A Scalable Code Similarity Detection with Online Architecture and Focused Comparison for Maintaining Academic Integrity in Programming

https://doi.org/10.3991/ijoe.v16i10.14289

Ricardo Franclinton, Oscar Karnalim, Mewati Ayub (✉)
Maranatha Christian University, Bandung, Indonesia
mewati.ayub@it.maranatha.edu

Abstract—Many code similarity detection techniques have been developed to maintain academic integrity in programming. However, most of them assume that the student programs are locally available, and the computation can be run on any computer specification. Further, their comparison in raising suspicion is time consuming as the student programs are pairwise compared one another. This paper proposes a scalable code similarity detection with online architecture and focused comparison. The former enables student programs shared among lecturers and guarantees that the computation is runnable. The latter shorten the execution time as only some students are considered, with inclusion criteria determined by the lecturers. To boost up the scalability, the similarity algorithm is cosine correlation, which computation is linear time. Our evaluation shows that focused comparison leads to fewer comparisons and cosine correlation leads to shorter execution time.

Keywords—Plagiarism detection, scalability, academic integrity, programming, computing education.

1 Introduction

Maintaining academic integrity is a serious concern in engineering education [1], [2], especially with the introduction of MOOC [3], [4]. Several strategies have been proposed in which one of the most popular ones is the use of Turnitin [5]. However, only few of them are applicable for programming courses [6], even though these courses are common in many engineering major curriculum. A possible reason behind this is the differences between standard text and source code [6].

In general, strategies for maintaining academic integrity in programming can be classified to five categories [7]. Educating the students about that kind of integrity is probably the most obvious one. This is usually carried out at the beginning of the course, with a lecturer or tutor explaining the acceptable practices [8]. Cheating can also be mitigated by discouraging such a behavior (e.g., incorporating additional assessment measures [2]), reducing the benefits of cheating (e.g., lowering the score of each assessment, making it not worthy to cheat), or putting more assessment restrictions (e.g., individualising the assessments [2]). It is also possible motivate the students to avoid cheating (e.g., the use of peer-assisted student support to keep the student retention high).

Source code plagiarism is a common form of breaching academic integrity in programming. It is about the reuse of source code without appropriate acknowledgment toward the original [9]. This kind of cheating is often discouraged by penalizing the perpetrators with the help of an automated detection tool like JPlag [10]. Per assessment, the student programs are pairwise compared one another and pairs with high similarity will be investigated further. If any of the pairs are confirmed to breach academic integrity, the students will be penalized according to the course's policy.

Many automated similarity detection techniques have been developed, in which the details can be seen in two studies [11], [32]. These techniques can be classified to attribute-counting-based, structure-based, and hybrid. The first category is faster than the second one (due to its less sensitive matching constraints) but tends to have lower effectiveness. Hybrid techniques combine those categories in search of the most balanced effectiveness-efficiency trade-off.

Among those three categories, attribute-counting-based detection techniques are believed to be the most scalable due to its fast computation. One of the earliest techniques of this kind was proposed in 1976 [12], relying on four software metrics to define the similarity. This inspired the introduction of other early techniques [13]–[15] with more similarity metrics on board.

Knowing these similarity metrics can be superficial, some attribute-counting-based techniques rely on source code content in determining the similarity. The content is split to shorter strings called n-grams [16] in which each string represents n adjacent tokens. The similarity algorithm itself can vary, but most of them are from information retrieval. Some of the applied ones are cosine correlation [17], overlap coefficient [18], latent semantic analysis [19], and code specific BM25 [20].

Most similarity detection techniques (including the popular ones such as JPlag [10], Plaggie [21], and Sherlock [22]) require the student programs to be locally available. This can be problematic if many classes with different lecturers are involved as the student programs should be shared among themselves manually. It also inhibits the comparison of student programs across assessments, courses from the same cohort, and/or courses from the previous cohorts.

These techniques also assume that the computation can be performed locally in the lecturer's personal computer. Not all personal computers are high-end and capable for such computation. This can be worse if enormous student programs are involved, assuming they are not only taken from a particular assessment for a particular class.

Lastly, these techniques pairwise compare all student programs even though only some of them are likely to cheat due to the motivating factors [23]. It leads to more computation and can slow down the execution time.

