education sciences

MDPI

Article

Maintaining Academic Integrity in Programming: Locality-Sensitive Hashing and Recommendations

Oscar Karnalim

Faculty of Information Technology, Maranatha Christian University, Bandung 40164, Indonesia; oscar.karnalim@it.maranatha.edu

Abstract: Not many efficient similarity detectors are employed in practice to maintain academic integrity. Perhaps it is because they lack intuitive reports for investigation, they only have a command line interface, and/or they are not publicly accessible. This paper presents SSTRANGE, an efficient similarity detector with locality-sensitive hashing (MinHash and Super-Bit). The tool features intuitive reports for investigation and a graphical user interface. Further, it is accessible on GitHub. SSTRANGE was evaluated on the SOCO dataset under two performance metrics: f-score and processing time. The evaluation shows that both MinHash and Super-Bit are more efficient than their predecessors (Cosine and Jaccard with 60% less processing time) and a common similarity measurement (running Karp-Rabin greedy string tiling with 99% less processing time). Further, the effectiveness trade-off is still reasonable (no more than 24%). Higher effectiveness can be obtained by tuning the number of clusters and stages. To encourage the use of automated similarity detectors, we provide ten recommendations for instructors interested in employing such detectors for the first time. These include consideration of assessment design, irregular patterns of similarity, multiple similarity measurements, and effectiveness–efficiency trade-off. The recommendations are based on our 2.5-year experience employing similarity detectors (SSTRANGE's predecessors) in 13 course offerings with various assessment designs.

Keywords: programming; plagiarism; collusion; similarity detection; recommendations; higher education

Citation: Karnalim, O. Maintaining Academic Integrity in Programming: Locality-Sensitive Hashing and Recommendations. *Educ. Sci.* **2023**, *13*, 54. https://doi.org/10.3390/educsci13010054

Academic Editors: Mike Joy and Peter Williams

Received: 3 December 2022
Revised: 21 December 2022
Accepted: 29 December 2022
Published: 3 January 2023

1. Introduction

Programmer is a high-demand job, which is projected to have a 22% demand increase in 2030 (https://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm#tab-6 accessed on 2 November 2022). In that job, reusing one's code is quite common since programmers often search online for references and many of them put their code in online public repositories (e.g., GitHub). However, while reusing code, some programmers fail to acknowledge the original sources, which can lead to plagiarism or collusion. The former refers to reusing code without acknowledgment and awareness of the original author [1]. The latter is similar, except that the original author is aware of the act. There is a need to inform future programmers about the matter in academia [2].

In programming education, instructors typically maintain academic integrity by informing students about their expectation regarding the matter [3] and penalizing students whose programs are suspected of plagiarism or collusion. Given that searching similar programs manually can be a tedious task [4], automated similarity detectors, such as MOSS [5], are sometimes employed to initially filter similar programs.

There are a number of automated similarity detectors [6], many of which detect similarities based on program structure. Although such a mechanism is effective, it is impractical for assessments with large submissions; extracting and comparing program structure can be time consuming [7].

A few similarity detectors aim for efficiency, but according to our observation on papers listed in a recent review [7], they do not report which parts of the programs are

similar. This is perhaps because mapping similar features to the programs is not straightforward. Reporting only similarity degrees in percentages complicates the investigation process as evidence should be provided for any suspicion [8]. Instructors need to locate the similarities by themselves. Further, the similarity detectors are only featured with a command line interface, at which instructors may experience technical difficulties with [9]. In addition, the similarity detectors are not publicly accessible. Instructors need to contact the authors or recreate these detectors, with no guarantee that the detectors will satisfy the instructors' needs.

Employing automated similarity detectors is another challenge, as the effectiveness of similarity detectors relies heavily on the assessment design. Common similarity detectors, for instance, may not work on weekly assessments that expect the solutions to be syntactically similar [10]. Moreover, not all similarities are useful for raising suspicion; some are expected among submissions, such as template code or obvious implementation [11]. Experience in employing similarity detectors on various assessment designs needs to be shared with instructors interested in using similarity detectors for the first time.

In response to the aforementioned issues, this paper presents SSTRANGE, an efficient automated similarity detector with locality-sensitive hashing [12]. The algorithm is relatively fast and tends to generate the same hash for similar submissions. SSTRANGE reports not only similarity degrees among Java and Python submissions but also the similarities in HTML pages. SSTRANGE is featured with a graphical user interface, and it is publicly accessible on GitHub (https://github.com/oscarkarnalim/SSTRANGE accessed on 2 November 2022) along with the documentation and the code. The tool is now being used by our faculty to raise suspicion of plagiarism and collusion. We also report ten recommendations from our 2.5-year experience employing automated similarity detectors (SSTRANGE's predecessors) in courses offered at our faculty with various assessment designs. It is summarized from discussions with instructors teaching 13 course offerings.

To the best of our knowledge, our study is the first of its kind. The study has four research questions:

- RQ1: Is locality-sensitive hashing more efficient than common similarity algorithms used for automated similarity detectors?
- RQ2: How much is the effectiveness trade-off introduced by locality-sensitive hashing?
- RQ3: What are the impacts of overlapping adjacent token substrings, number of clusters, and number of stages in locality-sensitive hashing for similarity detection?
- RQ4: Which are the recommendations for employing automated similarity detectors for the first time?

2. Literature Review

To maintain academic integrity in programming, instructors inform students about the matter and penalize those who are involved in such misconduct. While informing students about academic integrity is practical, penalizing students suspected of plagiarism and collusion can be challenging. Any suspicion should be supported with strong evidence [8], but manually checking all submissions for evidence is impractical [4]. Automated similarity detectors are therefore employed by some instructors to initially select similar submissions, which they will further investigate.

Many similarity detectors aim for effectiveness (i.e., reporting a large proportion of copied submissions) by comparing program structure among submissions. Token string is a popular example of such program structure; it is an array of string representing source code tokens ('words'). The representation is employed by a number of similarity detectors, including the popular ones [6]: MOSS [5], JPlag [13], and Sherlock [8]. Some similarity detectors consider the tokens without further preprocessing [4,14]; others generalize identifiers [15,16], generalize program statements [17,18], and/or ignore several types of tokens [19,20].

Three more advanced representations are sometimes used for similarity detectors. They are argued to be more semantic preserving. Compiled token string is an array of

tokens obtained from executable files [21,22]. Syntax tree is generated from grammars of the programming language [23]. Sometimes, the tree is linearized to the token string for efficient comparison [24,25]. Program dependency graphs map interactions among program statements [26,27].

For the purpose of efficiency, a number of similarity detectors consider program characteristics instead of program structure. Early instances of such similarity detectors consider occurrence frequencies of certain types of tokens [28] and/or syntax constructs [29,30]. The set of n-gram token substrings is therefore employed by later versions for better accuracy. The set is formed by breaking down token string to substrings with the size of n, and counting the occurrence frequencies [31,32].

For measuring similarities, efficient similarity detectors often adapt techniques from classification, clustering, and information retrieval [7]. Occurrence frequencies of program characteristics are considered features.

Classification-based similarity detectors argue that plagiarism and collusion are expected to have a particular pattern in their features. Ullah et al. [33] use principal component analysis and multinomial logistic regression to detect cross-language plagiarism and collusion. Yasaswi et al. [34] use support vector machine with learning features trained from source code of a Linux kernel. Elenbogen and Seliya [35] use decision trees to detect plagiarism and collusion via programming style.

Clustering-based similarity detectors argue that suspicious submissions can be clustered as one group based on their similarity. Jadalla and Elnagar [36] group submissions with a density-based partitional clustering algorithm, DBScan. Acampora and Cosma [19] rely on fuzzy C-means for clustering while Moussiades and Vakali [37] rely on their dedicated algorithm, WMajorClust.

Information-retrieval-based similarity detectors preprocess submissions to index prior comparisons. In such a manner, program similarity can be calculated in linear time. Cosine correlation is employed by Flores et al. [38] and Foltynek et al. [39]. Latent semantic analysis is employed by Ullah et al. [40] and Cosma and Joy et al. [41].

A few similarity detectors combine two similarity measurements. Ganguly et al. [42], for example, combine field-based retrieval with a random forest classifier for higher effectiveness. Mozgovoy et al. [43] introduce an information-retrieval-based filtering prior to comprehensive comparison with Plaggie [18], to compensate efficiency trade-off introduced by Plaggie that relies on program structure.

To help instructors collect evidence, some similarity detectors map and highlight detected similarities in suspected submissions (see MOSS for example). However, to the best of our knowledge, such a reporting mechanism is not available for efficient similarity detectors, even for those relying on set of n-gram token substrings. Instructors exclusively rely on reported similarity degrees, which are clearly insufficient evidence. High similarity does not necessarily entail academic dishonesty [44]. Further, existing efficient similarity detectors are neither featured with a graphical user interface for instructors' convenience nor are they publicly accessible (to immediately test the tools on their teaching environment).

A number of studies report experiences using similarity detectors on teaching environments. Bowyer and Hall [45] report their experience on employing MOSS in a C programming course. They found that the tool can facilitate detection of academic misconduct although it does not cover the 'ghost author' case, where a student pays a person to complete their work. Pawelczak [46] also reports how their own similarity detector is used in a C programming course in engineering. They found that the tool usage reduces instructors' workload but discourages solidarity among students. Le Nguyen et al. [47] integrate MOSS and JPlag on Moodle, and ask both instructors and students about their experience with the tool for a programming assessment. They found that the integration is helpful for classes with a large number of students, where manual checking is not possible. It can also deter students for breaching academic misconduct. They also suggested that assessments should be neither reused nor adapted from other assessments. Karnalim et al. [10] summarize how instructors check for irregularities in similar programs to raise

suspicion of plagiarism and collusion. They found that in addition to verbatim copying, unusual superficial look, and inadequate de-identification can be a basis for evidence. Incompleteness or over-completeness of the work can also be useful.

To the best of our knowledge, none of these studies discuss their experience employing similarity detectors on various assessment designs. They are only focused on a particular assessment design and that might be less useful for instructors interested in using similarity detectors for the first time.

