all involved students, personalized to their own programs. This is either to provide other students with real examples of coincidental similarity or to warn other students if a large portion of similar code is found. If there is no similarity, the system generates a report that simulates the similarity (a simulation report), but only for the submitter. This is to ensure that students with unique programs are not deprived of the educative aspect of the system.
(5) The link to the similarity report or the simulation report is emailed to the submitter. In the case of a similarity report, the submitter is encouraged to revise and resubmit the program and explain why the similarity occurred. Resubmission is allowed to provide another chance for students who are potentially committing academic misconduct. Being perceived as academic integrity violators may severely affect students' mental state. Based on the experience of one of the authors as a programming instructor, some students drastically alter their behavior when given another chance, validating the educative purpose of the system. The resubmission is really required only for students who are potentially committing academic misconduct, but it is generalized to all students with similarity reports due to the difficulty of differentiating coincidental and non-coincidental similarity. The requirement to explain is intended mainly to deter students from engaging in academic misconduct, as this might lead to difficulties in explaining the similarity. It is also useful for students' own reflection. Both earlier submissions and student responses regarding the reported similarities are available for the instructors as an additional consideration while detecting academic misconduct.
(6) Other involved students who are sent similarity reports as a result of this submission are encouraged to revise and resubmit while explaining the similarity. However, those who have already been sent a report for this assessment task and have not responded to it will not be further notified, as they might find it distracting to receive multiple similarity reports, one each time a new submission shares the same similarity.

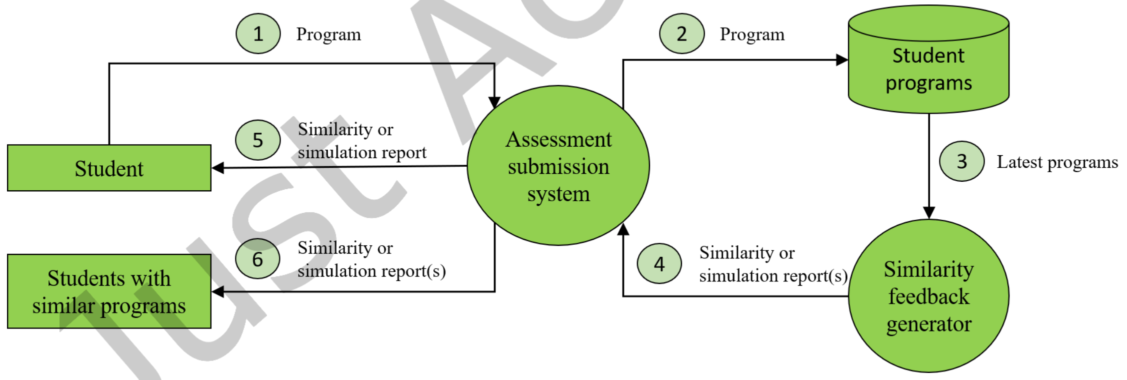

1
Program
2
Program
Student programs
Student
5
Similarity or simulation report
Assessment submission system
3
Latest programs
6
Similarity or simulation report(s)
4
Similarity or simulation report(s)
Students with similar programs
Similarity feedback generator

Fig. 1. System overview with six consecutive steps per submission

The similarity feedback generator consists of four components, as shown in Figure 2. The similarity detector identifies similar programs while the long common code remover prevents long common code segments from being reported as similar. If the submitted program is similar to at least one other program and the shared code segments are not common and long, similarity reports will be generated for all involved students. Otherwise, a simulation report will be generated solely for the submitter.

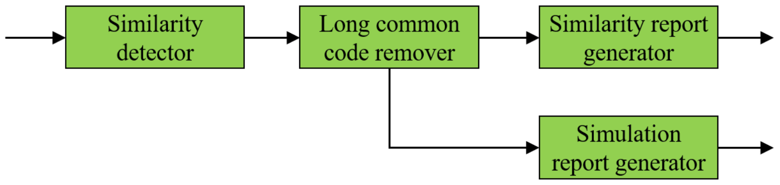


Fig. 2. Overview of similarity feedback generator

3.1 Similarity Detector

Our similarity detector needs to be time-efficient as each program will be compared on submission to the most recent programs from all other students. Further, it needs to be less precise than common similarity detectors for identifying plagiarism and collusion (e.g., JPlag [42] or MOSS [46]) so that students cannot easily learn how to evade suspicion of plagiarism and collusion. Our detector uses a technique from the field of information retrieval, applied to syntax trees [24]. The information retrieval technique, cosine similarity [8], is known for its time efficiency, partly because it uses the indexes as the basis for comparison; and the comparison is typically linear in time because it compares just the submission in question with all other submissions, rather than comparing all submissions pairwise. The submitted program will be considered similar to another program if their average similarity is greater than or equal to both 75% and the average similarity degree of all program pairs. This threshold was shown in our study to be effective, but is easily modified when using the system. The use of syntax trees can increase recall (the proportion of retrieved copied programs to all copied programs), although at a cost of reduced precision. Both of these benefits have been empirically confirmed [24] in comparison to the technique used by JPlag [42], a common similarity detector.

3.2 Long Common Code Remover

This component prevents long common code segments from being reported as similar. It is adapted from a module to semi-automatically remove common segments [25]. The component works in three stages. First, common code segments are selected based on the programs' token strings once identifiers, constants, and some primitive data types have been generalized to mitigate the impact of superficial variation.

Next, common segment candidates are formed by concatenating sets of 10 to 30 adjacent tokens, but only those that begin with an identifier or keyword, start at the beginning of a line, and end at the end of a line. The settings are explained in an earlier publication [25].

Finally, common segments, identified as those that occur in at least 75% of student programs, are excluded from comparison by explicitly marking them as mismatches. The threshold needs to be fairly high because the removal of common code segments takes place entirely without human validation, so a strict constraint is necessary both to reduce false results and to expedite the removal process [25]. Our manual observation confirms that the threshold is suitable for our data sets.

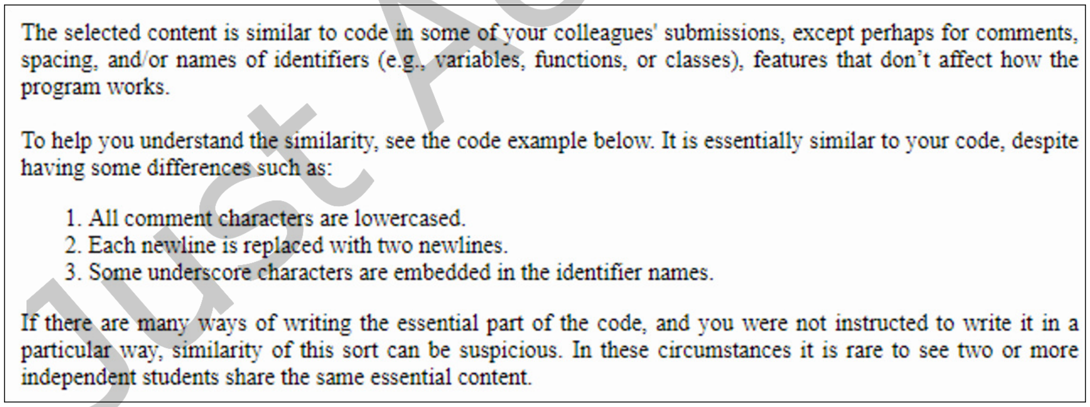
3.3 Similarity Report Generator

If the submitted program is unexpectedly similar to other students' latest programs and there is at least one similar segment that was not removed by the long common code remover, a similarity report is generated in five steps. First, the submitted program is paired to the most recent submission of every other student. For each program pair, the matched segments are detected based on their token strings from the long common code remover. The similarity algorithm uses running Karp-Rabin greedy string tiling (RKRGST) [58], selected for its ability to prioritize long matches (which are unlikely to arise by coincidence) and deal with code segment relocation in a reasonable amount of time [26]. The algorithm's minimum matching length is set to cover approximately two program statements: 20 tokens for Java and 10 for Python [25]. All adjacent matches will be merged.

Second, for each matched segment, the difference between programs is classified as one of four disguise types: level 0, level 1, level 2, and level 3 or higher. These levels are derived from Faidhi and Robinson's taxonomy of code disguises [9]. Level 0 similarity, which occurs when the segments are identical at surface level, is detected by comparing the original forms of the segments. Level 1 similarity, which occurs when the segments differ only in comments and white space, is detected by comparing the segments at token level but without any tokens generalized. Level 2 similarity occurs when the segments also differ in terms of identifier names, and is detected by comparing the segments' token strings with all identifiers generalized. Level 3 or higher similarity occurs when the segments contain other disguises in addition to lower level disguises. However, since we want the system to be somewhat lacking in precision, it only reports modifications to constants and data types. This detection works by comparing the segments' token strings after generalizing constants and some data types.

At the third step, matched segments from all program pairs are mapped to the submitted program. All overlapping segments are merged, with their similarity level assigned to the lowest level of the involved segments. For example, if there are three overlapping segments with level 2, level 1, and level 2 similarity, the merged segment will have level 1 similarity. This is to ensure that if there are any segments with obvious similarity, they are still reported as they are.

Fourth, each matched segment mapped to the submitted program is associated with an explanation about the similarity (see Figure 3 for an example). The explanation starts by stating that the segment is also found in some other students' submissions with their differences generally described based on the similarity type. An illustrating code example is provided as a case study, to encourage deeper understanding. The example is generated by disguising the segment of the submitter's own code based on the similarity type. For example, if the segment's similarity type is level 2, the disguises will be about identifier renaming, comment modification, and/or white space modification. The disguised segment is placed below the explanation text. The explanation ends with a statement about how the similarity can be suspicious.


The selected content is similar to code in some of your colleagues' submissions, except perhaps for comments, spacing, and/or names of identifiers (e.g., variables, functions, or classes), features that don't affect how the program works.

