Syntax Trees and Information Retrieval to Improve Code Similarity Detection

Oscar Karnalim*
University of Newcastle
New South Wales, Australia
Oscar.Karnalim@uon.edu.au

Simon
University of Newcastle
New South Wales, Australia
Simon@newcastle.edu.au

ABSTRACT

In dealing with source code plagiarism and collusion, automated code similarity detection can be used to filter student submissions and draw attention to pairs of programs that appear unduly similar. The effectiveness of the detection process can be improved by considering more structural information about each program, but the ensuing computation can increase the processing time. This paper proposes a similarity detection technique that uses richer structural information than normal while maintaining a reasonable execution time. The technique generates the syntax trees of program code files, extracts directly connected n-gram structure tokens from them, and performs the subsequent comparisons using an algorithm from information retrieval, cosine correlation in the vector space model. Evaluation of the approach shows that consideration of the program structure (i.e., syntax tree) increases the recall and f-score (measures of effectiveness) at the expense of execution time (a measure of efficiency). However, the use of an information retrieval comparison process goes some way to offsetting this loss of efficiency.

KEYWORDS

source code similarity detection, plagiarism and collusion in programming, syntax tree, information retrieval, computing education

ACM Reference Format:
Oscar Karnalim and Simon. 2020. Syntax Trees and Information Retrieval to Improve Code Similarity Detection. In *Proceedings of the Twenty-Second Australasian Computing Education Conference (ACE'20), February 3–7, 2020, Melbourne, VIC, Australia.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3373165.3373171

1 INTRODUCTION

Academic misconduct appears to be an increasing issue in higher education. Assessment of students relies on the understanding that the work being assessed has actually been carried out by the students who submitted it. Students can subvert this understanding by submitting work that includes substantial amounts of unacknowledged content from other sources. The impact of this behaviour has been expressed by Harris [15]:

> Students who knowingly plagiarize cheat others by inflating their marks at the expense of students who are sincerely trying to do their own work. They cheat themselves by denying their own education and they cheat the author by claiming the work to be theirs [15, p133].

In computing education, concerns about academic integrity apply not only to text-based assessments but also to programming assessments, and a number of publications offer support to academics in addressing these concerns. Simon et al., for instance, explain the need for students to be clearly informed about academic integrity in programming [28].

In programming assessment, source code plagiarism and collusion are common ways of breaching the rules of academic integrity [26]. Both entail the reuse of source code written by others without appropriately acknowledging the original authors [5, 11]. The distinction is that plagiarism tends to entail copying from writers who are are not party to the copying, while with collusion, all authors take part in the misconduct.

Several strategies can be used to deal with academic misconduct [25]. At the beginning of each course, the students can be educated about academic integrity and acceptable practices, with information from either institutional policies or domain-specific codes of practice [13, 27, 28]. Extra assessment measurements such as oral presentations [14] can be applied to help establish the authorship of submitted work. In addition, computing educators often make use of automated similarity detection tools [30] to draw the marker's attention to suspiciously similar submissions.

Code similarity detection is usually applied to the set of student programs for a given assessment item in a given class, and is thus more likely to detect collusion than plagiarism. Detection of plagiarism, copying of publicly available code, would entail establishing and maintaining a collection of code from vast numbers of public web sites, which would not be practical. Other forms of academic misconduct, such as contract cheating, are also beyond the reach of similarity detection; to detect these, academics remain reliant on other techniques such as recognising an unexpected programming style or substantial use of programming features that have not been taught in the class.

Code similarity detection generally works by converting each program to a string of tokens and then comparing the strings with a string matching algorithm such as running Karp-Rabin greedy string tiling [34]. This approach can be somewhat lacking in effectiveness: it can cast suspicion on program pairs that are not

*Also with Maranatha Christian University.

ACE'20, February 3–7, 2020, Melbourne, VIC, Australia

ACM ISBN 978-1-4503-7686-0/20/02…$15.00
https://doi.org/10.1145/3373165.3373171

copied, while failing to identify pairs that have been copied and subsequently disguised.

This paper proposes to improve the effectiveness of code comparison with a technique that still compares token strings, but strings that include more structural information by combining structural nodes taken from the syntax trees of the programs, which tend to be less affected by surface disguises. However, the generation and comparison of syntax trees can be computationally expensive, significantly slowing the process. To help compensate for this reduction in efficiency, we apply a linear-time similarity measurement from the field of information retrieval rather than the quadratic-time string comparison algorithm.