3. Our Search for Suitable Similarity Detectors

For contextual purpose, we will explain our search for suitable similarity detectors to maintain academic integrity in our programming courses. In 2016, we were interested in employing similarity detectors and our initial goal was to nullify syntactic variations as much as possible (assuming these were attempts to disguise copied programs). We first experimented with JPlag [13] and developed a prototype that can detect similarity based on program binaries [48]. Two years later, we realized that nullifying many syntactic variations is not practical for our weekly assessments since programs would be considered similar regardless of whether they are copied or not. It is difficult to gather evidence for raising suspicion. Further, high similarities are expected among assessments with easy tasks (e.g., writing *hello world* program) and those with detailed directions (e.g., translating algorithms to program code) [10,44]. In addition, implementing advanced program disguises might be considered a substantial programming task for students [8].

In 2019, we started developing STRANGE, a similarity detector that focuses on gathering evidence. The tool relies on syntactic similarity and explains all reported similarities in human language [15]. It works by converting programs to token strings, generalizing some of the tokens, and comparing them one another with Karp–Rabin greedy string tiling (RKRGST). The tool had been used since January 2020 in our faculty and was further developed to CSTRANGE (Comprehensive STRANGE) in 2021 [49]. CSTRANGE relies on three levels of granularity in reporting similarities, making it useful for many kinds of assessments. Further, it is featured with a graphical user interface. CSTRANGE's detection is similar to that of STRANGE except that it uses three levels of preprocessing. The tool has been used in our courses since January 2022.

Both STRANGE and CSTRANGE focus on effectiveness and comprehensiveness; their processing time is quite slow for large submissions. Hence, this motivates us to develop SSTRANGE (Scalable STRANGE), an efficient similarity detector with locality-sensitive hashing, a clustering-based similarity measurement. The tool highlights similarities for supporting suspicion and features a graphical user interface. Unique to SSTRANGE, it optimizes the reporting mechanism and shows only crucial information for instructors. The tool, its source code, and its documentation are available on GitHub (https://github.com/oscarkarnalim/SSTRANGE accessed on 2 November 2022).

4. SSTRANGE: Scalable STRANGE

The tool accepts Java and Python submissions, compares them with one another, and generates similarity reports in an HTML format. Figure 1 shows that SSTRANGE works in five steps. First, it accepts a directory containing student submissions in which each submission can be represented as a single Java/Python code file, a project directory of Java/Python code files, or a zip file containing such a project directory. All submissions are then converted to token strings (tokenisation) with the help of ANTLR [50]. Comments and white space are ignored since they can be easily modified with limited programming knowledge [13]. Further, identifiers are replaced with their general names, as they are commonly modified as part of disguising the copied programs [51]. The replacement also applies for constants and some data types.

Figure 1. Five steps of SSTRANGE for detecting program similarities.

Second, each submission's token string is converted to an index, a set of key-value tuples at which keys refer to token substrings and values refer to their occurrence frequencies [52]. The token string is split to overlapping n-adjacent tokens, where n is the minimum matching length of matches (modifiable but set to 20 by default). For instance, if the token string is {'public', 'static', 'void', 'main'} and n is set to 2, the resulting substrings are 'public static', 'static void', and 'void main'. A larger n typically results in a higher proportion of reported copied programs, but with a trade-off ignoring copied programs with moderate evidence. Occurrence frequencies of any distinct token substrings are then counted and mapped to indexes.

Third, all indexes are compared with each other using either MinHash or Super-Bit. MinHash [53] is a locality-sensitive hashing algorithm that generates the same hash for similar submissions and put them in the same cluster. The algorithm relies on Jaccard coefficient [52], a similarity measurement that considers which distinct token substrings are shared. MinHash requires two arguments. The number of clusters (buckets) determines how many buckets will be used for clustering (two by default); programs falling on the same cluster will be considered similar. More clusters tend to limit the number of reported copied programs. The number of stages determines how many clustering tasks will be performed (one by default). Programs are considered similar if they fall on the same cluster at least once. More stages tend to report more copied programs.

Super-Bit [54] is another locality-sensitive hashing algorithm. Instead of relying on Jaccard, it uses Cosine similarity [52], which considers occurrence frequencies of shared distinct token substrings. Similar to MinHash, it requires number of clusters and numbers of stages.

Fourth, matches for submissions with the same hash (i.e., located in the same bucket) are selected by mapping shared token substrings to original forms of the submissions. Any adjacent or overlapping matches will be merged, ensuring that each large match is reported as a single match instead of several shorter matches.

Fifth, pairwise similarity reports are generated for submissions, in which the average similarity degree [13] is no less than the minimum similarity threshold (modifiable but set to 75% by default). The average similarity degree is calculated as $2 \times match(A, B) \ / \ (len(A) + len(B))$, where $match(A, B)$ is the matched tokens, while both $len(A)$ and $len(B)$ are the total tokens of each involved submission. The instructor can also set the maximum number of pairwise similarity reports shown (10 by default). A larger minimum similarity threshold and/or smaller maximum number of reports means stronger evidence for reported copied programs and faster similarity report generation. However, the tool might not report copied programs with the moderate level of evidence. These might be useful for those teaching large classes; instructors are only able to investigate a small proportion of the programs and they want to prioritize copied programs with strong evidence.

Some similarities are not useful for raising suspicion [11]. They can be legitimately copiable code or code needed for compilation. They can also be a result of intuitive or suggested implementation. The first two are template code and instructors are expected to provide such code, which will be removed from submissions prior to comparison with a simple search mechanism. The other two are naturally common code; they are first selected based on the occurrence frequencies and then removed in the same manner as that of template code. The removal mechanisms are adapted from CSTRANGE.

SSTRANGE's graphical user interface for input can be seen in Figure 2. Assessment path refers to a directory path containing all submissions at which each of them is represented either as a code file, a directory, or a zip file. Submission type refers to the

representation of each submission (code file, directory, or zip file). Submission language refers to programming language of the submissions, either Java or Python. Explanation language refers to human language of the generated similarity reports. It can be either English or Indonesian.

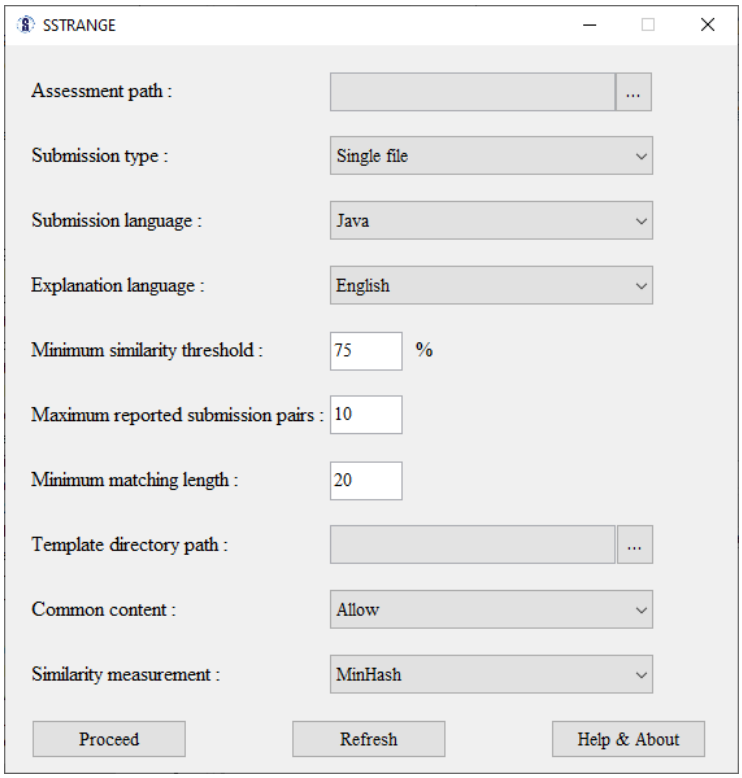

Figure 2. SSTRANGE's user interface for input.

Minimum similarity threshold can be set from 0% to 100%. Only programs whose average similarity degree no less than such a threshold will be reported. Maximum reported submission pairs can be set with any positive integer no more than the total number of program pairs. It defines the maximum number of program pairs reported. Minimum matching length can be set with any positive integer no less than two. Larger minimum matching length can result in stronger evidence for raising suspicion.

Template directory path refers to a directory containing all template code files. Common content (code), if not allowed, will be selected and removed prior to comparison.

SSTRANGE provides five similarity measurements: MinHash, Super-Bit, Jaccard, Cosine, and running Karp–Rabin greedy string tiling. MinHash and Super-Bit are locality-sensitive hashing similarity measurements that are also considered as clustering-based similarity measurements. Exclusive to these two, pop-ups asking the number of clusters and the number of stages will be shown prior to processing the inputs. Jaccard and Cosine [52] are information-retrieval-based similarity measurements. They tend to result in higher effectiveness than MinHash and Super-Bit, but with a trade-off in processing time. Both measurements apply similar steps as MinHash and Super-Bit. Running Karp–Rabin greedy string tiling (RKRGST) [55] is a structure-based measurement that is common for similarity detectors [7]. It converts submissions to token strings and pairwise compares them. RKRGST is probably the most effective among our five similarity measurements, but it is also the slowest.

Based on the inputs, SSTRANGE will generate similarity reports at which their navigation layout can be seen in Figure 3. The layout is inspired from CSTRANGE and it lists all reported program pairs with their average similarity degrees and same cluster occurrences (automatically set to one for Jaccard, Cosine, and RKRGST). The instructor can

sort the program pairs based on a particular column by clicking its header. Information about similarity measurements and the tool is also provided on the right panel. Further investigation can be conducted by clicking each program pair's 'observe' button. It will redirect the instructor to the pairwise similarity report.