To help you understand the similarity, see the code example below. It is essentially similar to your code, despite having some differences such as:

1. All comment characters are lowercased.
2. Each newline is replaced with two newlines.
3. Some underscore characters are embedded in the identifier names.

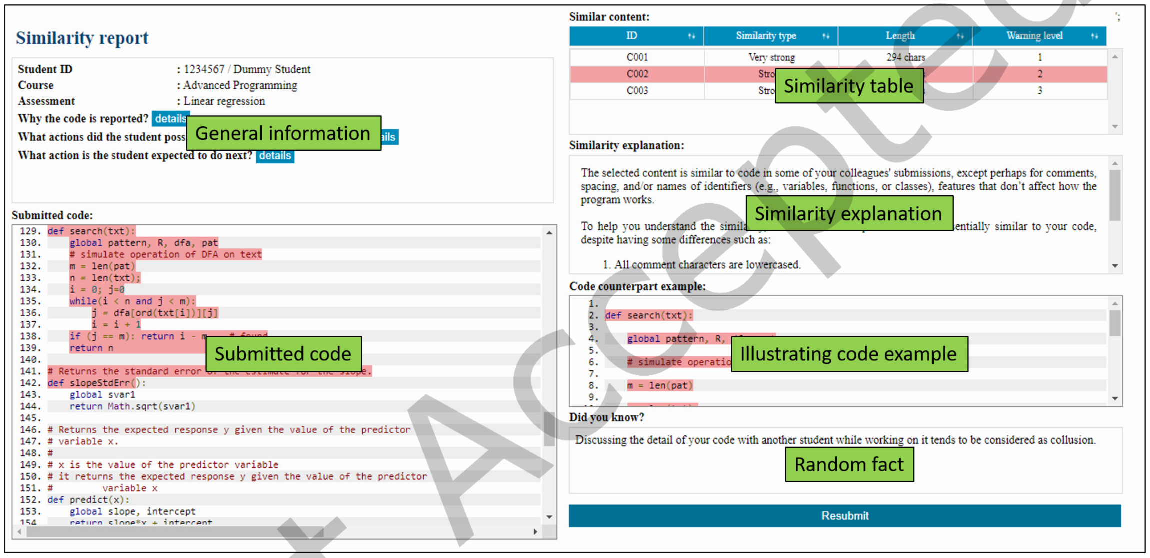
If there are many ways of writing the essential part of the code, and you were not instructed to write it in a particular way, similarity of this sort can be suspicious. In these circumstances it is rare to see two or more independent students share the same essential content.

Fig. 3. Example of similarity explanation that will be placed right above the illustrating code example

For the disguising process, we adapt CSD, a disguising tool to educate students about code similarity [23]. It covers five code components: comments, white space, identifiers, data types, and constants. The first two generate code with level 1 similarity; the third is for level 2; and the rest are for level 3 or higher. Level 0 similarity does not need any disguises as it is a verbatim copy. Students are expected to easily understand the code example since the disguises are natural and reasonably common among student programs. Out of sixty disguises, five are not used in this process. Three disguises that remove some comments can be replaced with their counterparts that remove all comments. A disguise that replaces spaces with tabs results in changes that are unlikely to be visible to students. A disguise that reformats the code indentation is time-inefficient.

The disguises to be applied are selected by listing modifiable code components for the similarity level (e.g., identifiers, comments, and white space for level 2). For each such component, a disguise will be randomly selected from the relevant disguises that can superficially change the given code segment, listed by recognizing modifiable code content [23].

Finally, the submitted program, the matched segments, the explanation, and the illustrating examples are embedded in an HTML page that forms the similarity report. The report's layout is a modified version of the side-by-side comparison view of STRANGE [26], and is illustrated in Figure 4. The report has six panels containing general information, the submitted code, a similarity table, a list of similarity explanations, a list of illustrating code examples, and a random fact.


Similarity report

Student ID	: 1234567 / Dummy Student
Course	: Advanced Programming
Assessment	: Linear regression

Why the code is reported? details
What actions did the student poss ils
What action is the student expected to do next? details

Submitted code:

```
def search(txt):
    global pattern, R, dfa, pat
    # simulate operation of DFA on text
    m = len(pat)
    n = len(txt);
    i = 0; j=0
    while(i < n and j < m):
        j = dfa[ord(txt[i])][j]
        i = i + 1
    if (j == m): return i - m
    return n

# Returns the standard error
def slopeStdErr():
    global svar1
    return Math.sqrt(svar1)

# Returns the expected response y given the value of the predictor
# variable x.
#
# x is the value of the predictor variable
# it returns the expected response y given the value of the predictor
#         variable x
def predict(x):
    global slope, intercept
```

Similar content:

ID	Similarity type	Length	Warning level
C001	Very strong	294 chars	1
C002	Stro		2
C003	Stro		3

Similarity explanation:

The selected content is similar to code in some of your colleagues' submissions, except perhaps for comments, spacing, and/or names of identifiers (e.g., variables, functions, or classes), features that don't affect how the program works.

To help you understand the simila entially similar to your code, despite having some differences such as:

1. All comment characters are lowercased.

Code counterpart example:

```

def search(txt):

    global pattern, R,

    # simulate operatio

    m = len(pat)

```

Did you know?

Discussing the detail of your code with another student while working on it tends to be considered as collusion.

Fig. 4. Layout of similarity report where the matched segments are highlighted and observable. The figure provides a brief overview of the report without expecting the text to be readable. The labels in boxes are added for clarity and the displayed code is taken from Sedgewick and Wayne [47].

The general information panel contains submission metadata (student ID, course, and assessment information), along with some general knowledge about programming plagiarism and collusion. Specifically, the knowledge covers why the report is generated (since the program is unexpectedly similar to some other students' programs), what actions the student might have taken that would lead to this similarity (listing possible reasons for both non-coincidental similarity [48] and coincidental similarity), and what action the student is expected to do next (revise and resubmit while explaining the similarity). Each item of information can be accessed by clicking its corresponding 'details' button.

The submitted code is displayed with Google Prettify[1] to enhance its readability. Further, similar code segments are highlighted in red to draw the student's attention. Clicking a segment will highlight its corresponding entry in the similarity table, update the explanation, and update the illustrating code example.

[1]https://github.com/google/code-prettify

The similarity table panel lists any similar code segments found in the submitted code. Each segment is represented by four information components. The segment ID facilitates convenient referencing in any inquiries between the student and the instructor, as the report is accessible to both. The similarity type measures how similar the segment is to other students' segments based on Faidhi and Robinson's disguise taxonomy [9]. Since the levels are not self-explanatory for students, they are changed to 'very strong' for level 0, 'fairly strong' for level 1, 'strong' for level 2, and 'moderate' for level 3 or higher. Segment length captures how many characters are involved in the segment; this is generally inversely related to the likelihood of coincidental similarity. The length is given in characters, not tokens, since students are not familiar with the latter term. Warning level represents the segment's priority for manual investigation. Longer segments with more obvious similarity (i.e., lower disguise level) are more likely to be prioritized. The level is generated by ranking all the segments based on the formula $(6 - level) * length$, where level is the segment's disguise level, six is the maximum disguise level, and length is the segment's length.

The remaining panels are fairly self-explanatory. The similarity explanation panel displays relevant explanation about the selected code segment, as illustrated in Figure 3. The illustrating code example panel shows the disguised form of the selected segment. The random fact panel randomly displays one fact taken from the general information panel.

Upon investigating the similarity report, the student is expected to revise and resubmit their program while providing their own explanation of the similarity.

The generated similarity report can contain both coincidental and non-coincidental similarity, and we leave students to justify which is which, for two reasons. First, the two types cannot be differentiated simply by segment length since they depend heavily on the design and context of the assessment task [49]. Second, non-coincidental similarity needs to be manually validated prior to being reported as it can be used as evidence for raising suspicion (burden of proof [21]).

The similarity report can provide four types of knowledge from Narciss' feedback content categories [37]. Knowledge about concepts can be found in the general information panel, in the random fact panel, and in observation of the reported code segments with their illustrating code examples. Knowledge about task constraints and knowledge about mistakes implicitly occur if many of the reported segments are unusually long with obvious similarity. Knowledge about how to proceed is explicitly mentioned in the general information panel, requiring the student to resubmit the program while explaining the reasons for the reported similarity.

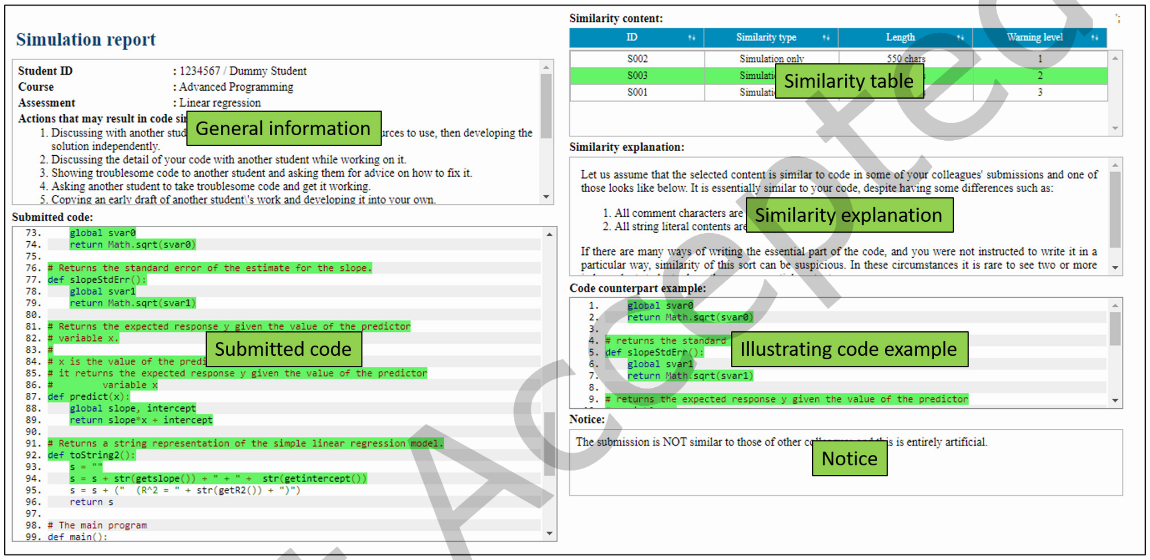
3.4 Simulation Report Generator

If a submitted program is not found to be unduly similar to any other student's program, a simulation report will be generated in four steps. First, the submitted program's 'matched' segments will be randomly selected from the token string previously generated for the long common code remover. For the purpose of readability, segments will be constrained to consist of one or more whole lines. Further, to make each segment comparable to those in the similarity report, its length is no less than RKRGST's minimum matching length (20 tokens for Java and 10 for Python, see subsection 3.3). If no code segments fulfill both criteria, the first step is re-executed with a template program. In this manner, the simulation report will still be generated even for programs that only contain long common code and programs whose length is less than the minimum matching length. The student will be informed via code comments that the program is not theirs and is provided only for the purpose of the simulation.