To the best of our knowledge, this is the first time that the syntax tree structure has been combined with a similarity measure from information retrieval in the task of source code similarity detection for academic plagiarism and collusion.

Our research questions are:

RQ1 How are the effectiveness and efficiency of code similarity detection affected by the use of a data structure that reflects program structure as opposed to a simple linear sequence?

RQ2 How are the effectiveness and efficiency of code similarity detection affected by the use of a comparison method from information retrieval as opposed to a standard string comparison method?

2 RELATED WORK

A number of similarity detection techniques have been introduced to deal with source code plagiarism and collusion in academia [30]. Initially, these techniques relied on superficial features. For example, Ottenstein [21] determines source code similarity on the basis of the numbers of unique operators, unique operands, operators, and operands.

Reflecting an awareness that such features do not accurately reflect the source code, the token string — a compact array-like representation of source code — was introduced [33]. This representation has become popular and has been applied in many detection techniques; token strings adequately depict the source code content and are easy to generate.

JPlag [23], a benchmark tool for source code similarity detection, relies on a token string representation and uses a comparison adapted from running Karp-Rabin greedy string tiling (RKRGST) [34]. Several other techniques followed JPlag in combining these two approaches with various modifications. Bejarano et al. [2] simplified the algorithm by removing the iteration in finding matched substrings. Đurić and Gašević [8] excluded template code prior to comparison. Sulistiani and Karnalim [30] incorporated information retrieval techniques to initially filter the source code files before actually comparing them.

More advanced structural representations were also introduced, aiming for improved effectiveness. A low-level token string [16], for instance, is a token sequence extracted from the executable files that result from compiling the source code. This representation was argued to be more concise than source code token strings since no delimiters are involved. This technique sees through some trivial disguises as the compilation optimises the content. However, as the representation relies heavily on the programming language being used, it does not readily extend to additional programming languages. To date, these techniques have been used only with Java [16] and C# [24].

A syntax tree represents how the source code tokens form a valid program based on their parsing rules. This can increase the accuracy of similarity detection as the syntax tree is impervious to trivial disguises. However, since direct syntax tree comparison is computationally expensive [12], some alternative comparisons have been proposed. Fu et al. [12] used the tree kernel method (adapted from natural language processing) to heuristically measure the similarity between programs. Kikuchi et al. [19] and Wang et al. [32] treated the syntax trees as token strings, respectively linearising them in pre-order [19] and converting them to hash value sequences [32].

A program dependency graph describes how program instructions interact with one another. Song et al. [29] combined this with a parse tree (an unoptimised version of the syntax tree) in a composite kernel to measure similarity.

Information retrieval (IR) is the task of identifying relevant documents from a large collection based on a piece of information [7] — for example, identifying web pages that include a specific search string. Some of its mechanisms have been applied to enhance efficiency in source code similarity detection, since most IR measurements are fast to compute. Flores et al. [10] and Karnalim [17] used cosine correlation in source code similarity detection, relying respectively on source code characters and low-level token strings. Sulistiani and Karnalim [30] used the same algorithm as an initial filter for string-matching comparison, inspired partly by Burrows et al. [3]. Mozgovoy et al. [20] applied the same filtering mechanism, but with a different set of algorithms.

Techniques from IR can be useful in other ways besides enhancing efficiency. Cosma and Joy [6] incorporated latent semantic analysis (LSA) to enrich the performance of existing detection techniques. To measure code similarity across programming languages, Flores et al. [9] and Ullah et al. [31] also used LSA, while Arwin and Tahaghoghi [1] used both LSA and BM25, a function that ranks documents according to the frequencies of the search terms within them.

3 PROPOSED TECHNIQUE

Source code token strings are still frequently used, despite the existence of many more advanced representations, as the latter representations are often computationally expensive. For example, the low-level token string requires the source code to be compiled [16] prior to comparison.

This paper introduces a source code similarity detection technique that enhances effectiveness by providing more structural information than the commonly used technique (which relies on token string and running Karp-Rabin greedy string tiling [34]), but without an overwhelming loss in efficiency. The technique bases its comparison on a linearised form of the internal nodes of the syntax tree. This form is arguably resistant to modification as it is automatically generated based on how the source code tokens form the whole program. To enhance the efficiency of the technique, source code similarity is determined with a linear-time information

retrieval algorithm called cosine correlation in the vector space model [7].