In response to the aforementioned gaps, this paper proposes a scalable code similarity detection with online architecture and focused comparison. The architecture enables lecturers to easily share their student programs among themselves. They just need to upload the student programs to the server, and these programs will be automatically stored for future comparisons by any lecturers. It also assures that the computation can be performed regardless the lecturer's personal computer's specification since that computation will be carried out by the server. The focused comparison can shorten the execution time as not all student programs will be compared, with the inclusion criteria defined by the lecturers. For scalability, cosine correlation from information retrieval is used to measure the similarities.

2 Methodology

Four stages are required in using our detection: student program collection, perpetrator candidate selection, plagiarism detection, and in-depth discussion. It accepts either Java or Python student programs as the input.

Student program collection means that all participating lecturers should upload their own student programs to the server. At this stage, student programs from previous courses can also be uploaded if needed. For weekly assessments with different class schedules, an agreement can be made among lecturers to upload the student programs no later than a particular day.

Perpetrator candidate selection is performed manually by each lecturer. Per class, a set of students is selected based on the lecturer's suspicion. These students can be those who lack of programming skill, seldom attend the classes, or have previously breached academic integrity. For objectivity, such criteria can be discussed among participating lecturers at the beginning of the course.

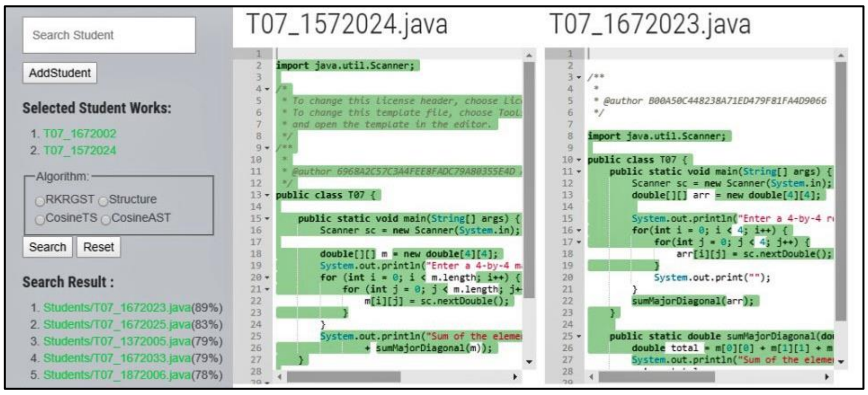
Plagiarism detection is carried out separately per class. The perpetrator candidates' programs are given as queries to the detection technique, and per query, any similar student programs will be retrieved in descending order based on their similarity degree. Fig. 1 shows our detection technique's layout for this stage in which 'selected student programs' are the perpetrator candidates' (selected via search box above) and 'search result' lists any similar student programs for a particular query (selected by clicking that query from 'selected student programs'). To avoid over information, only five search results are given per query.

Search result per query is determined by comparing the query to all student programs uploaded in the server (except the query itself). Compared to other detection techniques that pairwise compare all possible combinations, this is more time efficient due to its linear computation.

The comparison itself (referred as *CosineTS*) is performed in twofold. At first, the student programs are converted to token strings with the help of ANTLR [24], in which comments and whitespaces are removed as they are easy to disguise. After that, the query's string will be compared to each student program's string with cosine correlation, a similarity measurement adapted from information retrieval [16]. Compared to string matching algorithms used in many detection techniques, this is also more time efficient thanks to its linear time complexity [17]. Suspected student pairs are formed by pairing each query and one of its search results.

In addition to *CosineTS*, three other comparison modes are provided. *RKRGST* converts the student programs to token strings and then measure the similarities via running Karp-Rabin greedy string tiling [25], a common string-matching algorithm for code similarity detection techniques [32]. *Structure* works similarly but the token strings are the result of linearising the syntax trees in a pre-order manner, inspired from two former studies [26], [27]. These tokens are expected to be more resistant to surface modification as most of them cannot be modified directly at source code level. *CosineAST* is similar to *Structure* except that the similarity measurement is cosine correlation instead of running Karp-Rabin greedy string tiling. This is actually a simplified version of a technique proposed in [17], expecting to be less time consuming.

Several studies [28], [29] state that high similarity does not necessarily entail plagiarism. Hence, there is a need to revalidate similarities in the suspected pairs, whether they are likely from plagiarism [30]. Our detection technique supports this investigation for each pair by showing the code content of the student programs side-by-side as seen in Fig. 1. For convenience, similar fragments are highlighted in green. This layout is remodeled from JPlag [10] with the help of Plago [31]. If *Structure* or *CosineAST* is used as the similarity algorithm, the layout will include syntax tree tokens and show the code contents as two lists of tokens (see Fig. 2). Similar to the standard layout, the similarities are highlighted in green.