Second, the submitted program's modifiable code components are listed and each is given a 50% chance of having an artificial difference applied to it. Each of the generated differences is then classified as one of the four similarity types described in subsection 3.3. The mechanism is somewhat similar to the second step of the similarity report generator.

The final two steps are similar to the fourth and the fifth steps of the similarity report generator. Each matched segment is provided with an explanation and an illustrating code example, and the segments are embedded in an HTML page along with the submitted program.

Figure 5 shows an example of the simulation report. Its layout is similar to that of the similarity report. It contains panels for general information, the submitted code, similarity table, similarity explanation, and illustrating code examples. However, there are several differences to ensure that students do not misinterpret it as a similarity report. First, the highlighting color is green instead of red, implying less urgency to observe. Second, all segments' similarity types are described as 'simulation only'. Third, each similarity explanation explicitly states that it is based on assumption, not on actual code similarity. Fourth, instead of showing a random fact, a notice explicitly states that this is just a simulation. Fifth, the general information panel only contains submission metadata and a list of actions that may result in code similarity.



Simulation report

Student ID	: 1234567 / Dummy Student
Course	: Advanced Programming
Assessment	: Linear regression

Actions that may result in code si

1. Discussing with another stud rces to use, then developing the solution independently.
2. Discussing the detail of your code with another student while working on it.
3. Showing troublesome code to another student and asking them for advice on how to fix it.
4. Asking another student to take troublesome code and get it working.
5. Copying an early draft of another student\'s work and developing it into your own.

Submitted code:

```
    global svar0
    return Math.sqrt(svar0)

# Returns the standard error of the estimate for the slope.
def slopeStdErr():
    global svar1
    return Math.sqrt(svar1)

# Returns the expected response y given the value of the predictor
# variable x.
#
# x is the value of the predi
# it returns the expected response y given the value of the predictor
#          variable x
def predict(x):
    global slope, intercept
    return slope*x + intercept

# Returns a string representation of the simple linear regression model.
def toString2():
    s = ""
    s = s + str(getslope()) + " + " +  str(getintercept())
    s = s + ("  (R^2 = " + str(getR2()) + ")")
    return s

# The main program
def main():
```

Similarity content:

ID	Similarity type	Length	Warning level
S002	Simulation only	550 chars	1
S003	Simulati	[illegible]	2
S001	Simulati	[illegible]	3

Similarity explanation:

Let us assume that the selected content is similar to code in some of your colleagues' submissions and one of those looks like below. It is essentially similar to your code, despite having some differences such as:

1. All comment characters are
2. All string literal contents are

If there are many ways of writing the essential part of the code, and you were not instructed to write it in a particular way, similarity of this sort can be suspicious. In these circumstances it is rare to see two or more

Code counterpart example:

```
    global svar0
    return Math.sqrt(svar0)

# returns the standard
def slopeStdErr():
    global svar1
    return Math.sqrt(svar1)

# returns the expected response y given the value of the predictor
```

Notice:

The submission is NOT similar to those of other c s is entirely artificial.

Fig. 5. Layout of simulation report with comparable information to that of similarity report. The figure provides a brief overview of the report with labels in boxes added for clarity; the text is not expected to be readable.

The simulation report only provides knowledge about concepts as no mistakes are detected, no task constraints are relevant, and no actions are required of the student after observing the report.

4 THE APPROACH

At the beginning of the course, the instructors explain their expectations regarding academic integrity. However, only general explanations are required as the assessment submission system will provide the details later. They should also inform the students that the assessment submission system will be used and it will report program similarities, with the likelihood of coincidental similarity inversely related to program length, since legitimately long common segments are automatically excluded.

For each assessment, students are only expected to respond to a similarity report once. It would not be appropriate to require students to keep resubmitting until no similarities are detected, for two reasons. First, the detection of common code segments is not perfectly accurate as it is fully automated. Second, some relatively short similar segments are coincidental but not common to multiple programs.

If the course issues many small assessment tasks rather than a few large ones, it can be daunting to respond to the similarity report once for each assessment task. Consequently, the requirement to respond can be made entirely voluntary – but any responses might still be considered when detecting plagiarism and collusion. Thus students will only respond to the similarity report if they submit the program early and/or they are at risk of being suspected.

Reporting program similarities while allowing resubmission might be seen as suggesting that plagiarism and collusion are somewhat acceptable. We try to mitigate this by informing students that the reported similarities are not final, and a stronger similarity detector will be used to check the final submissions for evidence of plagiarism and collusion. Programs that are deemed unique by our assessment submission system might be found similar to others when checked using the stronger detector.

Along with educating students about programming plagiarism and collusion, it is important to detect such misbehavior and apply the appropriate penalties. Otherwise, the knowledge becomes less relevant to students as they will not benefit from it during the course. Detecting programming plagiarism and collusion can be aided with a similarity detection tool. For assessments that allow many distinct solutions at algorithmic level, MOSS [46] and JPlag [42] are commonly used. Otherwise, detectors capturing finer variation (e.g., looping variance) are preferred [28], like Sherlock's detection modes without program tokenization [21] and STRANGE [26]. Instructors can also use earlier submissions, along with student responses regarding the reported similarities, as additional considerations when checking for academic misconduct.

Our approach motivates students who are potentially committing plagiarism or collusion to re-attempt the assessment with their own work. We believe that they are unlikely at this stage to base their work on another student's solution as they know that a stronger similarity detector will be used when checking for plagiarism and collusion. Of course for this particular assessment task they have already seen another student's work, and that might influence their work; but we hope and expect that, at two assessment tasks a week, they will soon acquire the habit of completing the tasks without inappropriate assistance.

There are approaches that allow students to plagiarize in formative assessment, using the plagiarism as an early opportunity to educate the students about academic integrity [20, 34], and our own approach shares some features with such approaches. While we do not encourage students to plagiarize or collude, we acknowledge that some will do so as they struggle to understand what is acceptable and what is not, and our goal is to help them attain that understanding. Of course some students might use this early feedback to see what misconduct they can get away with; but because we use a stronger similarity detector on their final submissions, this is likely to lead to disappointment and disciplinary consequences.

5 EVALUATION

Our system and intervention were evaluated by two quasi-experiments conducted over two academic semesters; the control group (87 students) is from the second semester of 2019 while the intervention group is from the second semester of 2020 (76 students). As with many quasi-experiments, we assume that the control and intervention groups have students with comparable academic skill; as explained in the next paragraphs, this is because in one case they are enrolled in different offerings of the same course and in the other because they are the same students in different courses. None of the instructors are part of the research team, and all evaluation has been granted the appropriate ethics approval.

In the *course-focused* experiment, the control and intervention groups are from consecutive offerings of the same course, introductory programming. This is a compulsory course for first-year information systems undergraduates. Each assessment task is to be completed in Java and then translated to Python; both languages are heavily used in the information systems major and the instructors believe that this mechanism can help students to learn both at once. While the choice to translate the solutions to another programming language is clearly subject to discussion, no further explanation will be given here as it is not within the scope of this study. The control group has 26 students while the intervention group has 33 students. Assessments of the control group are comparable to those of the intervention group since they are given in the same course by the same lead instructor. We have explicitly checked the assessment specifications and found only minor changes between offerings.

Each introductory programming assessment has three programming tasks: an easy task, to implement the new syntax constructs covered in the lecture; a medium task, requiring students to think about how to solve the problem in addition to implementing the new syntax constructs; and a challenging task, similar to but more difficult than the medium task. To illustrate this, consider an assessment about single looping. The easy task is "given a number *N*, print 1 to *N*"; the medium task is "given a number *N*, print the first *N* fibonacci numbers without using an array"; and the challenging task is "given a number *N*, create a 2D right triangle of asterisks, of size *N*, using only one loop".

Each task usually only allows a few solutions at algorithmic level. However, this does not mean that all student submissions must be similar. Our approach still considers syntactic variation (e.g., looping forms) and small details (e.g., the use of parentheses). Further, the approach automatically excludes long similar segments that are common among student submissions such as "public static void main (String [] args)". Moreover, per submission, solutions for all three tasks are concatenated prior to comparison.

In the *student-focused* experiment, the control and intervention groups consist of essentially the same students in consecutive courses. The control group consists of 61 students enrolled in a different introductory programming course, a compulsory first-year course for information technology undergraduates. The assessment design is similar to that of the course-focused experiment except that it only uses Python as the programming language. The intervention group consists of 43 students enrolled in advanced algorithms and data structures, a compulsory second-year course that uses Java as its programming language. Most of the students in this course had been in the control group in the preceding course. Each assessment is about implementing a data structure (e.g., stack or linked list) to solve a task (e.g., simple arithmetic operation for stack) based on a particular template.