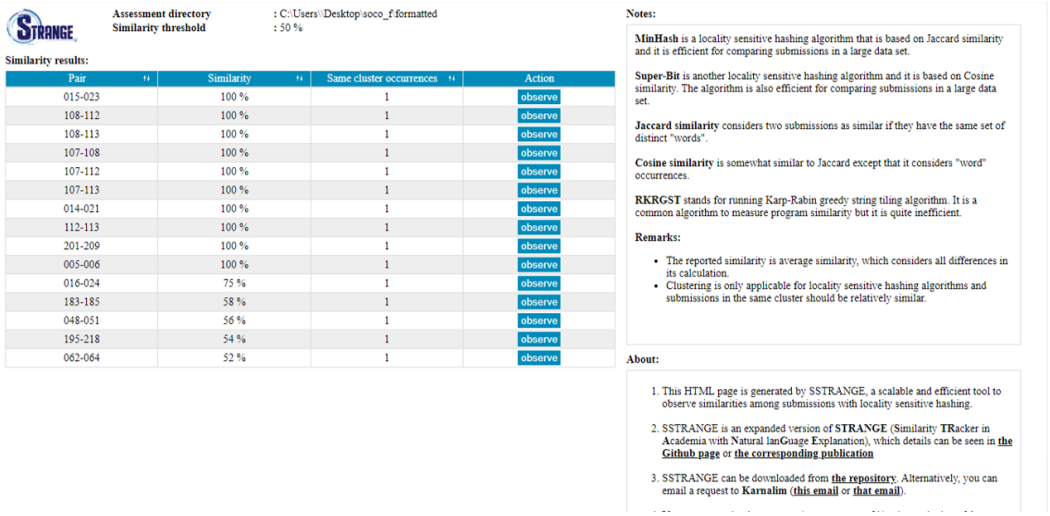

Figure 3. SSTRANGE's navigation layout at which each column can be sorted in ascending/descending order by clicking the arrow toggle sign at the header.

Layout of the pairwise similarity report can be seen in Figure 4 and again, it is inspired from CSTRANGE. It shows the compared programs and highlights the similarities. Once a similar code segment is clicked, it will be highlighted in a darker red color for both programs. Further, the similarity will be explained in the explanation panel.

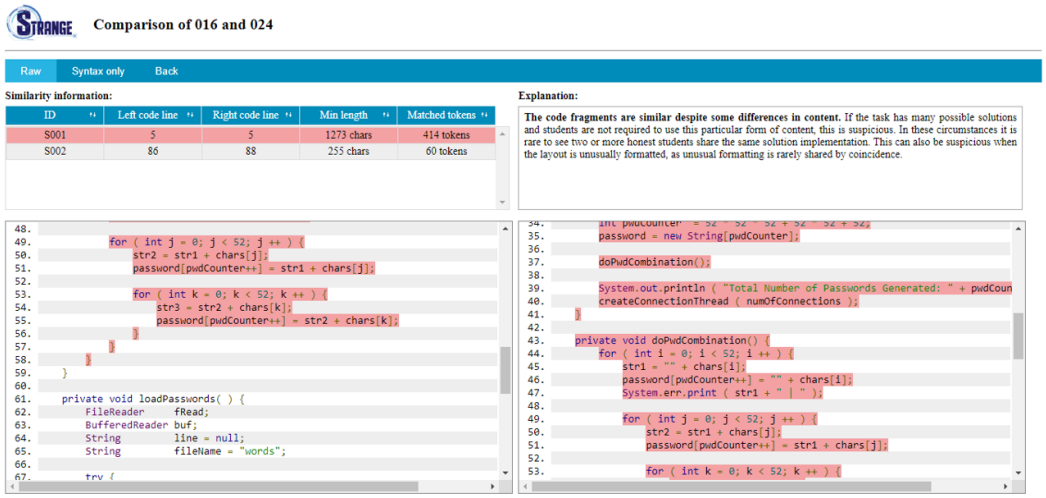

Figure 4. SSTRANGE's generated pairwise similarity report.

The instructor can see all similarities listed in the table. Each similar code segment is featured with the left code line (number), right code line (number), minimum (number of matched) character length, and number of matched tokens. If needed, similar code segments can be sorted based on a particular feature by clicking the column header.

For instructors interested in integrating SSTRANGE to larger systems or just simply to automate the input process, they can run SSTRANGE via a command prompt interface. The tool requires eleven arguments: assessment path, submission type, submission language, explanation language, minimum similarity threshold, maximum reported submission pairs, minimum matching length, template directory path, common content flag, similarity measurement, and resource path. The first ten arguments are the same to those of SSTRANGE's

graphical user interface, while resource path, the last one, refers to a directory path where all SSTRANGE's supporting files are located. For locality-sensitive hashing algorithms, two more arguments can be added: number of clusters and number of stages. Further details can be found on the documentation on Github.

5. Methodology for Addressing the Research Questions

This section describes how we addressed the four research questions mentioned at the end of the introduction. The first three were addressed by evaluating SSTRANGE, while the last one was addressed by reporting our instructors' experience in employing similarity detectors.

5.1. Addressing RQ1, RQ2, and RQ3: SSTRANGE's Evaluation

This section addresses the first three research questions. RQ1 is about whether locality-sensitive hashing is more efficient than common similarity algorithms used for automated similarity detectors. RQ2 is about how much effectiveness trade-off is introduced by locality-sensitive hashing. RQ3 is about the impact of overlapping adjacent tokens, number of clusters, and number of stages in locality-sensitive hashing for similarity detection.

For RQ1 (efficiency), we employed processing time (seconds) as the metric where a lower value is preferred. It was measured for all similarity measurements starting from the first step (tokenisation) to the fourth (matches selection). The last step (report generation) is not included as such a mechanism is essentially the same for all similarity measurements.

For RQ2 (effectiveness), we employed f-score [52], the harmonic mean of precision and recall. The metric is represented as percentage (0–100%), where a high value is preferred. It is measured as $(precision \times recall)/(precision + recall)$. Precision is the proportion of copied and retrieved program pairs to all retrieved program pairs. Recall, on the other hand, is the proportion of copied and retrieved program pairs to all copied program pairs.

Five scenarios were considered in the evaluation. Each of them refers to one of our similarity measurements: MinHash, Super-Bit, Jaccard, Cosine, and RKRGST. The first two are our locality-sensitive hashing measurements and the rest are common measurements in similarity detection, especially RKRGST [7]. Each scenario was conducted on the SOCO dataset [56] provided by Ragkhitwetsagul et al. [57]. It is a publicly available dataset that is often used in evaluating similarity detectors. The dataset has 258 Java submissions with 97 copied program pairs from a programming competition. All scenarios were set to retrieve 100 suspected program pairs.

For RQ3 (impact of overlapping adjacent token substrings, number of clusters, and number of stages), f-score and processing time were considered as comparison metrics. The impacts of overlapping adjacent tokens in building indexes were measured by comparing that to scenarios without such overlapping. Jaccard and Cosine scenarios were also involved in addition to MinHash and Super-Bit, as they also rely on indexes.

The impact of number of clusters was measured by comparing the performance of MinHash and Super-Bit scenarios in nine arbitrary numbers of clusters, starting from two to ten in one interval (2, 3, 4, . . . , 10); number of stages is set to one by default. The impact of number of stages was measured in a similar manner, but with the first ten positive integers as arbitrary numbers of stages and two as the number of clusters.

Prior to addressing the three research questions, it is recommended to know the most effective minimum matching length for each scenario. Such optimal length varies across datasets and scenarios. This was obtained by testing ten constants that are first multiples of ten (10, 20, 30, . . . , 100). The next ten multiples of ten would be considered if at least one scenario had not reached its effectiveness peak. The number of clusters of MinHash and Super-Bit was set to two and their number of stages was set to one. These are minimum reasonable values for both arguments.

5.2. Addressing RQ4: Recommendations

This section addresses RQ4, which is about recommendations on employing automated similarity detectors for the first time. They are compiled from two-and-one-half years of instructors' experience employing SSTRANGE's predecessors, from January 2020 to June 2022.

For each academic semester, instructors interested in employing similarity detectors in their courses were contacted. They were given a short briefing about how to use a similarity detector and how to interpret the similarity reports. The similarity detector was usually run for each two or three weeks, covering assessments with a passed deadline during that period. We then asked them about their experience and concerns after each use. Any urgent concerns were addressed immediately but only relevant ones were reported here. At the end of the semester, we asked them about overall experience and concerns. Some of which would be considered for the next semester's integration. All of the findings were qualitatively summarized and converted to recommendations.

Course offerings in which similarity detectors were employed can be seen in Table 1; they are sorted based on their semesters. Each offering is featured with its major (IT for information technology and IS for information system), expected student enrollment year, and number of enrolled students. In total, there are 13 offerings of 9 distinct courses, with 435 students involved. Introductory programming is counted as two different courses given that it was offered to two majors. All courses were compulsory for the students and they were offered online due to the pandemic.

Table 1. Courses in which similarity detectors were employed.

Semester	Name	Major	Enrollment Year	Students
2020 1st sem	Basic data structure	IT	1st year	46
	Adv. data structure	IT	2nd year	9
2020 2nd sem	Introductory prog.	IS	1st year	33
	Adv. data structure	IT	2nd year	43
2021 1st sem	Object oriented prog.	IS	1st year	37
	Introductory prog.	IT	1st year	45
2021 2nd sem	Introductory prog.	IS	1st year	35
	Business application prog.	IS	3rd year	19
	Data structure	IT	2nd year	33
	Adv. object oriented prog.	IT	3rd year	32
2022 1st sem	Object oriented prog.	IS	1st year	15
	Introductory prog.	IT	1st year	55
	Machine intelligence	IT	3rd year	33

In the first semester of 2020, we used STRANGE in basic and advanced data structure courses offered to IT undergraduates. Basic data structure covered concepts of data structure and several linear data structures (e.g., array, stack, queue, and linked list) in Python. Students were given weekly assessments that should be completed in lab sessions (two hours each). Per assessment, students were expected to translate and/or implement algorithms to code.

Advanced data structure covered non-linear data structures such as trees and graphs. The course also introduced a bridging language from Python to Java and C#. Python was the first programming language for IT undergraduates, while Java and C# were two programming languages that they would heavily use in later courses. The bridging language was written on top of Java but with some syntax simplified. The assessment design is comparable to that of basic data structure except that some of the topics are about translating from one programming language to another.

In the next semester, STRANGE was employed on another offering of advanced data structure. This time, students were expected to complete two assessments per week. One should be completed in the corresponding lab session (two hours), while another should be submitted before next week. The assessments were essentially about the same

tasks—translating and/or implementing algorithms to code. Homework assessments were designed to encourage students finishing tasks that they had not completed during the lab session. Previous offering of advanced data structure (first semester of 2020) had a limited number of enrolled students (9) as it was dedicated to repeating students.