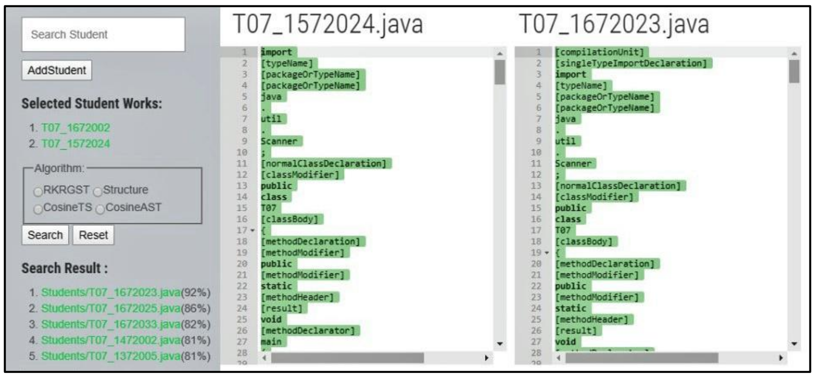


```

import java.util.Scanner;

/*
 * To change this license header, choose Lic
 * To change this template file, choose Tool
 * and open the template in the editor.
 */
/**
 *
 * @author 6968A2C57C3A4FEE8FADC79A80355E4D
 */
public class T07 {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        double[][] m = new double[4][4];
        System.out.println("Enter a 4-by-4 m
        for (int i = 0; i < m.length; i++) {
            for (int j = 0; j < m.length; j+
                m[i][j] = sc.nextDouble();
            }
        }
        System.out.println("Sum of the eleme
                + sumMajorDiagonal(m));
    }

```

```


/**
 *
 * @author B00A50C448238A71ED479F81FA4D9066
 */

import java.util.Scanner;

public class T07 {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        double[][] arr = new double[4][4];

        System.out.println("Enter a 4-by-4 r
        for(int i = 0; i < 4; i++) {
            for(int j = 0; j < 4; j++) {
                arr[i][j] = sc.nextDouble();
            }
            System.out.print("");
        }
        sumMajorDiagonal(arr);
    }

    public static double sumMajorDiagonal(dou
        double total = m[0][0] + m[1][1] + m
        System.out.println("Sum of the eleme
```

Fig. 1. The layout for plagiarism detection stage.

After the suspected pairs of each class have been revalidated, an in-depth discussion should be conducted, assuring that no independent programs are listed in the suspected pairs. Some students may feel discouraged if they are wrongly accused. It is advised that the discussion involves former lecturers or tutors of the suspected students. If the work seems to be copied from another class or course, the lecturer from that class or course should also be invited. At the end of this stage, suspected students have been selected and they will be penalized according to the course's policy.


Search Student
AddStudent
Selected Student Works:
1. T07_1672002
2. T07_1572024
Algorithm:
RKRGST
Structure
CosineTS
CosineAST
Search
Reset
Search Result :
1. Students/T07_1672023.java(92%)
2. Students/T07_1672025.java(86%)
3. Students/T07_1672033.java(82%)
4. Students/T07_1472002.java(81%)
5. Students/T07_1372005.java(81%)

T07_1572024.java

```
import
[typeName]
[packageOrTypeName]
[packageOrTypeName]
java
.
util
.
Scanner
;
[normalClassDeclaration]
[classModifier]
public
class
T07
[classBody]
{
[methodDeclaration]
[methodModifier]
public
[methodModifier]
static
[methodHeader]
[result]
void
[methodDeclarator]
main
```

T07_1672023.java

```
[compilationUnit]
[singleTypeImportDeclaration]
import
[typeName]
[packageOrTypeName]
[packageOrTypeName]
java
.
util
.
Scanner
;
[normalClassDeclaration]
[classModifier]
public
class
T07
[classBody]
{
[methodDeclaration]
[methodModifier]
public
[methodModifier]
static
[methodHeader]
[result]
void
```

Fig. 2. Investigation layout for *Structure* and *CosineAST*.

3 Evaluation and Discussion

This section evaluates the impact of focused comparison, the impact of cosine correlation and the impact of our proposed comparison modes (*CosineTS*, *RKRGST*, *CosineAST*, and *Structure*).