Our technique is substantially more time-efficient than techniques proposed by Kikuchi et al. [19] and Wang et al. [32], as the cosine correlation is linear in time while their similarity algorithms are quadratic due to their use of string-matching algorithms. Our technique can also be more sensitive: rather than considering the whole syntax tree [19], it considers only the internal nodes, which are more robust to code modification. Further, it keeps the original program structure rather than hashing it and losing potentially valuable information [32].

Our detection technique involves two stages: preprocessing, converting each source code file to a token index; and comparison, comparing the files pairwise (via their indexes) and alerting the human markers to suspicious cases.

The preprocessing phase is conducted once for each source code file and has five sub-phases, as shown in Figure 1. Once the source code file has been read, the syntax tree is generated using ANTLR [22], a process that automatically excludes the comments and white space.

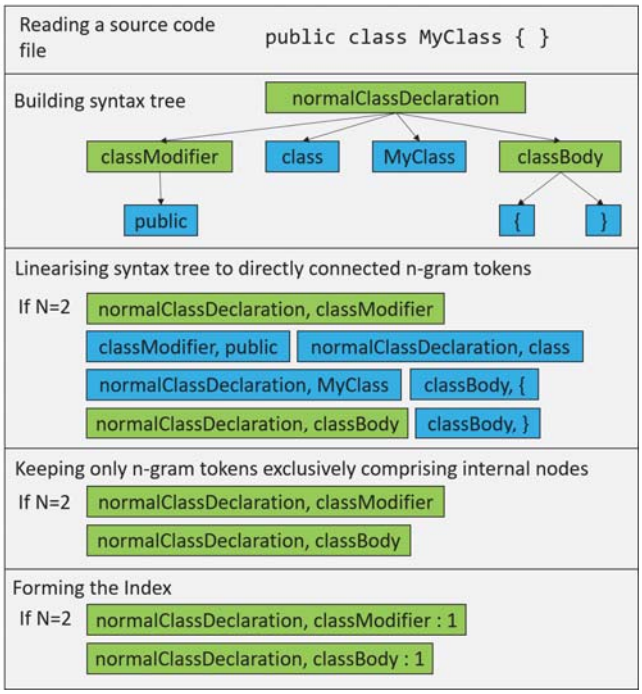

Figure 1: Sub-phases in the preprocessing phase, with a trivial example

Directly connected n-gram tokens are then generated by linearising the syntax tree in a pre-order manner, with each token formed by merging *n* adjacent directly connected tokens. The direct connection ensures that each resulting *n*-gram token depicts the real structure of the syntax tree. Taking as an example the syntax tree in Figure 1, *{MyClass, classBody}* would be considered as a 2-gram token if naively linearised, even though those two nodes are not directly connected on the tree.

We are aware that naive linearisation can produce some meaningful n-gram tokens when all of the members are source code tokens (e.g., *{public, class}* and *{class, MyClass}* from the syntax tree in Figure 1). However, they are still ignored as our technique focuses on parse structure rather than source code tokens, which can be easily disguised at source code level. This is also the reason why all n-gram tokens with at least one source code token (the blue rectangles in the third sub-phase of Figure 1) are excluded in the following sub-phases.

Finally, filtered n-gram tokens are stored in an index, a hash map containing key-value tuples in which the key refers to a token's mnemonic and the value to its frequency of occurrence. In the final phase of Figure 1, each tuple has an occurrence frequency of 1 as there are only two tuples and they are distinct.

The comparison phase first pairs all student programs with one another. For example, if there are three student programs called *P1*, *P2*, and *P3*, three pairs will be generated: *P1-P2*, *P1-P3*, and *P2-P3*. The similarity degree of each pair is then calculated with cosine correlation [7], assuming that all programs are mapped on a vector space model based on their indexes. A pair is considered as suspicious if its similarity degree is greater than or equal to both 75% and the average similarity degree of all pairs. The former threshold (75%) ensures that each suspected pair displays high similarity, even though on challenging assessments the latter threshold will typically be quite low.

4 EVALUATION

Our proposed technique incorporates five identifiable factors that are intended to enhance its performance:

- the use of program structure by way of a syntax tree
- a comparison method from information retrieval
- combining tokens into n-grams
- retaining only n-grams that are directly connected
- discarding terminal nodes of the tree

This study evaluates each of these factors in turn for their efficiency and effectiveness, and then similarly evaluates the combination of all five factors against a technique that is in common use in source code similarity detection for academic plagiarism and collusion.

The evaluation was conducted on 94 assessment tasks taken from an introductory programming course using Java, with anything from three to 22 student programs in each task. Together there are 1028 student programs, and comparing all of the programs within each task results in 6044 pairwise comparisons.