STRANGE was also employed on introductory programming for IS undergraduates. The course covered basic programming concepts, including input, output, branching, looping, function, and array. Students were given weekly lab and homework assessments. Lab assessments were to be completed within the lab session, while homework assessments were to be completed at home and submitted before next week's lab session. Each assessment consisted of three tasks: easy, medium, and challenging. Per task, students were expected to solve it in both Java and Python (counted as two assessments). On most occasions, they solved the tasks in Java first and then converted the solutions to Python.

In the first semester of 2021, two more courses were considered. Object-oriented programming was the successive course of introductory programming for IS undergraduates. It covered the concepts of object-oriented programming (class, interface, overriding, abstraction, etc.) but only in Java. The course also covered basic application development with Java Swing. Per week, students were given lab assessments. Homework assessments were also issued but only for the first half of the semester. Each assessment required the students to implement object-oriented concepts and/or solving a case study.

Introductory programming offered to IT undergraduates was another course employing STRANGE in that semester. The course and its assessment design were comparable to that offered to IS undergraduates except that IT introductory programming only used Python as the programming language.

In the second semester of 2021, four courses employed STRANGE: introductory programming, business application programming, data structure, and advanced object-oriented programming. The first half were offered to IS undergraduates, while the remaining were offered to IT undergraduates. The offering of IS introductory programming was similar to that in the second semester of 2020. Business application programming was the successive course of object-oriented programming offered one semester before. It covered advanced concepts of Java application development and database. The course issued weekly lab assessments about developing simple applications.

The second half were offered to IT undergraduates. Data structure was a new course introduced in the new curriculum to replace both basic and advanced data structure. Assessments were mostly about implementing linear data structure in Python, and they were issued as homework that should be completed within one week. Advanced object oriented programming covered the same materials as IS business application programming: Java application development with database. Each week, the offering issued two assessments covering the same tasks. The first one should be completed in the lab session, while the second one should be completed within one week. The purpose of homework assessments was similar to that of IT advanced data structure offered in the second semester of 2020: encouraging students to finish tasks that they had not completed in the lab session.

In 2022, we replaced STRANGE with CSTRANGE as the latter is featured with a graphical user interface and reports three layers of similarity. CSTRANGE was employed on three courses during the first semester: IS object-oriented programming, IT introductory programming, and IT machine intelligence. Object-oriented programming and introductory programming were offered in the same manner as those offered in the first semester of 2021.

Machine intelligence was about artificial intelligence (AI) and the assessments expected students to implement such AI in Python with some predefined libraries. Students were expected to complete such assessments in the lab session. However, if they were not able to do that, they could resubmit the revised version within one week.

6. Results and Discussion

This section reports and discusses our findings from addressing the research questions.

6.1. The Most Effective Minimum Matching Length for the Similarity Measurements

Before addressing RQ1–RQ3, it is recommended to search the most effective minimum matching length (MML) for each similarity measurement. While a larger MML threshold can prevent many coincidental matches to be reported, it can also lower the chance of copied programs to be suspected as the resulted similarity degree is reduced. Figure 5 shows that each similarity measurement has different optimal MML. MinHash and RKRGST reach its peak at 20 with 70% and 94% f-score respectively; Super-Bit reaches its peak at 60 with 81% f-score, and Jaccard reaches their peak at 30 with 93% f-score. Cosine is the only measurement that did not reach its peak even after its MML was set to 100. After evaluating another set of ten MML thresholds (110 to 200), it appears that Cosine reached its peak at 130 with 84% f-score.

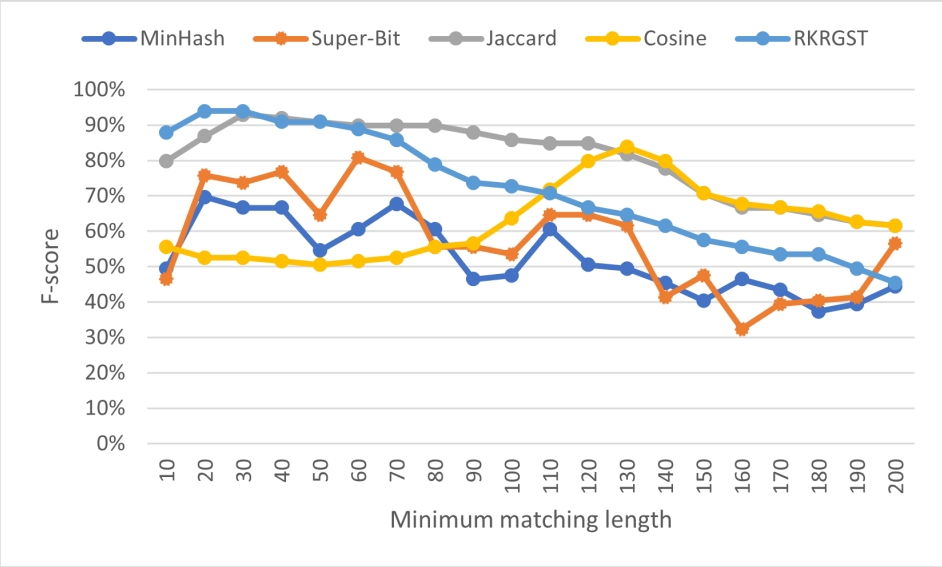

Figure 5. F-score (%) across various minimum matching lengths for all scenarios.

Setting each similarity measurement with its most effective MML is reasonable. Table 2 shows that on most similarity measurements, processing time only becomes a little longer on larger MML. RKRGST is the only similarity measurement at which such a trend is not applicable due to its greedy mechanism for searching matches.

Table 2. Processing time (seconds) of various minimum matching lengths for all scenarios.

MML	MinHash	Super-Bit	Jaccard	Cosine	RKRGST
10	6	4	12	11	1324
20	5	5	13	12	1441
30	6	6	13	12	1354
40	7	6	14	13	1400
50	7	7	15	13	1339
60	7	7	15	14	1328
70	8	7	15	14	1337
80	8	8	16	15	1373
90	8	8	17	17	1384
100	11	10	17	16	1355
110	11	10	22	17	1444
120	10	11	20	18	1390
130	15	14	17	18	1448
140	15	15	19	19	1315
150	16	13	22	20	1253
160	13	13	19	21	1238
170	14	14	19	22	1200
180	15	14	20	25	1179
190	15	15	20	26	1170
200	16	16	21	21	1141

6.2. Efficiency of Locality-Sensitive Hashing (RQ1)

In terms of efficiency, similarity measurements with locality-sensitive hashing (MinHash and Super-Bit) are the fastest with five and seven seconds processing time respectively (see Table 3). For MinHash, it had a 60% decrease in processing time compared to Jaccard, its base measurement (13 to 5 s). Super-Bit also resulted in a comparable decrease when compared to its base measurement (Cosine); the processing time decreased from 18 to 7 s. Unlike Jaccard and Cosine, MinHash and Super-Bit do not consider the whole content of submissions for comparison as they only take some representative portion of it.

RKRGST, a common measurement for similarity detection is substantially slower than other similarity measurements (1441 s or about 24 min). Although RKRGST applies a hashing mechanism, it does not index the tokens before comparison. Further, it searches for the longest possible matches. These result in cubic time complexity while other similarity measurements only have linear time complexity.

Table 3. Performance of most effective MML for all scenarios.

Metric	MinHash	Super-Bit	Jaccard	Cosine	RKRGST
MML	20	60	30	130	20
F-score (%)	70%	81%	93%	84%	94%
Processing time (s)	5	7	13	18	1441

6.3. Effectiveness of Locality-Sensitive Hashing (RQ2)

Table 3 shows that MinHash and Super-Bit, two similarity measurements with locality-sensitive hashing, are the least effective compared to other measurements. This is expected as the offset of improved efficiency (shorter processing time).

Compared to its base measurement, MinHash's best f-score (70%) was 23% lower than Jaccard's (93%). This also happens on Super-Bit although the decrease was not substantial; Super-Bit's best f-score (81%) was only 3% lower than Cosine's (84%). Locality-sensitive hashing measurements only take a random portion of submissions' content for comparison and few of them might be less representative. Their mechanism to not consider the whole submissions' content is also the reason why both measurements have less consistent f-scores over increasing MMLs.

RKRGST results in the highest f-score (94%). However, it is comparable to Jaccard (93%) and its f-score difference to Super-Bit and Cosine was moderate (13% and 10%, respectively). RKRGST's greedy mechanism to search the largest possible matches can be somewhat replaced with our mechanism to merge matches on remaining similarity measurements.

6.4. Impact of Overlapping Adjacent Token Substrings in Building Indexes (RQ3)

Figure 6 depicts that overlapping adjacent token substrings positively affects effectiveness; it results in substantially higher f-score (more than 40%). The overlapping mechanism provides more flexibility in detecting matches. It considers any arbitrary position in the token strings as starting points, not only those divisible by MML.

The f-score increase becomes substantially higher on similarity measurements that consider occurrence frequencies (Cosine and Super-Bit). They rely more heavily on generated indexes than Jaccard and MinHash.

The overlapping mechanism slightly slows down the processing time (see Figure 7). This is expected as such a mechanism results in more token substrings to index and compare. For a string with X tokens, it considers $(X - n)$ substrings, while the non-overlapping one only considers (X/n) substrings. However, the increase in processing time is acceptable given that all similarity measurements are efficient by default (linear time complexity). Further, the overlapping mechanism substantially increases effectiveness.