3.1 The impact of focused comparison

Focused comparison is expected to be more time efficient as not all student programs are compared. To prove this, the comparison was compared with the naïve one (which exhaustively include all possible comparison pairs) for ten different numbers of student programs, starting from 0 to 100 with 10 for each adjacent difference. The focused comparison was featured with 10% number of student programs as the queries.

Fig. 3 shows that focused comparison results in fewer comparison pairs than the naïve one and the difference becomes more salient when many student programs are involved. This is expected as that comparison makes the number of student programs linear to the number of comparison pairs while naïve comparison considers the relation as quadratic.

Taking the most extreme scenario with 100 student programs, focused comparison can exclude 3950 comparison pairs, which leads to 79.8% reduction. If each comparison pair takes one second, this can save about one hour of execution time.

We are aware that when the number of student programs are low (e.g., when the number of student programs is 10), the difference becomes harder to see. However, it still leads to fewer comparison pairs than the naïve one, except when no student programs are considered.

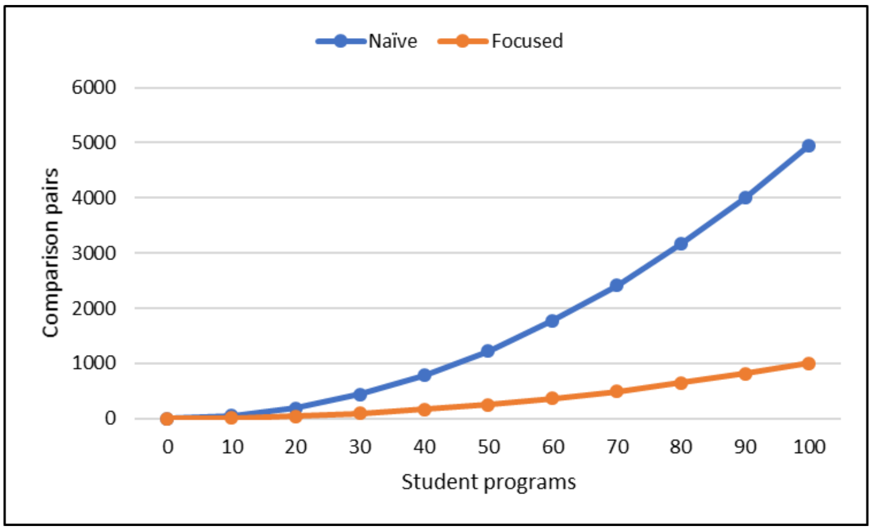


Fig. 3. The number of comparison pairs: focused vs naïve comparison

3.2 The impact of cosine correlation

Theoretically, cosine correlation is faster than running Karp-Rabin greedy string tiling [25], a common string-matching algorithm for this task [32], as the former only takes linear computation time while the latter takes it quadratically. This subsection evaluates how much is the time reduction caused by replacing the latter with the former.

The evaluation involves two token representations: regular and syntax tree token strings. They are used in our comparison modes. Regular token string is resulted from tokenising the source code directly with ANTLR and it is used for *CosineTS* and *RKRGST*. Syntax tree token string is resulted from linearising the syntax tree in pre-order manner. It is used for *CosineAST* and *Structure*. Each representation has one mode with cosine correlation (either *CosineTS* or *CosineAST*) and another mode with running Karp-Rabin greedy string tiling (either *RKRGST* or *Structure*).

The reduced execution time was measured in twofold. At first, the execution time of each mode is measured by searching the copied programs of a student program in two sets of introductory programming assessments (with 2426 Python files in total). After that, the time difference between the two modes is calculated and normalised to the execution time of the string-matching mode (either *RKRGST* or *Structure*).

Fig. 4 shows that replacing running Karp-Rabin greedy string tiling with cosine correlation results in shorter execution time. It reduces about 11% for regular token string and 38.9% for the syntax tree one. Time reduction for syntax tree token string is larger since the strings have more tokens as a result of linearising the syntax trees, and such larger number of tokens leads to longer execution time for string-matching algorithm due to its quadratic computation.