In this experiment, the assessments of the intervention group clearly differ from those of the control group; in the control course, the assessment tasks are expected to result in fairly distinct solutions [27]. On the other hand, assessment tasks in the intervention course rely on template code provided by the lecturer, have more constraints, and result in longer programs, so similarities among programs are more likely. However, these differences do not harm the validity of the experiment since they disfavor our approach; in the intervention group it is less likely that we will be able to detect the sort of similarity that suggests academic dishonesty, so our approach needs to be particularly effective to have a perceptible impact.

All courses in the evaluation issue two assessments each week. The lab assessment task is to be completed during the two-hour lab session, and the homework assessment task is to be completed before the following lab session. In just one of the courses, advanced algorithms and data structures, the homework assessment is similar to the lab counterpart, and students are expected to carefully check their lab solutions and add any features that they had failed to complete in the lab.

At the beginning of each course offering, control and intervention, the instructors verbally informed students about acceptable practices in their courses, including penalties for those who breach the rules. This took around 15 to 20 minutes of the first laboratory session. The information was occasionally repeated in later sessions to remind the students. Such repetition occurred more frequently with the control groups as the approach presented in this paper was not available to help guide the students.

During the intervention period, our approach was used for all assessments. Because there were so many assessment tasks, student responses to any reported similarities were set to be voluntary, and were considered only when detecting programming plagiarism or collusion. The detection was performed with STRANGE [26], which focuses on finer granularity (e.g., loop variance) than a common similarity detection tool such as JPlag [42]. The finer-grained tool is more effective on assessments such as ours, which do not allow many possible solutions at algorithmic level [28]. Students whose programs seemed to be a result of misconduct would be awarded no marks for that assessment task. Further, they would be personally warned if the misconduct recurred. The instructors also used student responses regarding the similarity reports as an additional consideration. If the programs are similar but the reasons are logical, the students are not suspected of academic misconduct. However, the instructors chose not to consider earlier submissions for each assessment item since these submissions were perceived as ‘work in progressâĂŹ.

It is worth noting that the intervention period was during the covid-19 pandemic and all class sessions were conducted online. While department policy, teaching designs, and assessment designs were still comparable to those before the pandemic, interactions between instructors and students became more challenging. This might have reduced students’ capability to understand the course materials, possibly increasing their temptation to cheat [19]. However, this increases the validity of our findings rather than reducing it, as our approach needed to be particularly effective in order to bring about a measurable decrease in detected similarities when the temptation to cheat might be higher.

While there is an increasing number of discussions about academic integrity during the pandemic, it is unlikely to favor our approach during the intervention period. Our instructors believe that they gave comparable information about academic integrity at the beginning of the courses. Moreover, most of the discussions are among instructors, not students. In addition, the increase in discussion of academic misconduct is still small compared to the increase in discussions of general issues such as public health and the economy. Nevertheless, we are still open to two small possibilities. First, that the instructors unintentionally took more effort to warn students about academic integrity, making them more aware of programming plagiarism and collusion. Second, that a few students read relevant news and became more aware of the matter.

At the end of each of the four course offerings, students were invited to complete a short survey. In order to maintain the validity of the survey, students were not subsequently told which answers are correct; those in the student-focused experiment addressing RQ1 and RQ2 (subsections 5.1 and 5.2) would be invited to complete the survey once more, while others might inform students who had not taken the survey. Although we worked to make the survey questions as clear as possible, we acknowledge the possibility that some students might misinterpret them, resulting in low-quality answers.

To further understand student behavior while using the system, statistics of the generated reports are described. We also summarize the kinds of justification used to explain the reported similarities. While it would be interesting to conduct further analysis into how students used the system, we are unable to do that since our ethics approval does not permit analysis that involves student identification and behavior tracking.

5.1 RQ1: Improved Student Awareness of Programming Plagiarism and Collusion

Student awareness of programming plagiarism and collusion is measured via a voluntary survey given at the end of each course offering in our quasi-experiments.

The survey adapts questions from Simon et al. [48] asking whether students believe particular scenarios are academically acceptable. Three of the 14 original questions were not used, as our approach does not cover copying from online sources and the courses do not specifically use message boards in their learning management system. The remaining questions have been slightly modified to focus on plagiarism and collusion among students. Each question has three answer options: ‘yes’, ‘no’, and ‘do not know’. These questions do not have universally correct answers, as instructors are known to differ on what they find acceptable in their own courses [52].

Table 1 shows the questions, along with the answers deemed correct by the instructors in these courses. The correct responses were determined via informal discussions between the instructors and one of the authors before the semesters started (mid-2019 and mid-2020). The instructors were given the survey questions and the correct responses were discussed until consensus was reached.

Table 1. Survey questions about general knowledge of programming plagiarism and collusion, with the answers deemed correct by the instructors of these courses

ID	Is this academically acceptable?	Answer
GQ01	Purchasing code written by other students to incorporate into your own work	No
GQ02	Paying another student to write the code and submitting it as your own work	No
GQ03	Basing an assessment largely on work that you wrote and submitted for a previous course, without acknowledging this	No
GQ04	Incorporating the work of another student without their permission	No
GQ05	Copying another student's code and changing it so that it looks quite different	No
GQ06	Copying an early draft of another student's work and developing it into your own	No
GQ07	Discussing with another student how to approach a task and what resources to use, then developing the solution independently	Yes
GQ08	Discussing the detail of your code with another student while working on it	Yes
GQ09	Showing troublesome code to another student and asking them for advice on how to fix it	Yes
GQ10	Asking another student to take troublesome code and get it working	No
GQ11	After completing an assessment, adding features that you noticed when looking at another student's work	Yes

Table 2. Statistics of the survey responses

Experiment	Group	Responses	Total students
Course-focused	Control	25	26
	Intervention	27	33
Student-focused	Control	47	61
	Intervention	39	43

Table 2 summarizes the numbers of responses for both the course-focused and student-focused experiments. Although the survey was voluntary, the response rate is quite high, perhaps because students were interested in the survey's subject matter. Further, the survey was directly distributed by one of the authors (physically for 2019 and virtually for 2020), who explained to the students that participation might positively affect their future approach to learning.

Participants in the student-focused experiment were invited to complete the survey twice, once at the end of each course, and it is possible that their second response was influenced by their first. We took three steps to try to mitigate this possibility: we did not inform the control group that they would be invited to take the survey again a year later; we did not reveal the correct answers; and in the second course we did not inform students that the survey was the one they had taken a year earlier.

In the analysis, the average proportion of correct answers of the intervention group is compared with that of the control group for the full set of questions and for each individual question. Differences between the groups are measured with a two-tailed unpaired t-test with 95% confidence rate. The t-tests were performed under several assumptions. First, the responses form a continuous scale for the full set of questions (proportion of correct responses) and an ordinal scale for each individual question (correct or incorrect). Second, the responses represent student awareness of programming plagiarism and collusion with and without our approach. Third, the

Table 3. Statistically significant changes in proportion of correct answers about general knowledge

Experiment	Question	Control	Intervention	Variance	P-value
Course-focused	All	31%	64%	Equal	0.01
	GQ05	40%	81%	Equal	< 0.01
	GQ10	16%	70%	Equal	< 0.001
	GQ11	36%	77%	Equal	< 0.01
Student-focused	All	29%	68%	Equal	< 0.001
	GQ03	6%	23%	Unequal	0.03
	GQ04	77%	92%	Unequal	0.04
	GQ05	43%	79%	Equal	< 0.001
	GQ07	70%	97%	Unequal	< 0.001
	GQ09	76%	97%	Unequal	< 0.001
	GQ10	15%	82%	Equal	< 0.001

responses are normally distributed. Fourth, the sample size is reasonable given that it involves 72 responses of the control groups and 66 responses of the intervention groups. Fifth, the variance of the responses is considered while measuring the test.

A t-test is preferred to other statistical tests since it is applicable to both analyses: the overall improvement with continuous-scale responses and individual question improvement with ordinal-scale responses.

For the course-focused experiment, Table 3 shows that overall, students in the intervention group have a better general knowledge of programming plagiarism and collusion, with a 33% higher average correct response rate. However, only three individual questions show statistically significant changes: GQ10 with 54% improvement, GQ05 with 41%, and GQ11 with 41%.

Many students in the intervention group view it as academic misconduct to ask a colleague to fix some errors (GQ10). They are made aware that when part of the program is written by a colleague, this is collusion, and results in similarity between their program and that of the colleague because, when fixing the errors, the colleague typically re-implements their own solution. The correct response rate for the control group is particularly low since some students thought it acceptable to have other students fix their code directly.

That knowledge also explains the significant improvements in GQ05 and GQ11. GQ05 is about copying a colleague's program and modifying it so that it looks different from the original. The intervention group see this scenario as academic misconduct as the program is based on another student's program. Changing the 'look' does not change the ownership of the program as most of the modifications are not related to program flow. GQ11 is about implementing additional features noticed when looking at another colleague's work. Many students in the intervention group agree with the instructors that this is not academic misconduct since the features are implemented individually.

For the student-focused experiment, when exposed to our approach, students are more aware of general knowledge by 39%, with six questions showing statistically significant changes. As with the course-focused experiment, GQ05 and GQ10 show the greatest improvement. In the intervention group (their second course in the experiment) students are particularly aware that asking a colleague to fix errors and submitting a modified version of a colleague's program are both academic misconduct.

GQ03, GQ04, GQ07, and GQ09 show significant increases just in the student-focused experiment. It is possible that the intervention group had more opportunity to learn since the assessments entailed longer and more complex solutions. Further, the students had more programming experience. GQ03 is about basing an assessment on work submitted for a previous course without acknowledging the reuse. The scenario is related to self-plagiarism, a topic that is widely misunderstood [7]. This is why the correct response rate is particularly low, even for the intervention group. The improvement might be due to student awareness of code similarity as a primary source of evidence for plagiarism and collusion. If the earlier submission and the current one are compared, they will show undue similarity. GQ04 is about incorporating a colleague's work without permission. The student's work is not entirely written by themselves and it is likely that the incorporated work will remain similar to that of the colleague.