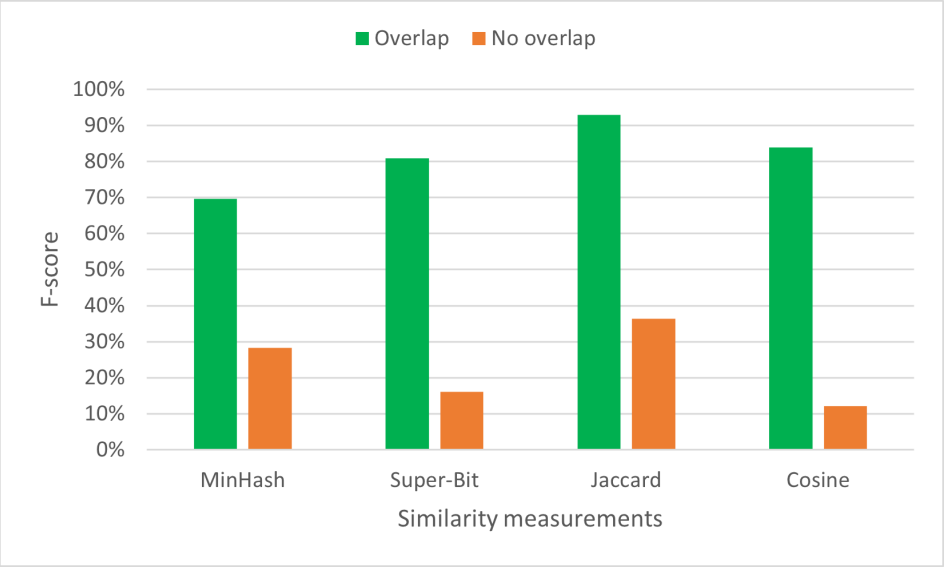

Figure 6. F-score (%) for both overlapping and non-overlapping similarity measurements.

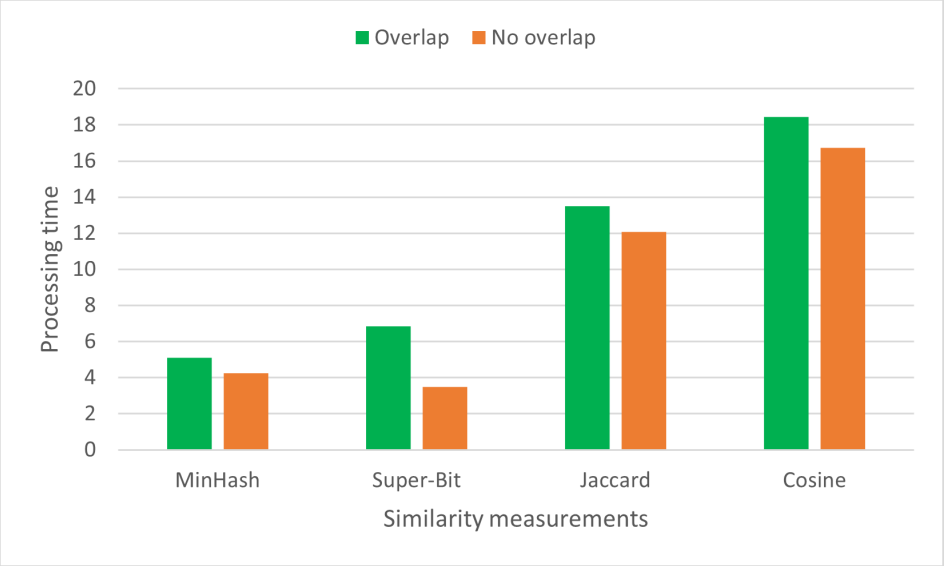

Figure 7. Processing time (s) for both overlapping and non-overlapping similarity measurements.

6.5. Impact of Number of Clusters in Locality-Sensitive Hashing (RQ3)

As shown in Figure 8, assigning different number of clusters can result in different f-score. Copied submissions and their originals might not fall in the same cluster as locality-sensitive hashing does not consider the whole content to cluster submissions. They only focus on seemingly representative ones. MinHash reaches its best f-score with seven clusters while Super-Bit reaches its best f-score with only two clusters.

Having more clusters does not necessarily slow down the processing time. Figure 9 shows that the processing time is quite stable across various numbers of clusters. Some iterations might perform slightly worse than others due to hardware and operating system dependencies. The processing time is relatively fast by default.

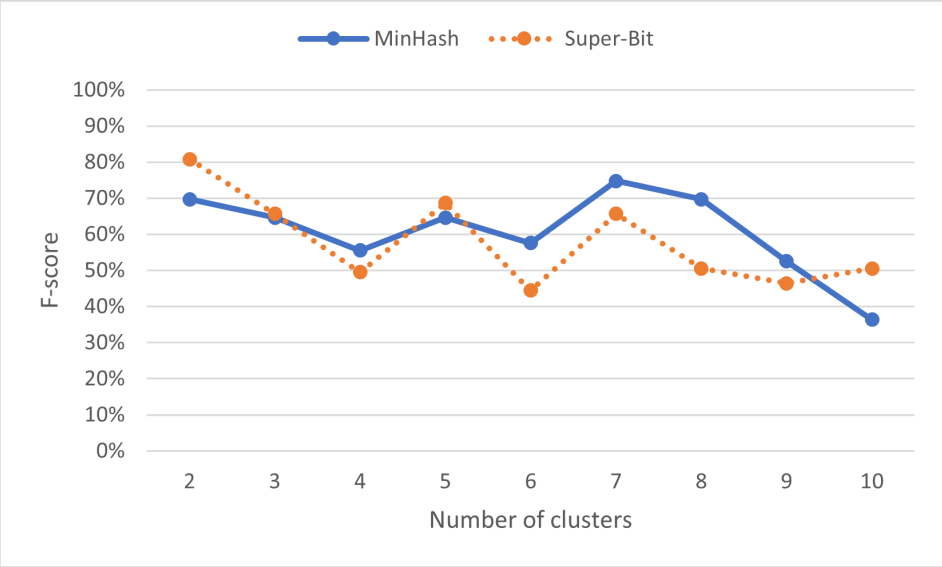

Figure 8. F-score (%) on various numbers of clusters in locality-sensitive hashing.

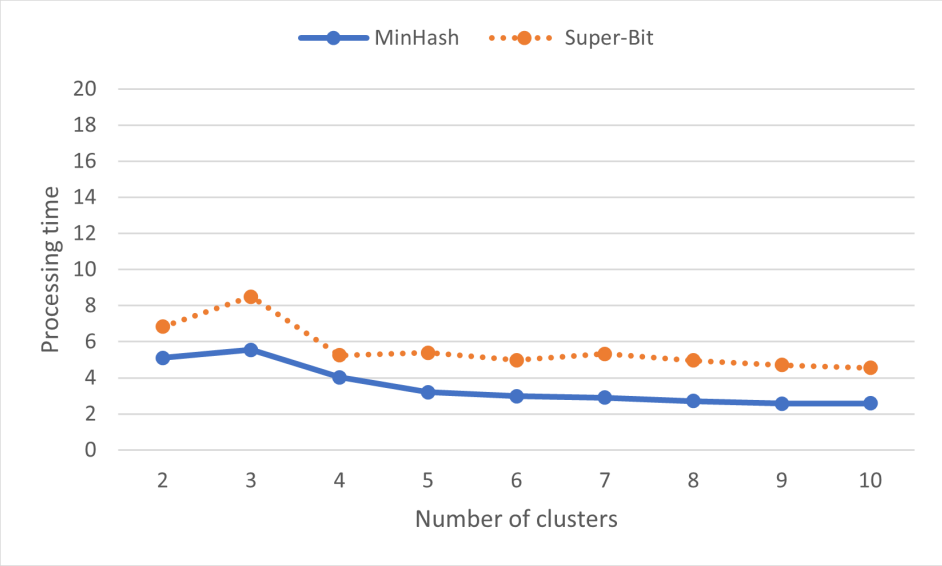

Figure 9. Processing time (s) on various numbers of clusters in locality-sensitive hashing.

6.6. Impact of Number of Stages in Locality-Sensitive Hashing (RQ3)

Having more stages means a higher chance of programs to be considered similar. On locality-sensitive hashing similarity measurements (MinHash and Super-Bit), programs are similar if they fall on the same cluster at least once. While this might be able to detect more copied programs, it can also result in more false positives: programs written independently but reported as suspicious. The most effective number of stages may vary across datasets. For the SOCO dataset, Figure 10 shows MinHash reaches its best f-score with nine as the number of stages, while Super-Bit reaches its best f-score when its number of stages is set to two or larger.

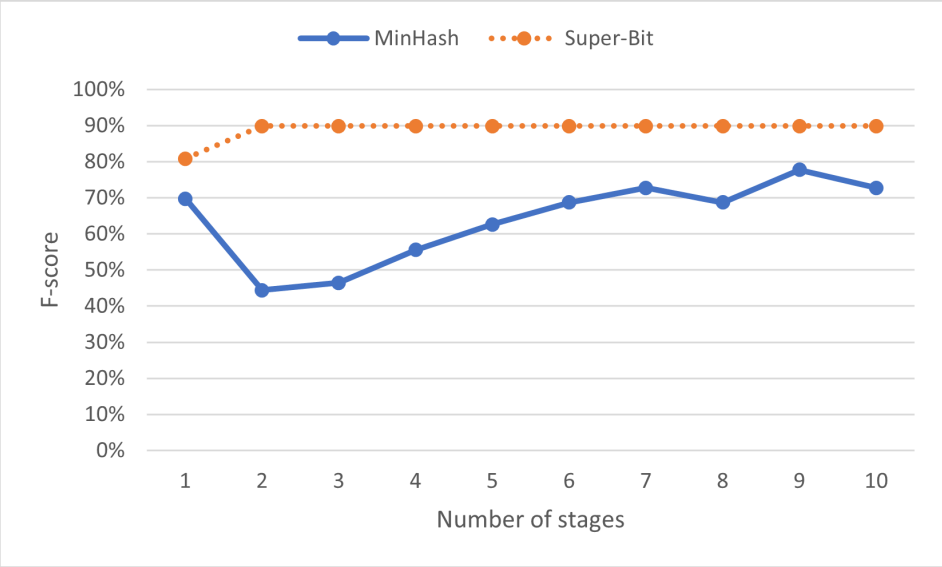

Figure 10. F-score (%) on various numbers of stages in locality-sensitive hashing.

The number of stages does not generally affect processing time (see Figure 11) perhaps since similarities on the dataset are relatively fast to compute. Variance might be a result of hardware and operating system dependency. It is interesting to see that assigning number of stages larger than one increases the processing time. Additional computation might be needed to introduce extra stages.