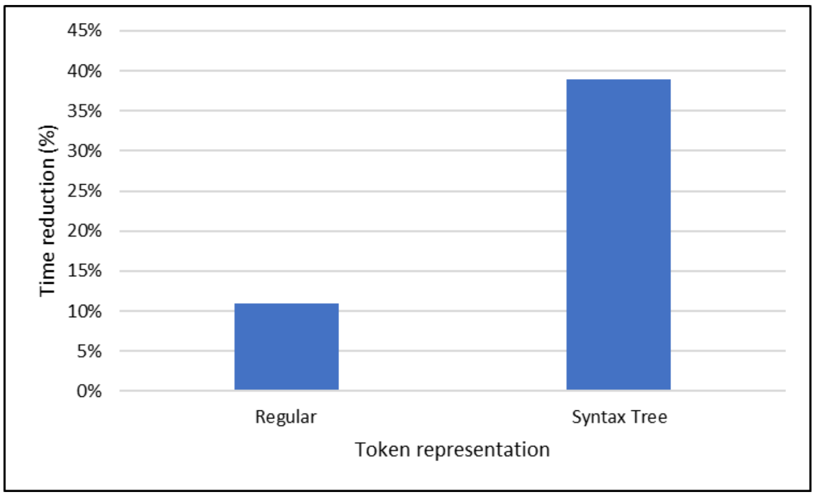


Fig. 4. The reduced time for regular and syntax tree token strings with cosine correlation

3.3 The impact of comparison modes

Four comparison modes (*CosineTS*, *RKRGST*, *CosineAST*, and *Structure*) are proposed for the detection. This subsection evaluates the impact of those modes under two evaluation metrics: f-score and execution time. The former covers effectiveness while the latter covers efficiency. This is expected to provide a brief summary about the characteristics of the proposed modes.

F-score is often used to measure effectiveness in general where higher value is preferred. It is the harmonic mean between precision and recall, calculated as in (1). Precision is the proportion of copied student programs in the suspected results. The equation can be seen in (2) and it is resulted from dividing the number of true positives with the sum of the number of true and false positives. Recall is the proportion of suspected results in the copied student programs. The equation can be seen in (3) and it is resulted from dividing true positives with the number of true positives plus the number of false negatives.

$$Fscore = \frac{2\ x\ Precision\ x\ Recall}{Precision + Recall} \tag{1}$$

$$Precision = \frac{true\_positives}{true\_positives + false\_positives} \tag{2}$$

$$Recall = \frac{true\_positives}{true\_positives + false\_negatives} \tag{3}$$

For measuring f-score, a Java introductory programming data set in [32] was used. The data set covers seven introductory programming materials: output, input, branching, looping, array, method, and matrix. Copied programs are mapped to six plagiarism levels defined by [13]: comment and whitespace modification, identifier renaming, component declaration relocation, method structure change, program statement replacement, and logic change. They are referred as level-1 to level-6 respectively where higher level is more difficult to apply and less frequently found in real cases. In total, the data set contains 7 original student programs (or queries), 105 independent student programs, and 355 copied student programs toward the originals. Each query has 20 independent student programs and up to 36 copied student programs (up to 9 programs per plagiarism level).

Execution time (in seconds) was recorded similarly as the one used for measuring the impact of cosine correlation. It is the amount of time required for searching the copied programs of a student program in 2426 Python files. Lower value is preferred for this metric as faster execution is proportional to higher scalability.

Fig. 5 shows that all four comparison modes are equally effective for the first plagiarism level. It is expected as that level is focused on modifying comments and whitespaces, two components that are ignored by all modes. On level-2 (which is about identifier renaming), *Structure* becomes the most effective as the modification does not change token order and its impact can be mitigated with the consideration of syntax tokens. This mode, however, becomes as effective as *RKRGST* on level-3 (which is about component declaration relocation); relocating declaration statements does not change the syntax tokens, leading to no improvement with those tokens on board.

For the remaining levels (which are about method structure change, program statement replacement, and logic change), *Structure* becomes the least effective as the modification affects syntax tokens, enlarging the number of mismatches. *CosineTS*, which is the least effective on the first three levels, gradually experiences effectiveness improvement, making it the second highest on level-6 (logic change).

In terms of efficiency (see Fig. 6), *CosineTS* is the most effective one, taking only about 89 seconds to process 2426 comparisons. This is followed by *RKRGST* that takes more time for calculating the similarities; its algorithm has quadratic complexity while *CosineTS*'s algorithm is linear time.

CosineAST and *Structure* are slower than the first two as syntax trees should be generated and linearized prior comparison. Time required for that tree generation can be longer if the code is complex [17]. *Structure* is the slowest one due to the combination of quadratic similarity algorithm and tree generation.