For the purpose of benchmarking, correct pairs for each assessment task (pairs that share suspicious similarity) were selected using JPlag [23], a common tool for source code similarity detection. The source code files were pairwise compared and the correct pairs are those whose degree of similarity was higher than both 75% and the average degree of similarity. While somewhat arbitrary, this threshold is arguably acceptable on this data set since, for each assessment task, all pairs meeting the combined criterion were suspiciously similar according to our manual observation.

Users of code similarity detection software will be aware that the optimal similarity threshold for raising suspicion can vary with the assessment item. For simple assessments such as ours, where

relatively high similarity is expected between completely independent submissions, a high threshold is needed to reduce the number of false positives, program pairs that are suspected but are not a result of copying. Advanced assessments, where similarity between submissions is expected to be much lower, will use a lower threshold.

To evaluate effectiveness we use three metrics:

- Precision: the proportion of correctly suspected pairs (true positives) to suspected pairs (positives); a higher precision indicates fewer false positives, program pairs detected as suspicious when they are not
- Recall: the proportion of correctly suspected pairs (true positives) to correct pairs (copied pairs); higher recall indicates fewer false negatives, program pairs not detected as suspicious when they are
- F-score: the harmonic mean of precision and recall, defined as in Equation (1)

$$f\text{-}score = \frac{2 \times precision \times recall}{precision + recall} \quad (1)$$

Figure 2 should help readers to understand the foregoing definitions of precision and recall.

		copied: yes	copied: no
suspected	yes	true positive: copied and suspected	false positive: not copied but suspected
	no	false negative: copied but not suspected	true negative: not copied and not suspected

Figure 2: Explanation of true and false positives and negatives

To evaluate efficiency we use the token string length and the execution time.

All metrics except execution time were measured by averaging their performance across assessment tasks. A two-tailed paired t-test with 95% confidence rate was used to test the significance of their differences, and we discuss only significant results. Execution time was measured as the time taken (on the same computer) for all 6044 pairwise comparisons in the data set, as the time required to process each assessment task is too small and is easily affected by hardware and operating system dependency. As a single value for each test, execution time cannot be tested for significance of differences.

In all of the tables in this section, significant differences are marked by asterisks. A single asterisk indicates that $p<0.05$, two asterisks indicate that $p<0.01$, and three asterisks indicate that $p<0.001$.

4.1 Program Structure vs Linear Sequence

This subsection evaluates the effectiveness and efficiency of three different approaches to forming a string of tokens for comparison.

The *linear* approach is a common technique for program similarity detection in academia; it converts source code files to token strings (with ANTLR [22]), removes the comments, compares the strings using running Karp-Rabin greedy string tiling (RKRGST) [34] with two as its minimum matching length, and calculates the degree of similarity using average normalisation.

The *syntax tree* approach also uses ANTLR, but to form the syntax tree of the program. This tree is then linearised in pre-order, resulting in a token string quite different from the linear one.

The *parse tree* approach again uses a tree, but the full parse tree for the program. A parse tree is typically substantially bigger than the corresponding syntax tree, as it contains every node required for parsing the program, whereas the syntax tree contains only the nodes that represent the essence of the program's structure [4].

For the purposes of comparison, having produced the syntax tree token strings and the parse tree strings, we compare them using the same approach as for the linear token strings: RKRGST with average normalisation.

Table 1: Effectiveness and efficiency of running Karp-Rabin greedy string tiling comparison on linear, syntax tree, and parse tree representations; t-statistic values given in parentheses for significant differences from linear

	Linear	Syntax tree	Parse tree
precision	75%	79%	60% *** (-4.0)
recall	52%	73% *** (8.7)	86% *** (9.8)
f-score	55%	72% *** (6.6)	67% *** (3.8)
token string length	204	355 *** (15.6)	960 *** (13.4)
execution time	14s	241s	5941s

Table 1 shows that using the syntax tree significantly improves effectiveness, by way of recall and f-score; parse tree does the same, but at a significant loss of precision. At the same time, both techniques show deterioration in efficiency, by way of a significant increase in average token string length and a substantial increase in execution time. This increase in execution time reflects two components: the computation required for generating the trees, and slower comparisons of the token strings due to the increased token string length.

To summarise, the use of program structure enhances the effectiveness of program similarity detection, but at the expense of a great increase in execution time. Further, the parse tree representation takes an order of magnitude more time than the more optimal syntax tree.