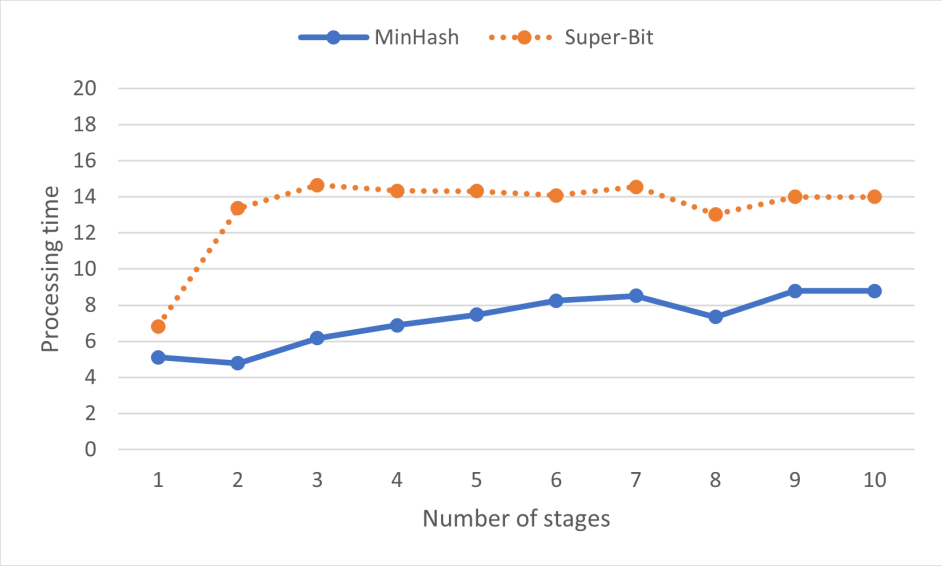

Figure 11. Processing time (s) on various numbers of stages in locality-sensitive hashing.

6.7. Experience and Recommendations on Employing Similarity Detectors (RQ4)

This section reports our instructors' experience on employing similarity detectors starting from early 2020 to mid 2022. Such a report is then converted to recommendations that can be useful for instructors interested in employing similarity detectors for the first time.

6.7.1. Experience from the First Semester of 2020

We first employed a similarity detector (STRANGE) in the first semester of 2020 with two courses for information technology (IT) undergraduates: basic and advanced data structure. Each course was taught by two instructors, one teaching the class session(s), while

another teaching the lab session(s). Basic data structure had two classes while advanced data structure had only one class. All four instructors found that the similarity detector helps them find copied programs. They were only expected to investigate programs suspected by the similarity detector instead of checking all programs.

Basic data structure's assessments are about translating and/or implementing algorithms to code. Such assessments typically resulted in solutions with similar program flow and high similarity alone was not sufficient for evidence. The instructors relied on irregular patterns found while investigating suspected cases. The patterns included similarities in comments, strange white space (e.g., unexpected tabs or spaces), unique identifier names, original author's student information, and unexpected program flow. While gathering evidence, the instructors needed to check whether similar code segments were legitimately copied from lecture materials or files provided by the instructors.

Advanced data structure's assessments were similar to those of basic data structure except that the solutions were in Java (at which the bridging language was built) and they used template code. High similarity alone was not sufficient and irregular patterns were often needed for evidence. The instructors specifically noted about two things. First, minimum matching length should be set relatively long so that each match was meaningful on its own. Java syntax tends to have many tokens, especially if compared to Python used in basic data structure. Second, although template code could be easily ignored while investigating suspected cases, it was convenient to have them automatically excluded from being reported. This inspired the development of template code removal in STRANGE and its successors.

6.7.2. Experience from the Second Semester of 2020

In the second semester of 2020, STRANGE was employed on two more courses: introductory programming for information system (IS) undergraduates (one instructor for both class and lab sessions) and IT advanced data structure (two instructors; one for class session and another for lab). Introductory programming had one class while advanced data structure had two classes. All instructors thought that STRANGE's similarity reports are relatively easy to understand.

For introductory programming, each assessment contained three tasks at with each of them resulted in one independent program. Hence, to limit the number of similarity reports, for each student, solutions of the three tasks were merged and treated as a single submission prior to comparison. For evidence itself, the instructor relied on irregular patterns of similarity, unexpected program flow in particular. For Python assessments, minimum matching length should be set lower than that of Java assessments. In this case, MML was set to ten for Python and 20 for Java. Both constants were assumed to cover two program statements. Given that many easy and medium assessment tasks expected similar solutions, there was a need to automatically select common code and exclude that prior to comparison. This feature was developed as part of CSTRANGE and SSTRANGE.

For advanced data structure, it was similar to the preceding semester, except that two assessments were issued weekly instead of one. The second weekly assessment covered the same tasks as the first but could be completed within a week instead of within two-hour lab session. Their experience was somewhat comparable to that offered in the preceding semester. One of the instructors noted that each week, students were unlikely to be suspected in only one assessment. They were either suspected in both assessments or not at all.

6.7.3. Experience from the First Semester of 2021

For the first semester of 2021, two courses employed STRANGE, the similarity detector. The first course was IS object-oriented programming. It had one class taught by one instructor for both of its class and lab sessions. The second course was introductory programming but unlike the one offered in the preceding semester, it is for IT undergraduates. Three instructors were involved in the teaching process: two for class sessions and two for lab

sessions. There were a total of three classes; one class instructor and one lab instructor taught two sessions each. Some instructors teaching this semester implied the need of a graphical user interface for the similarity detector. This inspired the development of such an interface on CSTRANGE and SSTRANGE.

The instructor of object-oriented programming noted that since the assessments defined the class structure, they expected many common code: attributes, constructors, getters, and setters. Finding strong evidence was quite complex although it was still manageable. Similar to many instructors previously employing the similarity detector, they relied on irregular patterns of similarity, especially strange ways of implementing methods. At the second half of the semester, assessments were focused on application development and finding strong evidence became more challenging. Such assessments typically expected long solutions and many of the reported similarities were code for generating application interface. To deal with this, the instructor focused on gathering evidence found in the main program and/or major methods.

Instructors of IT introductory programming had comparable experience to the instructor of IS introductory programming offered in the preceding semester. To have a reasonable number of similarity reports, all task solutions in one assessment were grouped based on the submitter and merged before being compared. Irregular patterns of similarity were often used as evidence. Given that such irregular patterns were quite common to use among instructors, there was a need to capture such patterns via more lenient similarity detection. This was implemented as another layer of similarity in CSTRANGE.

6.7.4. Experience from the Second Semester of 2021

In the second semester of 2021, four courses employed STRANGE: IS introductory programming, IS business application programming, IT data structure, and IT advanced object oriented programming. Each course was taught by one instructor except IT advanced object-oriented programming (two instructors, one for class sessions and another for lab sessions). All IS courses had one class each, while IT courses had two classes. Instructors of IS business application programming and IT advanced object-oriented programming found that STRANGE was quite slow for large assessments. This motivated the development of SSTRANGE, a scalable version of STRANGE.

For IS introductory programming, the instructor shared the same experience as previous offerings of such a course. Per assessment, solutions for all tasks were merged prior to comparison, limiting number of similarity reports. The instructor relied on irregular patterns of similarity for evidence, especially unexpected program flow. They specifically noted that fewer similarities were reported in homework assessments than the lab ones. Perhaps, it is because students had more time to complete homework assessments (one week instead of two hours), encouraging them to write the solutions by themselves. There was also a small possibility that few students might misuse given time to disguise their copying acts.

For IS business application programming, the assessments were about developing applications with database. The instructor found it difficult to gather evidence, as the expected solutions were typically long and some of the reported similarities are code for generating application interface. The issue was exacerbated by the fact that some code files and syntax other than the application interface code were also automatically generated (e.g., hibernate classes). Consequently, the instructor checked only the main program and a few major methods for evidence. Some assessments were about following detailed tutorial and the instructor decided not to check similarities on such assessments; all solutions were expected to have the same syntax, and even surface appearance.

IT data structure is a new course introduced in the new curriculum to replace both basic and advanced data structure. The assessments were intentionally given with general instructions (e.g., implement a data structure with a case study). This made detection of plagiarism and collusion much easier; programs with similar syntax is less likely to be a result of coincidence or following the instructions. The instructor used the tool for the

first time and at the beginning, they were uncertain why two programs having different superficial looks were considered as the same. They were made clear about the matter after a follow-up discussion.

For IT advanced object-oriented programming, the assessments were somewhat similar to those of IS business application programming. Consequently, the instructors checked evidence only in the main program and a few major methods.

6.7.5. Experience from the First Semester of 2022

At that time, CSTRANGE replaced STRANGE and it was employed on three courses. The first course was IS object-oriented programming (one class) taught by one instructor for both class and lab sessions. The second course was IT introductory programming (two classes), taught by two instructors: one managing class sessions, while another managing lab sessions. The third course was machine intelligence (two classes) at which the class sessions were taught by an instructor and the lab sessions were taught by another instructor.

For IS object-oriented programming, the assessments were somewhat comparable to those issued in the previous offering of the course. CSTRANGE was arguably helpful for the instructor in gathering evidence. They could rely on three levels of similarity instead of one. However, as CSTRANGE reported more information than STRANGE, it was slower, especially on assessments about application development.

For IT introductory programming, the instructors found that having superficial similarities reported helped them in gathering evidence. Some of the assessment tasks were trivial and they expected the same solutions even at syntax level. One instructor noted that the similarity detector could not detect all copied programs. They found some copied programs that were not reported by the tool. This is expected as a trade-off using a similarity detector for time efficiency: instructors are not required to check all programs but the automation does not guarantee to detect all copied programs [8]. The instructor also found that few students were suspected because they only finished easy tasks (which solutions were expected to be highly similar).

For IT machine intelligence, the assessments had detailed instructions and those relatively limited variation across the solutions. Further, students were only taught a few ways of using third-party libraries. The instructors expected the solutions to be similar. They relied on irregular patterns of similarity at which some of them were reported by CSTRANGE.