To sum up, *Structure* is exclusively beneficial to deal with modifications related to identifier renaming, while *RKRGST* is the most effective one for remaining levels. For scalability, *CosineTS* is the most preferred one due to its fast computation, followed by *RKRGST*, *CosineAST*, and *Structure*.

CosineTS is advised if many student programs are considered. However, if only few of them are involved, *RKRGST* can be used for higher effectiveness. *CosineAST* can be used as a replacement of *CosineTS* if students tend to disguise their programs with identifier renaming. This is also similar to *Structure*, which can replace *RKRGST* for dealing with identifier renaming.

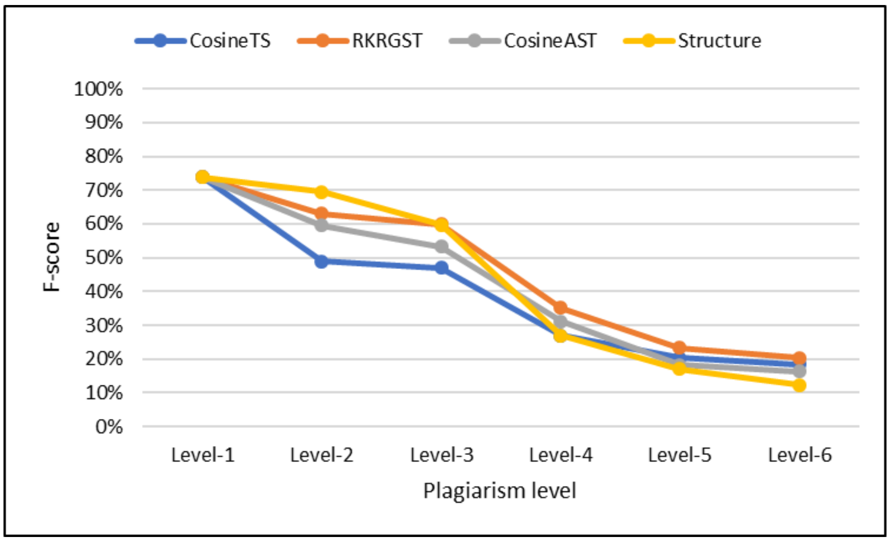


Fig. 5. The effectiveness of the comparison modes in terms of f-score

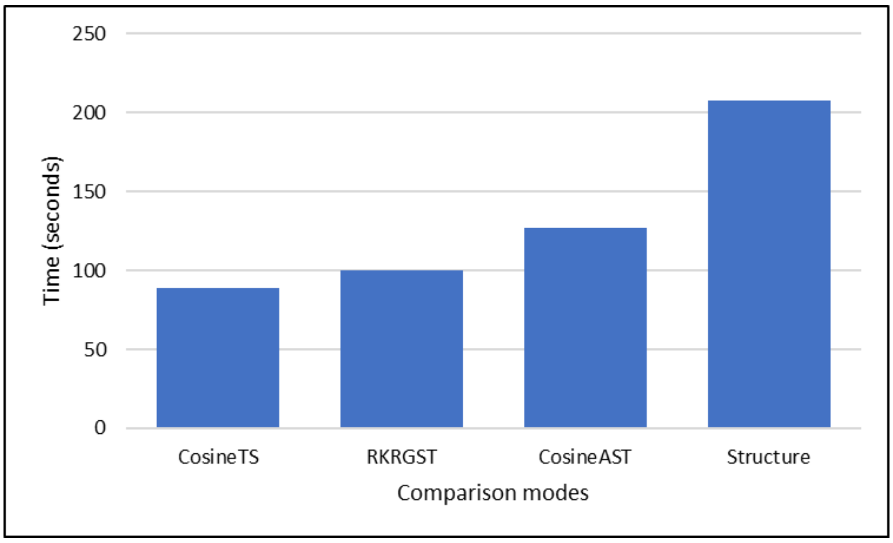


Fig. 6. The efficiency of the comparison modes in terms of execution time

4 Conclusion and Future Work

A scalable code similarity detection for maintaining academic integrity in programming is proposed in this paper. It is uniquely featured with online architecture and focused comparison. The former facilitates student program sharing among lecturers and assures that the computation can be performed regardless the lecturers' personal computers’ specification. The latter can shorten the execution time as only some student programs are considered. To enhance the scalability, the similarity measurement is cosine correlation, an algorithm with linear time complexity.