GQ07 and GQ09 are considered as acceptable scenarios since the program is written solely by the student. GQ07 is about discussing with another student how to approach a task and what resources to use, then developing the solution independently. Although the students use the same approach, their implementations are likely to differ. GQ09 is about showing troublesome code to another student and asking them for advice on how to fix it. The given advice still needs to be implemented in the program by its writer.

GQ11, which is about implementing additional features noticed when looking at another student's work, shows no significant difference in the student-focused experiment. While the instructors see this behavior as acceptable, students remain uncertain, perhaps because they think that incorporating the features is presenting the other student's work as their own. Further, in data structures assessments, some features only have one obvious implementation, and knowing the features means knowing how to implement them.

5.2 RQ2: Improved Student Awareness of Common yet Futile Program Disguises

Along with the questions discussed in the previous section, the survey included questions about program similarities and attempts to disguise them. These questions can be seen as measuring student awareness of common yet futile program disguises.

The disguises in question are those in the early levels of Faidhi and Robinson's taxonomy of code disguises [9]: verbatim copy (level 0), comment and white space modification (level 1), identifier renaming (level 2), and code segment relocation (level 3). These disguises are ignored by many code similarity detectors, which generalize white space and identifier names before comparing programs, and which can often accommodate code relocation. Each level of disguise is mapped to three questions, covering the first three levels of Bloom's revised taxonomy [3]: remember, understand, and apply. Each question asks students whether they agree or disagree with a statement, or do not know.

The questions are shown in Table 4, along with the answers deemed correct for these courses by the researchers. The correct responses were confirmed by the instructors via informal discussions before the semesters started (mid 2019 and mid 2020). CQ01–CQ04 are remember-type questions, CQ05–CQ08 are understand-type questions, and CQ09–CQ12 are apply-type questions. In each category, the questions are ordered by the disguise level they cover (e.g., CQ01 for level 0 and CQ02 for level 1).

Readers should remain aware that some survey questions might have different expected responses in different courses. For example, CQ12 is agreed by our instructors since most of their assessments are simple, expecting the solutions to be quite similar except perhaps for the order of program blocks. The responses were analyzed in the same manner as those for RQ1. We had initially planned to measure the impact based on the learning levels; however, that analysis showed no interesting patterns and is therefore not reported here.

Table 5 shows that in the course-focused experiment, student awareness of common yet futile program disguises is improved by 41%, with most of the questions showing a statistically significant improvement. CQ02, CQ03 and CQ10 are the questions with the greatest improvement. Students in the intervention group are particularly aware that both changing comments (CQ02) and changing variable names (CQ03) are futile. Variation just in comments and variable names was often found among student programs, and thus was reported to the students. They are also aware that trivial tasks such as printing "Hello, world!" are expected to result in high yet acceptable

Table 4. Survey questions about knowledge of code similarity, with the answers that apply to the courses in the study

ID	Statement	Answer
CQ01	Copying source code from another student without modification will lead to strong suspicions of plagiarism or collusion.	Agree
CQ02	Source code comments can be changed, added, or removed without understanding how the program works. Hence, two similar programs with different comments will lead to suspicions of collusion.	Agree
CQ03	Variable names cannot be changed without understanding how they are used in the program, so unique names guarantee that the author is not involved in plagiarism or collusion.	Disagree
CQ4	The order of function/method declarations commonly affects how a program works, so two programs with different orders of declaration are clearly both original.	Disagree
CQ05	The highest level of content similarity occurs when two programs share the same syntactic program flow, code layout, and comments.	Agree
CQ06	Source code comments describe in human language what the program is doing, so in programs that do the same thing, the comments are expected to be very similar.	Disagree
CQ07	The names of functions/methods can be changed just as variable names can be changed, so these names should be considered in the same way when checking for plagiarism and collusion.	Agree
CQ08	Changing the order of function/method calls requires more programming knowledge than changing the order of function declarations, so a different order of function/method calls is less likely to suggest plagiarism or collusion.	Agree
CQ09	Students are required to use one of three algorithms in a programming assessment. Student A has known programming since high school and loves to use advanced programming techniques. Student B copies Student A's work and submits it as their own, thinking that most students' programs will be similar as there are only three algorithms to choose from. Student B will not be suspected of collusion.	Disagree
CQ10	After completing a simple *Hello World* assignment, a student tries to make the comments really distinctive because they are all that will make the program different from those of other students. The studentâĂŹs thinking is valid.	Disagree
CQ11	Two international students who speak different languages are taking an introductory programming course. In an assignment, their programs are identical except that the variable names are written in their respective mother language. These two will be suspected of plagiarism or collusion.	Agree
CQ12	A programming assignment involves calculating a course grade from the marks for quizzes, tests, and assignments; each score will be calculated separately from its respective sub-scores. Student A builds the program by calculating the quiz score, test score, and assignment score sequentially as instructed. Student BâĂŹs program is similar except that the quiz score is calculated last. These students will not be suspected of plagiarism or collusion.	Agree

Table 5. Statistically significant changes in proportion of correct answers about code similarity

Experiment	Question	Control	Intervention	Variance	P-value
Course-focused	All	31%	72%	Equal	< 0.001
	CQ02	32%	89%	Equal	< 0.001
	CQ03	16%	70%	Equal	< 0.001
	CQ04	8%	37%	Unequal	0.01
	CQ05	48%	89%	Unequal	< 0.01
	CQ06	4%	52%	Unequal	< 0.001
	CQ07	44%	74%	Equal	0.02
	CQ08	44%	74%	Equal	0.02
	CQ09	44%	89%	Unequal	< 0.001
	CQ10	12%	74%	Equal	< 0.001
	CQ11	20%	63%	Equal	< 0.01
Student-focused	All	34%	65%	Equal	< 0.001
	CQ01	85%	100%	Unequal	< 0.01
	CQ02	49%	74%	Equal	0.01
	CQ03	9%	69%	Unequal	< 0.001
	CQ04	17%	49%	Equal	< 0.01
	CQ05	45%	97%	Unequal	< 0.001
	CQ08	32%	62%	Equal	< 0.01
	CQ09	49%	82%	Equal	< 0.01
	CQ10	11%	64%	Unequal	< 0.001
	CQ11	21%	49%	Equal	< 0.01
	CQ12	21%	51%	Equal	< 0.01

similarity (CQ10). Further, some of the assessment tasks in this course have only have a few possible solutions due to their simplicity.

For the student-focused experiment, students in the intervention group are generally more aware of common yet futile program disguises, with an increase of 31%. As in the course-focused experiment, CQ03 and CQ10 show the greatest improvement. Students are made aware that changing variable names is pointless, and that it is acceptable to have similar programs for trivial assessment tasks. CQ05, which discusses the highest level of content similarity, is another question with large improvement, as some template code segments are provided for verbatim student use, and can be reported to students as coincidentally similar segments.

CQ01 and CQ12 are significant just in the student-focused experiment. The assessment tasks in the intervention course have longer solutions with more variability in the order of code segments than those in the course-focused experiment. Further, the tasks expect more method declarations. It is therefore less likely that two complete programs will be coincidentally verbatim copies (CQ01), and, in the data structure course (the one with the approach), semantically identical programs with different ordering of program blocks are more likely to be independent given that the blocks usually represent method declarations (CQ12).

In contrast, CQ06 and CQ07 show no significant changes in the student-focused experiment. CQ06 is about the fact that programs to do the same thing do not always have the same comments. Students in the intervention course are unsure about this, as some data structures assessments involve direct translation from algorithm instructions to program statements, which can result in similar comments. CQ07 is about the futility of changing the names of methods or functions. It shows no significant change as the student-focused control group was already knowledgeable about the matter (68% correct responses, compared with 44% in the control group of the course-focused experiment).

5.3 RQ3: Reduced Code Similarity Degree of Submitted Programs

To address our third research question we need to measure the level of code similarity among student programs. As there is no agreed measure of code similarity, we use a common classification proposed by Roy et al. [45], which introduces four types of similarity. Type I similarity means that the similar programs differ only in comments and white space. Type II similarity allows differences in identifier names, data types, and constants in addition to those of type I similarity. Type III similarity occurs when the programs share the same logical flow. Type IV similarity is shown when programs display the same functionality, and is not applicable in our study as the assessment tasks expect all students' solutions to have the same functionality. There is a clear overlap between this classification and that of Faidhi and Robinson [9] as used in the previous section, but it seems appropriate to use a taxonomy of code disguises when explicitly discussing code disguises, and a classification of code similarity types when discussing the observed levels of similarity among student programs.

The three similarity types are measured with the help of both JPlag [42] and STRANGE, a simpler similarity detector that uses the same technique [26]. JPlag is used to detect type III similarity; STRANGE is used, without generalizing program statements, to detect type II similarity; and its simplest version, which only tokenizes programs and removes comments and white space, is used to detect type I similarity. JPlag uses RKRGST as the similarity algorithm. Following our RKRGST setting in subsection 3.3, we set its minimum matching length to the approximate length of two program statements (20 for Java and 10 for Python) [25]. The similarity degree for each program pair is calculated with JPlag's average normalization [42], as in Equation 1. *A* and *B* are the involved programs; *matches(A,B)* is the total length of similar content; and *length(A)* and *length(B)* are the length of *A* and *B* respectively. Average normalization is preferred to the alternative, maximum normalization, due to its capability to consider all code differences when calculating the similarity degree [26].

$$sim(A,B) = \frac{2 \times matches(A,B)}{length(A) + length(B)} \tag{1}$$

For the purpose of comparison, each assessment task from the intervention group (which we shall call an intervention assessment) is paired with a comparable assessment from the control group (a control assessment). The comparable assessment should cover the same material for the course-focused experiment, or be drawn from the same course week for the student-focused experiment. Statistics of the assessment pairs can be seen in Table 6.