4.2 A Comparison Method from Information Retrieval

If we are to take advantage of the enhanced effectiveness of using the programs' structure, we should seek a way to offset the increase in the execution times of comparisons. For this we enlist cosine correlation, a similarity measurement from the field of information retrieval.

For the two program structure detection techniques in section 4.1, we replaced RKRGST with cosine correlation as the comparison algorithm.

Table 2 shows that the use of cosine correlation reduces the execution time by some 40% for syntax tree and more than 95%

Table 2: Effectiveness and efficiency of comparison using cosine correlation and running Karp-Rabin greedy string tiling, with both syntax trees and parse trees

	Syntax tree		Parse tree	
	RKR	CosCor	RKR	CosCor
precision	79%	51% *** (-8.7)	60%	49% *** (-5.1)
recall	73%	85% *** (3.6)	86%	87%
f-score	72%	58% *** (-5.1)	67%	57% *** (-4.6)
token str lgth	355	355	960	960
exec time	241s	137s	5941s	138s

Table 3: Effectiveness and efficiency of different values of *n*, the number of tokens combined into each n-gram, for syntax trees with cosine correlation; each significance marker refers to the difference from the value in the preceding column

	n=1	n=2	n=3	n=4
precision	51%	59% *** (-4.8)	73% *** (-5.1)	70%
recall	85%	80% *** (3.7)	66% *** (4.6)	47% *** (8.5)
f-score	58%	63% *** (-3.4)	63%	49% *** (6.1)
tok str lgth	355	354	353	352
exec time	144s	145s	144s	146s

for parse tree. The near identical execution times for these two approaches attest to the linear nature of cosine correlation compared with the quadratic nature of RKRGST.

The improvement in efficiency comes with a deterioration in effectiveness. Although syntax tree shows improved recall, both tree structures show substantial loss of precision, which contributes to a loss in f-score.

It is clear that the use of this IR-based comparison algorithm provides faster comparison than RKRGST (and probably most other string-matching algorithms, as they are not linear in time). It can also increase recall, but with a trade-off on precision.

While it was instructive to retain the parse tree representation for this comparison, it appears to have no advantage over the syntax tree, so from here onward we shall work only with the syntax tree program structure.

4.3 Formation of N-Grams

As mentioned in section 4.1, the syntax tree is linearised by reading it in pre-order. For the comparison carried out in that section, each node formed a distinct element of the string to be compared. However, there are sometimes efficiencies to be gained by combining *n* contiguous tokens into an n-gram, which then becomes the element of comparison. This subsection evaluates how varying *n* affects the effectiveness and efficiency of the process.

Four *n* variants, from 1 to 4, were used in this evaluation. Each of them was applied to the syntax tree, using cosine correlation as the comparison algorithm. These variants are selected due to their frequent use in IR based source code similarity detection [3, 9, 10, 17, 18] and/or IR for text comparison [7].

For each variant except *n*=1, the metrics were compared with those of the *n–1* variant. Table 3 shows that recall decreases significantly with each step in *n*, while precision increases as *n* moves to 2 and 3. The overall impact on f-score is that it increases for *n*=2, remains the same for *n*=3, and drops significantly for *n*=4. Higher values of *n* lead to fewer suspected pairs because of the stricter matching condition, with each token being considered in conjunction with adjacent tokens. This reduction means that there are fewer incorrectly suspected pairs (higher precision), but also fewer correctly suspected pairs (lower recall).

In terms of efficiency, adding 1 to *n* subtracts 1 from the length of the token string, which is not a significant change. Changes in the execution time are also at the level of noise, so there is no reason to believe that varying *n* has any effect on the efficiency of the process.

While the f-score values clearly suggest that 2 and 3 are the best values of *n*, in the remaining tests we shall continue to test all four values, to explore their effects on the trade-off between precision and recall.

4.4 Considering only Directly Connected Nodes

The inverse relationship between *n* and recall in the previous section is the result of naively extracting n-gram tokens from the linearised syntax tree. Some of the linearised tokens are adjacent only by the coincidence of being in adjacent sub-trees. The larger the value of *n*, the greater the chance that such tokens will be merged into single n-gram tokens, leading to fewer correctly suspected pairs. In this step of the testing we deal with this issue by discarding any n-gram whose tokens are not directly linked on the syntax tree.

Table 4 shows the effects of discarding n-grams that include nodes that are not directly connected. No results are shown for *n*=1, as the notion of directly connected nodes has no meaning for a 1-gram. The table shows that considering only directly connected nodes increases recall, at the expense of precision, for each value of *n*. The only significant effect on f-score is when *n*=4, but this serves only to bring the f-score back into the same region as for lower values of *n* — presumably offsetting the reduction in recall that was due to coincidentally adjacent tokens.