6.8. Recommendations

Based on our instructors' experience, we have a number of recommendations (sorted by the importance):

1. **While choosing a similarity detector, it is recommended to consider the assessment design.** An effective similarity detector is not always the one that nullifies many code variations. Some variations need to be kept to prevent coincidental similarities from being reported, especially on assessments with small tasks, trivial solutions and/or detailed instructions [10,44]. Many of our assessments, for example, expect similar program flow in the solutions. Common similarity detectors, such as JPlag [13], might report most submissions to be similar as they report program flow similarity.
2. **Instructors need to generally understand about how the employed similarity detector works.** Some instructors can be uncertain about why two programs are considered similar despite having different superficial looks. They can also get confused when a number of copied programs are not reported by the tool. Instructors should read the tool's documentation. Otherwise, they can be briefed with other instructors who have used the tool before.
3. **If possible, assessments need to have space for students' creativity and/or space for using their own case studies.** Detection of plagiarism and collusion can be easier as reported similarities are less likely to be a result of coincidence [58].

4. **Strong evidence can be obtained by observing irregular patterns of similarity.** These include similarities in comments, strange white space, unique identifier names, original author's student information, and unexpected program flow. Many students conduct plagiarism and collusion because they are not fluent in programming [59]. They lack skill to disguise copied programs with unique characteristics.
5. **Template and common code should not become evidence for raising suspicion.** They are expected among submissions. For each similar code segment, it is important to check whether it is part of template code: whether it is legitimately copiable code (from lecture slides or assessment documents); whether it is compulsorily used for compilation; whether it is suggested by instructors; and whether it is the most obvious implementation for the task [11]. Several similarity detectors offer automatic removal of such code but the result is not always accurate (for the purpose of efficiency). JPlag [13], MOSS [5], and CSTRANGE [49] are three examples of similarity detectors that offer template code removal. The last two also cater for common code removal.
6. **Evidence should be based on manually written code.** Some assessments (e.g., those developing applications) encourage the use of automatically generated code and searching evidence on such code is labor intensive. It is preferred to focus on code parts that are manually written such as the main program or major methods.
7. **To have a reasonable number of similarity reports, several short programs in one assessment can be merged before comparison.** More programs tend to result in more similarity reports. The number of similarity reports needs to be reasonable given that each of the reports will be manually investigated.
8. **Assessments expecting the same solutions even at superficial level (e.g., following a detailed tutorial) should be ignored for similarity detection.** Searching for strong evidence in such assessments is labor intensive. Instructors are expected not to allocate a large portion of marks for such assessments.
9. **Investigation can be more comprehensive via the consideration of several similarity measurements.** Each similarity measurement has its own benefits and weaknesses. Having more than one similarity measurement can lead in more objective investigation. Sherlock [8] and CSTRANGE [49] are two examples of tools facilitating multiple measurements in the similarity reports.
10. **For assessments expecting many large submissions, the employed similarity detector should be efficient, preferably the one that works in linear time.** Some instructors do not have much time to run the detector and wait for the similarity reports. The issue is exacerbated on similarity detectors with non-linear time complexity as the processing time becomes much longer.

7. Limitations

Our study has four limitations:

1. Although the SOCO dataset is common for evaluation, SSTRANGE was evaluated only on a single dataset. Replicating the evaluation with different datasets can enrich the findings.
2. SSTRANGE's evaluation only considers f-score for effectiveness and processing time for efficiency. While these metrics are relatively common, utilizing other performance metrics might be useful for further observation.
3. While some recommendations for instructors interested in employing similarity detectors appear to be more aligned in justifying the need to improve SSTRANGE's predecessors, we believe they are somewhat applicable for general use of similarity detectors. Our 13 course offerings have various assessment designs, and they are quite representative for programming.
4. Recommendations for instructors interested in employing similarity detectors are based on instructors' experience in a single institution in a particular country. Some of them might be in need for stronger evidence. More comprehensive and well-

supported findings might be obtained by reporting such experience from more institutions, especially those located in different countries.

8. Conclusions and Future Work

This paper presents two contributions: SSTRANGE and our recommendations for employing similarity detectors. Unique to SSTRANGE, it is featured with locality-sensitive hashing, intuitive reports for investigation, and a graphical user interface. The tool, the code, and the documentation are publicly accessible. According to our evaluation, SSTRANGE's locality-sensitive hashing similarity measurements are more efficient than common similarity measurements while having reasonable offset on effectiveness. Best effectiveness performance can be obtained by tuning minimum matching length, number of clusters, and number of stages. There is also a need to rely on overlapping adjacent token substrings in building indexes.

Based on our 2.5 year experience employing SSTRANGE's predecessors, we recommend instructors to understand how a similarity detector works and its implications on the assessment design. While gathering evidence, it is necessary to focus on irregular patterns of similarity (e.g., similar comments and unexpected program flow), which can be more evident on assessments incorporating students' creativity and/or case studies. Template code, common code, and automatically generated code should be ignored. We also recommend instructors to merge short programs as a single submission and not to detect similarity on assessments expecting a similar superficial look. Consideration of multiple similarity measurements can lead to more objective investigation but if there are many large submissions, it is preferred to employ a similarity detector with linear time complexity for efficiency.

For future work, we plan to evaluate SSTRANGE on other datasets (preferably those from our faculty) with more performance metrics. We are also interested to record instructors' experience on employing the tool in their courses. Last but not least, we are interested in conducting focus group discussions about our recommendations for employing similarity detectors with instructors who have employed similarity detectors from other institutions. The discussions are expected to strengthen our recommendations.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Informed consent was obtained from all subjects involved in the study.

Data Availability Statement: The SOCO dataset was provided by Ragkhitwetsagul et al. [57] (https://github.com/UCL-CREST/ocd/blob/master/soco_f.zip accessed on 2 November 2022). SSTRANGE is provided on GitHub (https://github.com/oscarkarnalim/SSTRANGE accessed on 2 November 2022). Other data generated and/or analyzed during the current study are available from the corresponding author on reasonable request.

Acknowledgments: To Maranatha Christian University, Indonesia and their instructors for facilitating the study. To guest editors (Mike Joy from University of Warwick, UK and Peter Williams from University of Hull, UK) for the invitation to participate in this special issue and the fee waiver.

Conflicts of Interest: The author declares no conflict of interest.

References

1. Fraser, R. Collaboration, collusion and plagiarism in computer science coursework. *Inform. Educ.* **2014**, *13*, 179–195. [CrossRef]
2. Lancaster, T. Academic integrity for computer science instructors. In *Higher Education Computer Science*; Springer Nature: Cham, Switzerland, 2018; pp. 59–71. [CrossRef]
3. Simon; Sheard, J.; Morgan, M.; Petersen, A.; Settle, A.; Sinclair, J. Informing students about academic integrity in programming. In Proceedings of the 20th Australasian Computing Education Conference, Brisbane, QLD, Australia, 30 January–2 February 2018; pp. 113–122. [CrossRef]

4. Kustanto, C.; Liem, I. Automatic source code plagiarism detection. In Proceedings of the 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing, Daegu, Republic of Korea, 27–29 May 2009; pp. 481–486. [CrossRef]
5. Schleimer, S.; Wilkerson, D.S.; Aiken, A. Winnowing: Local algorithms for document fingerprinting. In Proceedings of the International Conference on Management of Data, San Diego, CA, USA, 9–12 June 2003; pp. 76–85. [CrossRef]
6. Novak, M.; Joy, M.; Kermek, D. Source-code similarity detection and detection tools used in academia: A systematic review. *ACM Trans. Comput. Educ.* **2019**, *19*, 27:1–27:37. [CrossRef]
7. Karnalim, O.; Simon.; Chivers, W. Similarity detection techniques for academic source code plagiarism and collusion: A review. In Proceedings of the IEEE International Conference on Engineering, Technology and Education, Yogyakarta, Indonesia, 10–13 December 2019. [CrossRef]
8. Joy, M.; Luck, M. Plagiarism in programming assignments. *IEEE Trans. Educ.* **1999**, *42*, 129–133. [CrossRef]
9. Ahadi, A.; Mathieson, L. A comparison of three popular source code similarity tools for detecting student plagiarism. In Proceedings of the 21st Australasian Computing Education Conference, Sydney, NSW, Australia, 29–31 January 2019; pp. 112–117. [CrossRef]
10. Karnalim, O.; Simon.; Ayub, M.; Kurniawati, G.; Nathasya, R.A.; Wijanto, M.C. Work-in-progress: Syntactic code similarity detection in strongly directed assessments. In Proceedings of the IEEE Global Engineering Education Conference, Vienna, Austria, 21–23 April 2021; pp. 1162–1166. [CrossRef]
11. Simon.; Karnalim, O.; Sheard, J.; Dema, I.; Karkare, A.; Leinonen, J.; Liut, M.; McCauley, R. Choosing Code Segments to Exclude from Code Similarity Detection. In Proceedings of the ACM Working Group Reports on Innovation and Technology in Computer Science Education, Trondheim, Norway, 15–19 June 2020; pp. 1–19. [CrossRef]
12. Chi, L.; Zhu, X. Hashing techniques: A survey and taxonomy. *ACM Comput. Surv.* **2018**, *50*, 11. [CrossRef]
13. Prechelt, L.; Malpohl, G.; Philippsen, M. Finding plagiarisms among a set of programs with JPlag. *J. Univers. Comput. Sci.* **2002**, *8*, 1016–1038.
14. Sulistiani, L.; Karnalim, O. ES-Plag: Efficient and sensitive source code plagiarism detection tool for academic environment. *Comput. Appl. Eng. Educ.* **2019**, *27*, 166–182. [CrossRef]
15. Karnalim, O.; Simon. Explanation in code similarity investigation. *IEEE Access* **2021**, *9*, 59935–59948. [CrossRef]
16. Ferdows, J.; Khatun, S.; Mokarrama, M.J.; Arefin, M.S. A framework for checking plagiarized contents in programs submitted at online judging systems. In Proceedings of the International Conference on Image Processing and Capsule Networks, Bangkok, Thailand, 6–7 May 2020; pp. 546–560. [CrossRef]
17. Durić, Z.; Gašević, D. A source code similarity system for plagiarism detection. *Comput. J.* **2013**, *56*, 70–86. [CrossRef]
18. Ahtiainen, A.; Surakka, S.; Rahikainen, M. Plaggie: GNU-licensed source code plagiarism detection engine for Java exercises. In Proceedings of the Sixth Baltic Sea Conference on Computing Education Research, Koli Calling Uppsala, Sweden, 1 February 2006; pp. 141–142. [CrossRef]
19. Acampora, G.; Cosma, G. A fuzzy-based approach to programming language independent source-code plagiarism detection. In Proceedings of the IEEE International Conference on Fuzzy Systems, Istanbul, Turkey, 2–5 August 2015; pp. 1–8. [CrossRef]
20. Domin, C.; Pohl, H.; Krause, M. Improving plagiarism detection in coding assignments by dynamic removal of common ground. In Proceedings of the CHI Conference Extended Abstracts on Human Factors in Computing Systems, San Jose, CA, USA, 7–12 May 2016; pp. 1173–1179. [CrossRef]
21. Ji, J.H.; Woo, G.; Cho, H.G. A plagiarism detection technique for Java program using bytecode analysis. In Proceedings of the Third International Conference on Convergence and Hybrid Information Technology, Busan, Korea, 11–13 November 2008; pp. 1092–1098. [CrossRef]
22. Jiang, Y.; Xu, C. Needle: Detecting code plagiarism on student submissions. In Proceedings of the ACM Turing Celebration Conference, Shanghai, China, 19–20 May 2018; pp. 27–32. [CrossRef]
23. Fu, D.; Xu, Y.; Yu, H.; Yang, B. WASTK: A weighted abstract syntax tree kernel method for source code plagiarism detection. *Sci. Program.* **2017**, *2017*, 1–8. [CrossRef]
24. Nichols, L.; Dewey, K.; Emre, M.; Chen, S.; Hardekopf, B. Syntax-based improvements to plagiarism detectors and their evaluations. In Proceedings of the 24th ACM Conference on Innovation and Technology in Computer Science Education, Aberdeen, UK, 15–17 July 2019; pp. 555–561. [CrossRef]
25. Kikuchi, H.; Goto, T.; Wakatsuki, M.; Nishino, T. A source code plagiarism detecting method using alignment with abstract syntax tree elements. In Proceedings of the 15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, Las Vegas, NV, USA, 30 June–2 July 2014; pp. 1–6. [CrossRef]
26. Cheers, H.; Lin, Y.; Smith, S.P. Academic source code plagiarism detection by measuring program behavioral similarity. *IEEE Access* **2021**, *9*, 50391–50412. [CrossRef]
27. Song, H.J.; Park, S.B.; Park, S.Y. Computation of program source code similarity by composition of parse tree and call graph. *Math. Probl. Eng.* **2015**, *2015*, 1–12. [CrossRef]
28. Ottenstein, K.J. An algorithmic approach to the detection and prevention of plagiarism. *ACM SIGCSE Bull.* **1976**, *8*, 30–41. [CrossRef]