For the student-focused experiment, the intervention assessments differ in three ways from their control counterparts. First, the intervention assessments are all based on template code. Second, the intervention assessments are more strongly directed, as simple tasks in data structures do not allow much variation. Third, the intervention assessments use Java as the programming language while the control assessments use Python, which typically lends itself to shorter program statements. While the first difference is easily mitigated by removing the template code prior to comparison with an automated tool [25], the other two differences are not easy to resolve and thus remain. Consequently, the intervention assessments are expected to result in higher similarity by default, and therefore not to show the benefits of using our approach. In these circumstances, if code similarity is less in the intervention group than in the control group, the system would seem to be particularly effective.

Analysis is conducted separately for each type of similarity. For each experiment, we calculate the differences in similarity between the control assessments and the corresponding intervention assessments, as follows:

Table 6. Statistics of the assessment pairs

Metric	Course-focused	Student-focused
Assessment pairs		
Lab	8	12
Homework	8	12
Total	16	24
Control group submissions		
Lab	186	606
Homework	184	599
Total	370	1205
Intervention group submissions		
Lab	159	475
Homework	184	442
Total	343	917

(1) For each pair of assessment tasks, the average similarity degree for the control assessment is measured by comparing the most recent submissions in pairs at task level and averaging all of the reported similarity degrees. The same computation is then done for the intervention assessment. We cannot compare complete submissions as students in the control group submitted the programs as individual tasks. Merging the programs is not possible, as we have access only to the individual de-identified programs.
(2) The change in similarity is calculated by subtracting the intervention assessment's average similarity from that of the control assessment; the significance is determined via two-tailed unpaired t-test with 95% confidence rate.
(3) All assessment pairs showing significant reductions are counted.

We also compute an overall difference, by averaging the average similarity degree of all intervention assessments, and subtracting it from that of all control assessments. For this computation the significance is measured with a paired t-test as each intervention assessment has a control counterpart.

We also analyzed the data based on the assessment type (lab or homework). However, the findings are somewhat similar to those of the general analysis, and are therefore not reported here.

Reductions of type I and type II similarities are expected since they are reported to students by our assessment submission system. However, they still show how effective our approach is. They are also useful to enrich our findings by comparing reductions in each type of similarity. Type III similarity is not reported at all by our approach, so its reduction can indicate that students with our approach are less likely to engage in programming

Table 7. Summary for type I code similarity

Metric	Course-focused	Student-focused
Significant pairs with reduced similarity	16	17
Significant pairs with increased similarity	0	2
Insignificant pairs	0	5
Average similarity degree of control	10%	12%
Average similarity degree of intervention	3%	5%
Overall similarity degree reduction	7%	7%
P-value for overall reduction	< 0.001	< 0.01

plagiarism and collusion. Although it is unlikely, we acknowledge that any reduction in type III similarity might be an accidental consequence of students' deliberate reduction in type I and II similarities.

We are aware that similarities might occur for reasons other than plagiarism and collusion. To make the similarity reduction meaningful, the assessments should be comparable, or at least disfavor the intervention group (i.e., having higher similarity by default). Comparable assessments were applied in our course-focused experiment while assessments that disfavor the intervention group were applied in the student-focused experiment (see the assessment designs at the beginning of section 5).

While the number of identified cases of plagiarism or collusion would be another promising metric, we have not used it since the process of identifying plagiarism and collusion is different before and during the pandemic. Before the pandemic, the instructors used much supplementary information, gained from physical interactions in the classroom, that was not available during the pandemic. Further, we do not have access to the number of identified cases of plagiarism and collusion in the control groups as we did not ask the instructors to record it.

Finally, we, as the researchers, cannot check for plagiarism and collusion, because we are not involved with the students and we do not know which similarities are coincidental. We can check only for program similarity, which is what we have done.

5.3.1 Reduced Type I Code Similarity. Table 7 shows that in the course-focused experiment, type I similarity is low regardless of the intervention; many students were already aware that two programs are considered similar if they share the same surface representation. However, the intervention group showed a 7% reduction in surface similarity, and the change is statistically significant for all assessment pairs. We would like to conclude that students became more reluctant to engage in acts of plagiarism and collusion. However, as surface similarity can be modified without understanding the program, the finding might be a result of some students attempting to trick the similarity detector by copying code and introducing unique identifier names. Findings on other types of similarity are necessary to support our conclusion.

It is interesting that the reduction occurs even in the first lab assessment. Although students had not seen their own similarity reports at that time, during the course introduction they had been shown examples of reports and informed that their programs would be checked for similarity that might suggest plagiarism or collusion.

In the student-focused experiment, the intervention group shows a reduction in type I similarity, suggesting either that students were discouraged from plagiarizing or colluding or that they attempted to trick the similarity detector. However, not all differences are significant, and not all are reductions in similarity. As explained earlier, this is expected as the assessments of the intervention group are more strongly directed, resulting in higher intrinsic similarity.

Table 8. Summary for type II code similarity

Metric	Course-focused	Student-focused
Significant pairs with reduced similarity	16	23
Significant pairs with increased similarity	0	1
Insignificant pairs	0	0
Average similarity degree of control	35%	46%
Average similarity degree of intervention	15%	21%
Overall similarity degree reduction	20%	25%
P-value for overall reduction	< 0.001	< 0.001

Two student-focused assessment pairs showed a significant increase in similarity. As both of these pairs are from week 1 (lab and homework), it seems plausible that students in the intervention course became more cautious after observing their first-week similarity reports.

Although the reduction in similarity begins in the early weeks and we observe no relationship between the reduction and the assessment week, the intervention still serves a purpose in the remaining weeks, maintaining the students' perspective and behavior.

5.3.2 Reduced Type II Code Similarity. In the course-focused experiment, Table 8 shows that similarity is lower in the intervention group. The reduction is greater than that of type I similarity since the students were not previously aware of type II similarity, where programs can still be considered similar despite variation in identifier names, data types, and constants. In both control and intervention groups, the instructors did not explicitly inform students about this kind of similarity at the beginning of the courses since it is overly technical and detailed. The instructors stated only that not all programming aspects contribute to program similarity. In the student-focused experiment the findings are somewhat similar except for one assessment pair (the week 1 lab) where the intervention course shows a higher degree of similarity. Students in this second course were not fully aware of the nature of the intervention until they saw their first similarity report.

It is worth noting that on the week 1 homework, the intervention group shows an increase in type I similarity but a reduction in type II similarity. This does not mean that students can change the syntax of a program without affecting its superficial form. Rather, it shows the limitation of type I similarity detection in dealing with short similar code segments. The control assessment simply requires students to print static text; given surface variation in the string literals, many similar segments are too short to satisfy the predefined minimum matching length criteria (20 for Java and 10 for Python) and thus are considered as mismatches.

The fact that the intervention group shows a reduction in both type I and type II similarity suggests that the students are less likely to engage in plagiarism or collusion. However, observation of type III similarity is still required, as there might be students who try to escape the detection of academic misconduct by modifying syntactical features; for example, replacing a *while* loop with a *for* loop.

5.3.3 Reduced Type III Code Similarity. As shown in Table 9, most course-focused assessments show a reduction in type III similarity in the intervention group. Although the system does not specifically report such similarity, it appears that its use discourages students from plagiarizing or colluding. The 16% reduction is not as substantial as the 20% shown for type II since the assessment tasks permit less variation in their logical flow.

For the student-focused experiment, the intervention course still shows lower similarity, but its impact is less substantial and a few assessment pairs show a increase in similarity. Assessment solutions in the intervention group are expected to have more similar program flow as a particular structure needs to be followed, template

Table 9. Summary for type III code similarity

Metric	Course-focused	Student-focused
Significant pairs with reduced similarity	15	18
Significant pairs with increased similarity	0	5
Insignificant pairs	1	1
Average similarity degree of control	77%	81%
Average similarity degree of intervention	61%	74%
Overall similarity degree reduction	16%	7%
P-value for overall reduction	< 0.001	< 0.001

code is provided for the students, and some code segments can be readily and legitimately copied from external resources.

As the intervention group also has reduced type I and type II similarity for most assessments, it can be concluded that students in this course are less likely to engage in programming plagiarism and collusion. Students are made aware of the futility of copying and disguising programs, and this discourages them from doing so. While it remains possible that the reduction in similarity is the result of some students learning how to evade the detection of academic misconduct, this is quite unlikely: changing the logical flow of a copied program requires a level of programming knowledge that is probably rare among students who feel the need to copy one another's work. Further, our system does not report type III similarity to students, so they would not be aware that it was being detected.

5.4 Student Behavior When Using the System

This subsection describes student behavior while subject to our approach in both the course-focused and student-focused experiments. Specifically, it shows statistics of the generated reports and summarizes student justifications to the reported similarities.

Figure 6 shows the statistics of generated reports, grouped by experiment and assessment type. In general, most of the generated reports are similarity reports and only a small portion of them received student responses with justifications, except for the student-focused lab assessments. This general pattern is expected, as the criteria leading to the generation of a similarity report are easily met (the submitted program is unexpectedly similar to others), and student response and justification were not mandatory.

In our use of the system, students were not expected to provide justification of similarities for lab assessments since the assessments were to be completed in class in the short time span of two hours. However, in the course-focused lab assessments, a few students submitted their programs early and had time to respond to the similarity report. Our system records 22 responses for the whole semester.

Regarding the student-focused lab assessments, the number of responses to similarity reports is considerably higher since the lecturer explicitly encouraged students to justify the reported similarity as part of the description when submitting their subsequent homework assessments. This gave students time to examine the similarity report and provide the justification. Further, they could provide the justification without resubmitting the program, as submission of lab assessments takes place only during the class time. Future work may involve asking students to include a justification for their most recent similarity report when they submit the next assessment item.