Higher values of *n* show small but significant reductions in the average token string length. As in the preceding section, the execution times for different values of *n* appear to differ only at the level of noise.

4.5 Discarding Terminal Nodes of the Syntax Tree

When students work together on individual assignments, or copy the work of other students, they will often try to disguise the fact. However, novice programmers who feel the need to copy the work of other students are seldom capable of sophisticated program disguise, and will often resort to changing comments, white space, or variable names. When these items appear at all in the syntax tree, it is generally as terminal nodes: the program's structure, contained in the internal nodes, is seldom affected. Removing the terminal nodes from the syntax tree before linearising it, leaving only the

Table 4: Effectiveness and efficiency changes with syntax trees and cosine correlation when retaining only n-grams that consist entirely of directly connected edges; each cell gives the original value, from Table 3, followed by the new value with direct connections only

	n=2	n=3
precision	59% →56% ** (-2.5)	73% →62% *** (-4.1)
recall	80% →85% *** (4.2)	66% →83% *** (5.7)
f-score	63% →63%	63% →67%
token str lgth	354 →354	353 →351 *** (-76.6)
exec time	145s →144s	144s →144s
	n=4	
precision	70% →64% * (-2.1)	
recall	47% →74% *** (8.3)	
f-score	49% →64% *** (4.8)	
token str lgth	352 →344 *** (-71.5)	
exec time	146s →143s	

internal nodes, might improve the outcomes of the similarity detection process and increase the number of correctly suspected pairs.

Table 5 shows that discarding terminal nodes brings a slight drop in precision, but only for n=3, and small improvements in recall for n=1 and n=4.

Discarding the terminal nodes substantially reduces token string length, by about 60% for all values of n. However, as our similarity algorithm is computationally cheap and most of the computation is used to build the syntax tree, there is no evidence of improvement in execution times, the other measure of efficiency.

4.6 Overall Impact of Our Proposed Technique

The preceding sections have shown the variations in effectiveness and efficiency brought about by a number of individual steps. We now combine all of these steps and compare the final process with the commonly used technique that we described in section 4.1.

Table 5: Effectiveness and efficiency of discarding terminal nodes of the syntax tree, using cosine correlation and direct node connections only; each cell gives the original value, from Table 3 or 4, followed by the new value after discarding the terminal nodes of the syntax tree

	n=1	n=2
precision	51% →52%	56% →56%
recall	85% →90% ** (3.3)	85% →86%
f-score	58% →61% ** (2.7)	63% →63%
token str lgth	355 →151 *** (-14.7)	354 →150 *** (-14.7)
exec time	144s →144s	144s →146s
	n=3	**n=4**
precision	62% →59% * (-2.1)	64% →64%
recall	83% →82%	74% →78% * (2.1)
f-score	67% →64% * (-2.1)	64% →65%
token str lgth	351 →148 *** (-14.6)	344 →145 *** (-14.3)
exec time	144s →145s	143s →140s

Table 6: Effectiveness and efficiency of our fully enhanced technique compared with the standard technique (Linear from Table 1); each value of n is compared with the standard technique

	Standard	n=1	n=2
precision	75%	52% *** (-5.8)	56% *** (-4.9)
recall	52%	90% *** (9.3)	86% *** (8.4)
f-score	55%	61%	63% * (2.2)
tok str lgth	204	151 *** (-12.2)	150 *** (-12.4)
exec time	14s	144s	146s
	n=3	**n=4**	
precision	59% *** (-4)	64% ** (-3)	
recall	82% *** (7.6)	78% *** (7.6)	
f-score	64% * (2.4)	65% ** (3.2)	
tok str lgth	148 *** (-12.9)	145 *** (-13.6)	
exec time	145s	140s	

By way of reminder, the standard technique, which we called *linear* in section 4.1, involves converting the content of each program directly to a linear sequence of tokens, comparing these token strings using a well-known string matching algorithm (RKRGST), and calculating the degree of similarity using average normalisation.

For our proposed technique we replace the linear token string with a token string derived from the syntax tree; we replace the string matching algorithm with cosine correlation, a comparison technique from the field of information retrieval; we combine tokens into n-grams; we discard n-grams whose tokens are not all directly connected, along with n-grams that include terminal nodes from the syntax tree; and we explore different values of n for the n-grams.