According to our evaluation, focused comparison can exclude many comparison pairs if many student programs are involved. With 100 student programs on board, it can exclude 79.8% comparison pairs. This obviously leads to shorter execution time as time is proportional to the number of comparisons.

Replacing running Karp-Rabin greedy string tiling with cosine correlation can also shorten the execution time. The benefit becomes larger when the token strings are longer (such as those resulted from linearized syntax trees).

Our detection technique is featured with four comparison modes: *CosineTS*, *RKRGST*, *CosineAST*, and *Structure*. Among those, *CosineTS* is the most scalable one while *RKRGST* is the most effective one for most plagiarism levels. Other two modes can be used in dealing with small-sized student programs which modifications are mainly about renaming identifiers.

Our detection technique is considerably scalable as it can search copied programs from 2426 student programs for less than one and a half minute. It is also effective in dealing with superficial modifications that are commonly found in programming assessments (the first three plagiarism levels).

For future work, we plan to use the detection technique for some programming courses and summarise the experiences. It is expected to enrich our current findings from user perspective. In addition, we also plan to evaluate the comparison modes with other metrics to gain deeper understanding of their characteristics.

5 Acknowledgement

This research has been supported by a research grant provided by Maranatha Christian University.

6 References

[1] Metruk, R., Confronting the Challenges of MALL: Distraction, Cheating, and Teacher Readiness. *International Journal of Emerging Technologies in Learning (iJET)*, 2020, 15(2): 4–14. https://doi.org/10.3991/ijet.v15i02.11325

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

[3] Aikina, T. Y., and Bolsunovskaya, L., M., Moodle-Based Learning: Motivating and Demotivating Factors. *International Journal of Emerging Technologies in Learning (iJET)*, 2020, 15(2): 239–248. https://doi.org/10.3991/ijet.v15i02.11297

[4] Ozvoldova, M., and Ondrušek, P., Integration of Online Labs into Educational Systems. *International Journal of Online and Biomedical Engineering (iJOE)*, 2015, 11(6): 54-59. https://doi.org/10.3991/ijoe.v11i6.5145

[5] Batane, T., Turning to Turnitin to fight plagiarism among university students. *Journal of Educational Technology & Society*, 2010, 13(2): 1–12. https://doi.org/10.3126/nelta.v12i1.3440

[6] Simon, Cook, B., Sheard, J., Carbone, A. and Johnson, C., Academic integrity perceptions regarding computing assessments and essays. *10th Annual Conference on International Computing Education Research*, 2014: 107–114. https://doi.org/10.1145/2632320.2632342

[7] Sheard, J., Simon, Butler, M., Falkner, K., Morgan, M. and Weerasinghe, A., Strategies for maintaining academic integrity in first-year computing courses. *2017 ACM Conference on Innovation and Technology in Computer Science Education*, 2017: 244–249. https://doi.org/10.1145/3059009.3059064

[8] Simon, Sheard, J., Morgan, M., Petersen, A., Settle, A. and Sinclair, J., Informing students about academic integrity in programming. *20th Australasian Computing Education Conference*, 2018: 113–122. https://doi.org/10.1145/3160489.3160502

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

[10] Prechelt, L., Malpohl, G. and Philippsen, M., Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science*, 2002, 8(11): 1016–1038.

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

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

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

[14] Jones, E. L., Metrics based plagiarism monitoring. *Journal of Computing Sciences in Colleges*, 2001, 16(4): 253–261.

[15] Al-Khanjari, Z. A., Fiaidhi, J. A., Al-Hinai, R. A. and Kutti, N. S., PlagDetect: a Java programming plagiarism detection tool. *ACM Inroads*, 2010, 1(4): 66–71. https://doi.org/10.1145/1869746.1869766

[16] Croft, W. B., Metzler, D. and Strohman, T., *Search Engines: Information Retrieval in Practice*. USA: Addison-Wesley, 542 (2010).

[17] Karnalim, O. and Simon, Syntax trees and information retrieval to improve code similarity detection. *Twenty-Second Australasian Computing Education Conference*, 2020: 48–55. https://doi.org/10.1145/3373165.3373171

[18] Allyson, F. B., Danilo, M. L., José, S. M. and Giovanni, B. C., Sherlock N-overlap: invasive normalization and overlap coefficient for the similarity analysis between source code. IEEE Transactions on Computers, 2019, 68(5): 740-751. https://doi.org/10.1109/TC.2018.2881449

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

[20] Arwin, C. and Tahaghoghi, S. M. M., Plagiarism detection across programming languages. *29th Australasian Computer Science Conference*, 2006: 277–286.