Students were expected to respond to similarity reports on homework assessments, but as this was voluntary, the number of responses is still relatively low. Many students whose reports resulted from coincidental similarity

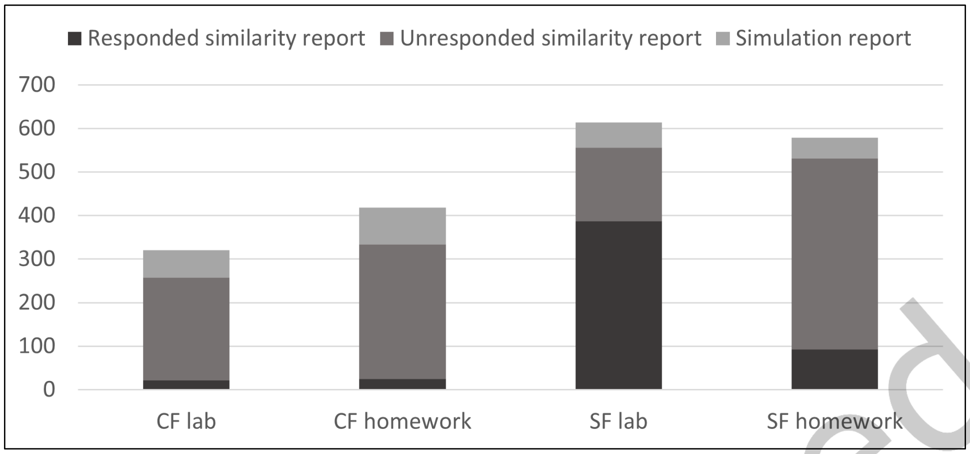


Fig. 6. Statistics of the generated reports by task type (lab or homework) and experiment (course-focused, CF, or student-focused, SF)

Table 10. Summary of student justifications

Category	Example
Coincidental similarity	The only way taught to do the task
	Adapted from accessible sources for completing assessments
	Simple and intuitive solution
	Asked the same instructor for hints
Academic misconduct	Discussed with some colleagues while completing the assessment
	Asked some colleagues for help
	Copied from the internet
	Copied from previous assessments
	Unaware of who copied the code
Ambiguous justification	Independently written
	Incorrect/incomplete solution
	Now changed to make the code more unique
	Unused code that had not been removed

were not interested in explaining this as the process would also involve a resubmission. An alternative scenario would be beneficial, whereby students can explain the similarity without resubmitting their programs.

As seen in Table 10, student justifications of the reported similarity can be classified into three categories: coincidental similarity, academic misconduct, and ambiguous justification. Most of the justifications fall into the first and third categories.

Justifications of coincidental similarity typically argue that the code segments are the only way to do given task (e.g., input and output), the code segments are adapted from accessible sources (e.g., template code or course slides), or the solution is simple and intuitive. It is interesting that a few justifications argue that the similarity occurred since students asked the same instructor for hints. Although the instructors typically provide only general hints, it is possible that in some circumstances they might be unwittingly giving technical hints. All justifications of coincidental similarity are true for some submissions, though we are aware that they can also be used as excuses in cases of academic misconduct.

Justifications regarding academic misconduct primarily cover collusion, with a few covering plagiarism. For the former, students either discussed the solution in detail with colleagues or asked them for technical help. For the latter, the code was either copied from previous assessments, copied by unknown colleagues, or copied from the internet (assuming their colleagues might have done the same and used similar resources).

Ambiguous justifications do not directly address why the similarity has occurred. These are likely to be written by students who are potentially committing academic misconduct, and are therefore unable or unwilling to accurately explain the reason. Most of them insist that the code was independently written, with justifications such as "I wrote the code by myself without any help" or "I did not see other students' programs". Others state that the reported code segments are part of an incorrect/incomplete solution, they are unused code that has not been removed, or they have been further modified for the current submission.

The instructors used the justifications as an additional consideration when detecting programming plagiarism and collusion, and they believe that they are useful. There were a few cases where the instructors were unsure whether a program pair resulted from academic misconduct despite its high similarity. If the students provided plausible justifications, the suspicion was dismissed.

6 CONCLUSION

This paper presents an approach to educate students about programming plagiarism and collusion via automated, personalized, and timely formative feedback. The system deals with the rationalization side of the fraud triangle [6] by providing knowledge about rationalization and showing the futility of engaging in academic misconduct.

According to our two quasi-experiments conducted over two academic semesters, students subject to our approach are more aware of programming plagiarism and collusion and of common yet futile disguises of copied programs. They provide more correct answers in surveys about the matters than students who were not subject to our approach. They are also less likely to have engaged in programming plagiarism and collusion as their submitted programs have lower similarity than those in the control group, even at the level of program flow (type III similarity). If the program similarity was reduced only for types I and II, this might simply show that students had learned how to trick the similarity detector.

Student justification of reported similarities seems to be useful for instructors since it can act as an additional consideration when detecting academic misconduct. However, many students with coincidentally similar program are reluctant to say this in a justification since it would involve resubmitting the program.

Besides the limitation mentioned at various points through the paper, our approach and its evaluation experiments have a number of further limitations, which are to be be addressed in future work.

(1) The approach would not be suitable for institutions that have an honor code or a zero tolerance of plagiarism and collusion, since it permits students to resubmit after receiving a report of code similarity. The approach would need to be further modified to render it useful for such institutions.
(2) The evaluation experiments were performed on early programming courses with weekly assessments, in which the assessment tasks have only a few semantically distinct solutions. Different findings might be obtained by conducting the experiments on courses with more advanced materials, larger assessment tasks, and/or tasks that are open to many semantically distinct solutions.
(3) In the evaluation, the intervention period was during the pandemic while the control period was before the pandemic. While the assessment designs are still comparable, we are aware that these external factors might affect the result and might need further investigation.
(4) The evaluation was performed on 87 students of the control groups and 76 students of the intervention group: all of the students accepted to the informatics faculty of the university in 2019. While we believe

that these numbers are sufficient, further evaluation on different student cohorts with greater numbers might be useful to confirm the findings.

(5) In the evaluation, the reduced similarity degree is based only on the students' final submissions. Considering earlier submissions in the analysis might enrich current findings. In particular, it might confirm that students are learning not to plagiarize or collude rather than engaging in such conduct and then disguising any reported similarities.

(6) The evaluation does not consider number of identified cases of plagiarism and collusion as a metric. Adding this metric to future comparisons might add more value to the findings.

(7) The evaluation experiments were performed only at one institution. Conducting the experiments at different institutions might enrich the findings.

(8) As with many quasi-experiments, we acknowledge the impossibility of making the control and intervention groups completely comparable. Further quasi-experiments might be needed to explore possible external factors influencing the findings.

(9) As previously mentioned, the number of student justifications of reported similarities is low. The system will be changed to make it possible for students to explain the reported similarity without resubmitting the program, either via the similarity report or via the submission page of the next assessment.

(10) The assessment submission system will be further developed for user convenience before being made available for public use. Features to be added include integration to common similarity detectors, dynamic configuration settings, and automatic enrolment from class lists.

While this is not a response to a limitation, we also plan to introduce gamification, to further promote students' engagement with the system while encouraging them to write unique programs and to submit their programs early. We are also exploring the possibility of adjusting the system so that more pressure is applied to students who are more likely to be engaging in plagiarism or collusion.

ACKNOWLEDGMENTS

To Australia Awards Scholarship for supporting the study of the first author. To Gisela Kurniawati, Sendy Ferdian Sujadi, and Rossevine Artha Nathasya from Maranatha Christian University, Indonesia, for facilitating the quasi-experiments in their courses.

REFERENCES

[1] Ibrahim Albluwi. 2019. Plagiarism in programming assessments: a systematic review. *ACM Transactions on Computing Education* 20, 1, Article 6 (2019). https://doi.org/10.1145/3371156

[2] Joe Michael Allen, Frank Vahid, Kelly Downey, and Alex Daniel Edgcomb. 2018. Weekly programs in a CS1 class: experiences with auto-graded many-small programs (MSP). In *ASEE Annual Conference & Exposition.*

[3] Lorin W. Anderson, David R. Krathwohl, Peter W. Airasian, Kathleen A. Cruikshank, Richard E. Mayer, Paul R. Pintrich, James Raths, and Merlin C. Wittrock. 2001. *A taxonomy for learning, teaching, and assessing: a revision of Bloom's taxonomy of educational objectives.* Longman.

[4] Paolo Antonucci, Christian Estler, Durica Nikolić, Marco Piccioni, and Bertrand Meyer. 2015. An incremental hint system for automated programming assignments. In *ACM Conference on Innovation and Technology in Computer Science Education.* 320âĂŞ325. https://doi.org/10.1145/2729094.2742607

[5] 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

[6] Harry Cendrowski and James Martin. 2015. The fraud triangle. In *The Handbook of Fraud Deterrence.* 41–46. https://doi.org/10.1002/9781119202165.ch5

[7] Georgina Cosma, Mike Joy, Jane Sinclair, Margarita Andreou, Dongyong Zhang, Beverley Cook, and Russell Boyatt. 2017. Perceptual comparison of source-code plagiarism within students from UK, China, and South Cyprus higher education institutions. *ACM Transactions on Computing Education* 17, 2 (2017). https://doi.org/10.1145/3059871

[8] W. Bruce Croft, Donald Metzler, and Trevor Strohman. 2010. *Search Engines: Information Retrieval in Practice.* Addison-Wesley.