Table 6 shows the results for the four values of n that we have been considering. In terms of efficiency, we see a reduction of some 25% in the average length of the token string, but a tenfold increase in the execution time of the comparison process. In terms of effectiveness, all values of n beyond 1 show significant increases in the f-score, the overall measure; but these increases mask a significant drop in precision and a significant increase in recall. It appears that a value of 4 for n provides the best overall f-score, and might offer the best trade-off between precision and recall.

5 DISCUSSION

Our first research question asks how the effectiveness and efficiency of code similarity detection are affected by the use of a data structure that reflects program structure as opposed to a simple linear sequence. The answer is that use of a linearised syntax tree improves the effectiveness of the detection, at the cost of a substantial deterioration in efficiency.

Our second research question asks how the effectiveness and efficiency are affected by the use of a comparison method from information retrieval as opposed to a standard string comparison method. The answer is that using the cosine correlation method substantially improves the efficiency, although not enough to completely offset the loss mentioned in the context of the first research question. However, the approach does bring about a clear loss in

effectiveness, which we have been able to offset with various modifications to the syntax tree approach of research question 1.

In the end, our program similarity detection technique brings about a clear and significant improvement in f-score, the overall measure of effectiveness. However, this improvement entails a trade-off between precision and recall, the two individual measures of effectiveness.

Referring back to figure 2 and the definitions in section 4, a drop in precision represents an increase in false positives, program pairs that are not copied but that are suspected; while an increase in recall represents a decrease in false negatives, program pairs that are copied but are not suspected. Overall, this combination would seem appropriate for the automated step in detecting academic misconduct. The number of students who copy without being detected is reduced, but at the expense that the detection process identifies more program pairs, some of which the marker will need to discard as not actually being copied.

This overall improvement in effectiveness is offset by a clear reduction in efficiency, specifically a tenfold increase in processing time on the evidence of our test data. This can be attributed almost entirely to the extra computation involved in generating the syntax tree for each program. Nevertheless, for our purposes the cost is acceptable: we were able to process 6044 pairwise program comparisons in less than three minutes on an Intel Core i5-8350U laptop.

6 CONCLUSION

This paper introduces an approach to source code similarity detection that combines program structure, by way of syntax trees, and an information retrieval measurement in place of the more usual string comparison technique. The use of program structure leads to higher effectiveness at the cost of execution time, while the change in the similarity measurement somewhat offsets this cost in execution time.

At a more detailed level, the use of n-grams improves precision at the expense of recall, but the loss of recall can be compensated by considering only directly connected nodes and by discarding terminal nodes.

Our proposed detection technique is slow in terms of preprocessing, due to the generation of the syntax trees. However, the speed of actual comparison is high, thanks to the replacement of a quadratic comparison algorithm with a linear one, even though it does not fully compensate the inefficiency caused. This makes the proposed technique ideal for use when the preprocessing can be performed just once for each program, prior to multiple comparisons, which is how we plan to use it in future work. The detection technique will be embedded in a submission system that will compare student submissions not only within the same class, but also between classes and indeed between cohorts, to check for the copying of programs or program segments across classes and across years.

ACKNOWLEDGMENTS

To Australia Awards Scholarship for supporting the study of the first author. To CORE, the computing research and education association of Australasia, for partially funding the travel cost to Australasian computer science week 2020 and this conference. To Sendy Ferdian from Maranatha Christian University, Indonesia for providing the data set. To William Chivers from University of Newcastle, Australia for his contribution to the overall work.

REFERENCES

[1] Christian Arwin and S. M. M. Tahaghoghi. 2006. Plagiarism detection across programming languages. In *29th Australasian Computer Science Conference - Volume 48*. Australian Computer Society, Hobart, 277–286.

[2] Andrés M. Bejarano, Lucy E. García, and Eduardo E. Zurek. 2015. Detection of source code similitude in academic environments. *Computer Applications in Engineering Education* 23, 1 (Jan 2015), 13–22. https://doi.org/10.1002/cae.21571

[3] Steven Burrows, S. M. M. Tahaghoghi, and Justin Zobel. 2007. Efficient plagiarism detection for large code repositories. *Software: Practice and Experience* 37, 2 (Feb 2007), 151–175. https://doi.org/10.1002/spe.750

[4] Keith D. Cooper and Linda Torczon. 2012. *Engineering a Compiler (Second Edition)*. Morgan Kaufmann.