[21] Ahtiainen, A., Surakka, S., and Rahikainen, M., Plaggie: GNU-licensed source code plagiarism detection engine for Java exercises. *Sixth Baltic Sea conference on Computing Education Research, Koli Calling 2006*, 2006: 141–142. https://doi.org/10.1145/1315803.1315831

[22] Joy, M. and Luck, M., Plagiarism in programming assignments. *IEEE Transactions on Education*, 1999, 42(2): 129–133. https://doi.org/10.1109/13.762946
[23] Sheard, J., Carbone, A. and Dick, M., Determination of factors which impact on IT students' propensity to cheat. *Fifth Australasian conference on Computing education*, 2003: 119–126.
[24] Parr, T., *The Definitive ANTLR 4 Reference*. Dallas: Pragmatic Bookshelf, 432 (2013).
[25] Wise, M. J., YAP3: improved detection of similarities in computer program and other texts. *27th SIGCSE* Technical *Symposium on Computer Science Education*, 1996: 130–134. https://doi.org/10.1145/236462.236525
[26] Kikuchi, H., Goto, T., Wakatsuki, M., and Nishino, T., A source code plagiarism detecting method using alignment with abstract syntax tree elements. *15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, 2014: 1–6. https://doi.org/10.1109/SNPD.2014.6888733
[27] Wang, L., Jiang, L. and Qin, G., A search of Verilog code plagiarism detection method. *13th International Conference on Computer Science & Education*, 2018: 1–5. https://doi.org/10.1109/ICCSE.2018.8468817
[28] Mann, S. and Frew, Z., Similarity and originality in code: plagiarism and normal variation in student assignments. *Eighth Australasian Conference on Computing Education*, 2006: 143–150.
[29] Yang, F.-P., Jiau, H. C. and Ssu, K.-F., Beyond plagiarism: an active learning method to analyze causes behind code-similarity. *Computers & Education*, 2014, 70(1): 161–172. https://doi.org/10.1016/j.compedu.2013.08.005
[30] Misić, M. J., Protić, J. and Tomasević, M. V., Improving source code plagiarism detection: lessons learned. *25th Telecommunication Forum*, Belgrade, Serbia, 2017: 1–8. https://doi.org/10.1109/TELFOR.2017.8249481
[31] Franclinton, R. and Karnalim, O., A Language-Independent Library for Observing Source Code Plagiarism. *Journal of Information Systems Engineering and Business Intelligence*, 2019, 5(2): 110–119. https://doi.org/10.20473/jisebi.5.2.110-119
[32] Karnalim, O., Budi, S., Toba, H. and Joy, M., Source code plagiarism detection in academia with information retrieval: dataset and the observation. *Informatics in Education*, 2019, 18(2): 321–344. https://doi.org/10.15388/infedu.2019.15

7 Authors

Ricardo Franclinton will be graduated with a Bachelor of Computer degree from Maranatha Christian University in 2020. His interest is about computer science education and solving competitive programming problems. In addition to academic works, he is also interested in organization. He was a member of a student senate from 2017 to 2019 where in 2019 he was elected as the president of student senate in the last year. Currently, he is working in a software house in Bandung.

Oscar Karnalim graduated with a Bachelor of Engineering degree from Parahyangan Catholic University in 2011, and completed his Master's degree at Bandung Institute of Technology (ITB) in 2014. His interest is about computer science education, especially source code plagiarism and educational tools. He works at Maranatha Christian University as a full-time lecturer. Currently, he is pursuing a PhD in Information Technology at University of Newcastle, Australia.

Mewati Ayub graduated with a Bachelor of Informatics from Bandung Institute of Technology (ITB) in 1986, and completed her Master's degree at Bandung Institute of Technology in 1996, and her doctoral degree at Bandung Institute of Technology in 2006. She has been working as a faculty member in the Faculty of Information Technology at Maranatha Christian University since 2006. Her specialty is in the field of computer science education, software engineering, and data analytics.

Article submitted 2020-03-17. Resubmitted 2020-06-17. Final acceptance 2020-06-21. Final version published as submitted by the authors.