[9] J. A. W. Faidhi and S. K. Robinson. 1987. An empirical approach for detecting program similarity and plagiarism within a university programming environment. *Computers & Education* 11, 1 (1987), 11–19. https://doi.org/10.1016/0360-1315(87)90042-X
[10] Gregor Fischer and Jürgen Wolff von Gudenberg. 2006. Improving the quality of programming education by online assessment. In *Fourth International Symposium on Principles and Practice of Programming in Java*. 208âĂŞ211. https://doi.org/10.1145/1168054.1168085
[11] Enrique Flores, Alberto Barrón-Cedeño, Lidia Moreno, and Paolo Rosso. 2015. Cross-language source code re-use detection using latent semantic analysis. *Journal of Universal Computer Science* 21, 13 (2015), 1708–1725.
[12] Max Fowler and Craig Zilles. 2021. Superficial code-guise: investigating the impact of surface feature changes on students' programming question scores. In *52nd ACM Technical Symposium on Computer Science Education*. 3âĂŞ9. https://doi.org/10.1145/3408877.3432413
[13] Robert Fraser. 2014. Collaboration, collusion and plagiarism in computer science coursework. *Informatics in Education* 13, 2 (2014), 179–195. https://doi.org/10.15388/infedu.2014.10
[14] Alex Gerdes, Johan Jeuring, and Bastiaan Heeren. 2012. An interactive functional programming tutor. In *17th ACM Annual Conference on Innovation and Technology in Computer Science Education*. 250âĂŞ255. https://doi.org/10.1145/2325296.2325356
[15] Tony Greening, Judy Kay, and Bob Kummerfeld. 2004. Integrating ethical content into computing curricula. In *Sixth Australasian Conference on Computing Education*. 91âĂŞ99.
[16] Sebastian Gross, Bassam Mokbel, Barbara Hammer, and Niels Pinkwart. 2015. Learning feedback in intelligent tutoring systems. *KI - KÃijnstliche Intelligenz* 29, 4 (2015), 413âĂŞ418.
[17] Dirk Grunwald, Elizabeth Boese, Rhonda Hoenigman, Andy Sayler, and Judith Stafford. 2015. Personalized attention @ scale: talk isn't cheap, but it's effective. In *46th ACM Technical Symposium on Computer Science Education*. 610âĂŞ615. https://doi.org/10.1145/2676723.2677283
[18] 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
[19] 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*. 238–243. https://doi.org/10.1145/3059009.3059065
[20] Chris Ireland and John English. 2011. Let them plagiarise: developing academic writing in a safe environment. *Journal of Academic Writing* 1, 1 (2011), 165–172. https://doi.org/10.18552/joaw.v1i1.10
[21] 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
[22] Francisco Jurado, Miguel A. Redondo, and Manuel Ortega. 2012. Using fuzzy logic applied to software metrics and test cases to assess programming assignments and give advice. *Journal of Network and Computer Applications* 35, 2 (2012), 695–712. https://doi.org/10.1016/j.jnca.2011.11.002
[23] Oscar Karnalim and Simon. 2020. Disguising code to help students understand code similarity. In *20th Koli Calling International Conference on Computing Education Research*. Article 13. https://doi.org/10.1145/3428029.3428064
[24] Oscar Karnalim and Simon. 2020. Syntax trees and information retrieval to improve code similarity detection. In *22nd Australasian Computing Education Conference*. 48âĂŞ55. https://doi.org/10.1145/3373165.3373171
[25] Oscar Karnalim and Simon. 2021. Common code segment selection: semi-automated approach and evaluation. In *52nd ACM Technical Symposium on Computer Science Education*. 335âĂŞ341. https://doi.org/10.1145/3408877.3432436
[26] Oscar Karnalim and Simon. 2021. Explanation in code similarity investigation. *IEEE Access* 9 (2021), 59935âĂŞ59948. https://doi.org/10.1109/ACCESS.2021.3073703
[27] Oscar Karnalim and Simon. 2021. Relationship between code similarity and course semester in programming assessments. In *IEEE International Conference on Advanced Learning Technologies*. 23–24. https://doi.org/10.1109/ICALT52272.2021.00014
[28] Oscar Karnalim, Simon, Mewati Ayub, Gisela Kurniawati, Rossevine Artha Nathasya, and Maresha Caroline Wijanto. 2021. Work-in-progress: syntactic code similarity detection in strongly directed assessments. In *IEEE Global Engineering Education Conference*. 1162âĂŞ1166. https://doi.org/10.1109/EDUCON46332.2021.9454152
[29] Oscar Karnalim, Simon, and William Chivers. 2019. Similarity detection techniques for academic source code plagiarism and collusion: a review. In *IEEE International Conference on Engineering, Technology and Education*. https://doi.org/10.1109/TALE48000.2019.9225953
[30] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2021. A tutoring system to learn code refactoring. In *52nd ACM Technical Symposium on Computer Science Education*. 562–568. https://doi.org/10.1145/3408877.3432526
[31] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2018. A systematic literature review of automated feedback generation for programming exercises. *ACM Transactions on Computing Education* 19, 1, Article 3 (2018). https://doi.org/10.1145/3231711
[32] Thomas Lancaster. 2018. Academic integrity for computer science instructors. In *Higher Education Computer Science*. 59–71. https://doi.org/10.1007/978-3-319-98590-9_5
[33] Nguyen-Thinh Le, Wolfgang Menzel, and Niels Pinkwart. 2009. Evaluation of a constraint-based homework assistance system for logic programming. In *17th International Conference on Computers in Education*. 51–58.
[34] Tri Le, Angela Carbone, Judy Sheard, Margot Schuhmacher, Michael de Raadt, and Chris Johnson. 2013. Educating computer programming students about plagiarism through use of a code similarity detection tool. In *International Conference on Learning and Teaching in*

Computing and Engineering. 98–105. https://doi.org/10.1109/LaTiCE.2013.37

[35] Chao Liu, Chen Chen, Jiawei Han, and Philip S Yu. 2006. GPlag: detection of software plagiarism by program dependence graph analysis. In *12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 872–881. https://doi.org/10.1145/1150402.1150522

[36] Maxim Mozgovoy, Sergey Karakovskiy, and Vitaly Klyuev. 2007. Fast and reliable plagiarism detection system. In *37th IEEE Annual Frontiers in Education Conference*. 11–14. https://doi.org/10.1109/FIE.2007.4417860

[37] Susanne Narciss. 2008. Feedback strategies for interactive learning tasks. In *Handbook of Research on Educational Communications and Technology*. 125–144.

[38] Lawton Nichols, Kyle Dewey, Mehmet Emre, Sitao Chen, and Ben Hardekopf. 2019. Syntax-based improvements to plagiarism detectors and their evaluations. In *24th ACM Conference on Innovation and Technology in Computer Science Education*. 555–561. https://doi.org/10.1145/3304221.3319789

[39] 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

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

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

[42] 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.

[43] Eric Roberts. 2002. Strategies for promoting academic integrity in CS courses. In *32nd IEEE Annual Frontiers in Education*. https://doi.org/10.1109/FIE.2002.1158209

[44] Francisco Rosales, Antonio García, Santiago Rodríguez, José L. Pedraza, Rafael Méndez, and Manuel M. Nieto. 2008. Detection of plagiarism in programming assignments. *IEEE Transactions on Education* 51, 2 (2008), 174–183. https://doi.org/10.1109/TE.2007.906778

[45] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. 2009. Comparison and evaluation of code clone detection techniques and tools: a qualitative approach. *Science of Computer Programming* 74, 7 (2009), 470–495. https://doi.org/10.1016/J.SCICO.2009.02.007

[46] 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

[47] Robert Sedgewick and Kevin Wayne. 2011. *Algorithms* (4 ed.). Pearson.

[48] Simon, Beth Cook, Judy Sheard, Angela Carbone, and Chris Johnson. 2014. Academic integrity perceptions regarding computing assessments and essays. In *10th ACM Annual Conference on International Computing Education Research*. 107–114. https://doi.org/10.1145/2632320.2632342

[49] 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 *Working Group Reports on Innovation and Technology in Computer Science Education*. 1âĂŞ19. https://doi.org/10.1145/3437800.3439201

[50] Simon, Raina Mason, Tom Crick, James H. Davenport, and Ellen Murphy. 2018. Language choice in introductory programming courses at Australasian and UK universities. In *49th ACM Technical Symposium on Computer Science Education*. 852–857. https://doi.org/10.1145/3159450.3159547

[51] Simon, Judy Sheard, Michael Morgan, Andrew Petersen, Amber Settle, and Jane Sinclair. 2018. Informing students about academic integrity in programming. In *20th Australasian Computing Education Conference*. 113–122. https://doi.org/10.1145/3160489.3160502

[52] 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*. 57–80. https://doi.org/10.1145/3024906.3024910

[53] Jaime Spacco, Davide Fossati, John Stamper, and Kelly Rivers. 2013. Towards improving programming habits to create better computer science course outcomes. In *18th ACM Conference on Innovation and Technology in Computer Science Education*. ACM, 243âĂŞ248. https://doi.org/10.1145/2462476.2465594

[54] Narjes Tahaei and David C. Noelle. 2018. Automated plagiarism detection for computer programming exercises based on patterns of resubmission. In *ACM Conference on International Computing Education Research*. 178–186. https://doi.org/10.1145/3230977.3231006

[55] Nghi Truong, Paul Roe, and Peter Bancroft. 2004. Static analysis of students' Java programs. In *Sixth Australasian Conference on Computing Education*. 317âĂŞ325. https://doi.org/10.5555/979968.980011

[56] Herbert H. Tsang, Alice Schmidt Hanbidge, and Tony Tin. 2018. Experiential learning through inter-university collaboration research project in academic integrity. In *23rd Western Canadian Conference on Computing Education*. https://doi.org/10.1145/3209635.3209645

[57] Aurora Vizcaino. 2005. A simulated student can improve collaborative learning. *International Journal of Artificial Intelligence in Education* 15, 1 (2005), 3âĂŞ40.

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

[59] Lisa Yan, Nick McKeown, Mehran Sahami, and Chris Piech. 2018. TMOSS: Using intermediate assignment work to understand excessive collaboration in large classes. In *49th ACM Technical Symposium on Computer Science Education*. 110–115. https://doi.org/10.1145/

3159450.3159490

[60] Feng-Pu Yang, Hewijin Christine Jiau, and Kuo-Feng Ssu. 2014. Beyond plagiarism: an active learning method to analyze causes behind code-similarity. *Computers & Education* 70 (2014), 161–172. https://doi.org/10.1016/J.COMPEDU.2013.08.005