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

[6] Georgina Cosma and Mike Joy. 2012. An approach to source-code plagiarism detection and investigation using latent semantic analysis. *IEEE Trans. Comput.* 61, 3 (Mar 2012), 379–394. https://doi.org/10.1109/TC.2011.223

[7] W. Bruce Croft, Donald Metzler, and Trevor Strohman. 2010. *Search Engines : Information Retrieval in Practice*. Addison-Wesley. 520 pages.

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

[9] Enrique Flores, Alberto Barrón-Cedeño, Lidia Moreno, and Paolo Rosso. 2015. Cross-language source code re-use detection using latent semantic analysis. *Journal of Universal Computer Science* 21, 13 (2015), 1708–1725.

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

[11] Robert Fraser. 2014. Collaboration, collusion and plagiarism in computer science coursework. *Informatics in Education* 13, 2 (Sep 2014), 179–195. https://doi.org/10.15388/infedu.2014.01

[12] Deqiang Fu, Yanyan Xu, Haoran Yu, and Boyang Yang. 2017. WASTK: a weighted abstract syntax tree kernel method for source code plagiarism detection. *Scientific Programming* 2017 (Feb 2017), 1–8. https://doi.org/10.1155/2017/7809047

[13] J. Paul Gibson. 2009. Software reuse and plagiarism: a code of practice. In *14th Annual ACM SIGCSE conference on Innovation and Technology in Computer Science Education*. ACM Press, New York, New York, USA, 55–59. https://doi.org/10.1145/1562877.1562900

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

[15] James K Harris. 1994. Plagiarism in computer science courses. In *Conference on Ethics in the Computer Age*. 133–135.

[16] Oscar Karnalim. 2016. Detecting source code plagiarism on introductory programming course assignments using a bytecode approach. In *10th International Conference on Information & Communication Technology and Systems*. IEEE, Surabaya, 63–68. https://doi.org/10.1109/ICTS.2016.7910274

[17] Oscar Karnalim. 2019. Source code plagiarism detection with low-level structural representation and information retrieval. *International Journal of Computers and Applications* (Mar 2019). https://doi.org/10.1080/1206212X.2019.1589944

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

[19] Hiroshi Kikuchi, Takaaki Goto, Mitsuo Wakatsuki, and Tetsuro Nishino. 2014. A source code plagiarism detecting method using alignment with abstract syntax tree elements. In *15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*. IEEE, Las Vegas, 1–6. https://doi.org/10.1109/SNPD.2014.6888733

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

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

[22] Terence Parr. 2013. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf.

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

[24] Faqih Salban Rabbani and Oscar Karnalim. 2017. Detecting source code plagiarism on .NET programming languages using low-level representation and adaptive local alignment. *Journal of Information and Organizational Sciences* 41, 1 (Jun 2017), 105–123. https://doi.org/10.31341/jios.41.1.7

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

[26] Simon, Beth Cook, Judy Sheard, Angela Carbone, and Chris Johnson. 2013. Academic integrity: differences between computing assessments and essays. In *13th Koli Calling International Conference on Computing Education Research*. ACM Press, Koli, 23–32. https://doi.org/10.1145/2526968.2526971

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

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

[29] Hyun-Je Song, Seong-Bae Park, and Se Young Park. 2015. Computation of program source code similarity by composition of parse tree and call graph. *Mathematical Problems in Engineering* 2015 (Apr 2015), 1–12. https://doi.org/10.1155/2015/429807

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

[31] Farhan Ullah, Junfeng Wang, Muhammad Farhan, Sohail Jabbar, Zhiming Wu, and Shehzad Khalid. 2018. Plagiarism detection in students' programming assignments based on semantics: multimedia e-learning based smart assessment methodology. *Multimedia Tools and Applications* (Mar 2018). https://doi.org/10.1007/s11042-018-5827-6

[32] Lisheng Wang, Lingchao Jiang, and Guofeng Qin. 2018. A search of Verilog code plagiarism detection method. In *13th International Conference on Computer Science & Education*. IEEE, Colombo, 1–5. https://doi.org/10.1109/ICCSE.2018.8468817

[33] Michael J. Wise. 1992. Detection of similarities in student programs: YAP'ing may be preferable to Plague'ing. In *23rd SIGCSE Technical Symposium on Computer Science Education*, Vol. 24. ACM Press, Kansas City, 268–271. https://doi.org/10.1145/134510.134564

[34] Michael J. Wise. 1996. YAP3: improved detection of similarities in computer program and other texts. In *27th SIGCSE Technical Symposium on Computer Science Education*, Vol. 28. ACM Press, Philadelphia, 130–134. https://doi.org/10.1145/236452.236525