29. Donaldson, John L. and Lancaster, Ann-Marie and Sposato, Paula H.. A plagiarism detection system. In Proceedings of the 12th SIGCSE Technical Symposium on Computer Science Education, St. Louis, MO, USA, 26–27 February 1981; Volume 13, pp. 21–25. [CrossRef]
30. Al-Khanjari, Z.A.; Fiaidhi, J.A.; Al-Hinai, R.A.; Kutti, N.S. PlagDetect: A Java programming plagiarism detection tool. *ACM Inroads* **2010**, *1*, 66–71. [CrossRef]
31. Allyson, F.B.; Danilo, M.L.; José, S.M.; Giovanni, B.C. Sherlock N-Overlap: Invasive normalization and overlap coefficient for the similarity analysis between source code. *IEEE Trans. Comput.* **2019**, *68*, 740–751. [CrossRef]
32. Inoue, U.; Wada, S. Detecting plagiarisms in elementary programming courses. In Proceedings of the Ninth International Conference on Fuzzy Systems and Knowledge Discovery, Chongqing, China, 29–31 May 2012; pp. 2308–2312. [CrossRef]
33. Ullah, F.; Wang, J.; Farhan, M.; Habib, M.; Khalid, S. Software plagiarism detection in multiprogramming languages using machine learning approach. *Concurr. Comput. Pract. Exp.* **2018**, *33*, e5000. [CrossRef]
34. Yasaswi, J.; Purini, S.; Jawahar, C.V. Plagiarism detection in programming assignments using deep features. In Proceedings of the Fourth IAPR Asian Conference on Pattern Recognition, Nanjing, China, 26–29 November 2017; pp. 652–657. [CrossRef]
35. Elenbogen, B.S.; Seliya, N. Detecting outsourced student programming assignments. *J. Comput. Sci. Coll.* **2008**, *23*, 50–57.
36. Jadalla, A.; Elnagar, A. PDE4Java: Plagiarism detection engine for Java source code: A clustering approach. *Int. J. Bus. Intell. Data Min.* **2008**, *3*, 121. [CrossRef]
37. Moussiades, L.; Vakali, A. PDetect: A clustering approach for detecting plagiarism in source code datasets. *Comput. J.* **2005**, *48*, 651–661. [CrossRef]
38. Flores, E.; Barrón-Cedeño, A.; Moreno, L.; Rosso, P. Uncovering source code reuse in large-scale academic environments. *Comput. Appl. Eng. Educ.* **2015**, *23*, 383–390. [CrossRef]
39. Foltýnek, T.; Všianský, R.; Meuschke, N.; Dlabolová, D.; Gipp, B. Cross-language source code plagiarism detection using explicit semantic analysis and scored greedy string tilling. In Proceedings of the ACM/IEEE Joint Conference on Digital Libraries in 2020, Virtual Event, China, 1–5 August 2020; pp. 523–524. [CrossRef]
40. Ullah, F.; Jabbar, S.; Mostarda, L. An intelligent decision support system for software plagiarism detection in academia. *Int. J. Intell. Syst.* **2021**, *36*, 2730–2752. [CrossRef]
41. Cosma, G.; Joy, M. An approach to source-code plagiarism detection and investigation using latent semantic analysis. *IEEE Trans. Comput.* **2012**, *61*, 379–394. [CrossRef]
42. Ganguly, D.; Jones, G.J.F.; Ramírez-de-la Cruz, A.; Ramírez-de-la Rosa, G.; Villatoro-Tello, E. Retrieving and classifying instances of source code plagiarism. *Inf. Retr. J.* **2018**, *21*, 1–23. [CrossRef]
43. Mozgovoy, M.; Karakovskiy, S.; Klyuev, V. Fast and reliable plagiarism detection system. In Proceedings of the 37th IEEE Annual Frontiers in Education Conference, Milwaukee, Wisconsin, 10–13 October 2007; pp. 11–14. [CrossRef]
44. Mann, S.; Frew, Z. Similarity and originality in code: Plagiarism and normal variation in student assignments. In Proceedings of the Eighth Australasian Computing Education Conference, Hobart, Australia, 16–19 January 2006; pp. 143–150.
45. Bowyer, K.W.; Hall, L.O. Experience using "MOSS" to detect cheating on programming assignments. In Proceedings of the 29th Annual Frontiers in Education Conference, San Juan, Puerto Rico, 10–13 November 1999. [CrossRef]
46. Pawelczak, D. Benefits and drawbacks of source code plagiarism detection in engineering education. In Proceedings of the IEEE Global Engineering Education Conference, Santa Cruz de Tenerife, Spain, 17–20 April 2018; pp. 1048–1056. [CrossRef]
47. Le Nguyen, T.T.; Carbone, A.; Sheard, J.; Schuhmacher, M. Integrating source code plagiarism into a virtual learning environment: Benefits for students and staff. In Proceedings of the 15th Australasian Computing Education Conference, Adelaide, Australia, 29 January–1 February 2013; pp. 155–164. [CrossRef]
48. Karnalim, O. Detecting source code plagiarism on introductory programming course assignments using a bytecode approach. In Proceedings of the 10th International Conference on Information & Communication Technology and Systems, Lisbon, Portugal, 6–9 September 2016; pp. 63–68. [CrossRef]
49. Karnalim, O.; Simon.; Chivers, W. Layered similarity detection for programming plagiarism and collusion on weekly assessments. *Comput. Appl. Eng. Educ.* **2022**, *30*, 1739–1752. [CrossRef]
50. Parr, T. *The Definitive ANTLR 4 Reference*; The Pragmatic Programmers: Raleigh, NC, USA, 2013.
51. Sraka, D.; Kaučič, B. Source code plagiarism. In Proceedings of the 31st International Conference on Information Technology Interfaces, Dubrovnik, Croatia, 22–25 June 2009, pp. 461–466. [CrossRef]
52. Croft, W.B.; Metzler, D.; Strohman, T. *Search Engines: Information Retrieval in Practice*; Pearson Education: London, UK, 2010.
53. Broder, A. On the resemblance and containment of documents. In Proceedings of the Compression and Complexity of SEQUENCES 1997, Salerno, Italy, 11–13 June 1997; pp. 21–29. [CrossRef]
54. Ji, J.; Li, J.; Yan, S.; Zhang, B.; Tian, Q. Super-Bit locality-sensitive hashing. In Proceedings of the 25th International Conference on Neural Information Processing Systems, Lake Tahoe, NV, USA, 3–6 December 2012; Curran Associates Inc.: Red Hook, NY, USA, 2012; pp. 108–116.
55. Wise, M.J. YAP3: Improved detection of similarities in computer program and other texts. In Proceedings of the 27th SIGCSE Technical Symposium on Computer Science Education, Philadelphia, Pennsylvania, USA, 15–17 February 1996; pp. 130–134. [CrossRef]
56. Arwin, C.; Tahaghoghi, S.M.M. Plagiarism detection across programming languages. In Proceedings of the Sixth Australasian Computer Science Conference, Reading, UK, 28–31 May 2006; pp. 277–286.

57. Ragkhitwetsagul, C.; Krinke, J.; Clark, D. A comparison of code similarity analysers. *Empir. Softw. Eng.* **2018**, *23*, 2464–2519. [CrossRef]
58. Bradley, S. Creative assessment in programming: Diversity and divergence. In Proceedings of the Fourth Conference on Computing Education Practice, Durham, UK, 9 January 2020; pp. 13:1–13:4. [CrossRef]
59. Vogts, D. Plagiarising of source code by novice programmers a "cry for help"? In Proceedings of the Research Conference of the South African Institute of Computer Scientists and Information Technologists, Vanderbijlpark, Emfuleni, South Africa, 12–14 October 2009; pp. 141–149. [CrossRef]