# Syntax-based Improvements to Plagiarism Detectors and their Evaluations

Lawton Nichols
UC Santa Barbara
lawtonnichols@cs.ucsb.edu

Kyle Dewey
CSU Northridge
kyle.dewey@csun.edu

Mehmet Emre
UC Santa Barbara
emre@cs.ucsb.edu

Sitao Chen
UC Santa Barbara
sitaochen@ucsb.edu

Ben Hardekopf
UC Santa Barbara
benh@cs.ucsb.edu

## ABSTRACT

Software plagiarism cheats students out of their own education and leads to unfair grading, making software plagiarism detection an important problem. However, many popular plagiarism detection tools are inaccurate, language-specific, or closed source, limiting their applicability. In this work, we seek to address these problems via a novel approach. We adapt the optimal Smith-Waterman sequence alignment algorithm to precisely measure the similarity between programs, greatly improving detection accuracy relative to competitors. Our approach is applicable to any language describable by an ANTLR grammar, which includes most programming languages. We also provide a new type of evaluation based on random program generation and obfuscation. Finally, we make our approach freely available, allowing for customizations and transparent reasoning about detection behavior.

**ACM Reference Format:**
Lawton Nichols, Kyle Dewey, Mehmet Emre, Sitao Chen, and Ben Hardekopf. 2019. Syntax-based Improvements to Plagiarism Detectors and their Evaluations. In *Innovation and Technology in Computer Science Education (ITiCSE '19), July 15–17, 2019, Aberdeen, Scotland UK*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3304221.3319789

## 1 INTRODUCTION

Plagiarism cheats plagiarizers out of their own education, and can lead to unfair grading of students who do not plagiarize. As such, the detection of software plagiarism is an important problem. While there is a large existing body of work on plagiarism detection (e.g., [4–6, 8, 10, 12, 14], we observe that plagiarism detection remains an unsolved problem. Specifically, existing plagiarism detection approaches tend to be inaccurrate, language-specific, or closed source [4], limiting their practicality.

Towards solving these problems, we observe the following:

- The syntax of most languages allows programs to differ in operationally indistinguishable ways, as by varying whitespace or variable names. This sort of syntactic noise is frequently exploited to obfuscate plagiarism [4].
- Most languages have features which are related to each other (e.g., both `if` and `switch` perform conditional code execution). A common plagiarism obfuscation is to substitute these features with each other [4].
- Many kinds of plagiarism obfuscations can be phrased as code additions, deletions, or modifications.

In this work, we directly exploit these observations to inform the design of a novel plagiarism detection approach. Towards removing superfluous syntactic information, we adopt a syntax-aware approach with *filtering* refinements, which strip away anything the user considers uninteresting. As a countermeasure to language feature substitution, we define *abstraction* refinements, which allow the user to specify how similar different language features are to each other. Finally, we observe that the Smith-Waterman algorithm [13], classically from bioinformatics, is well-suited to plagiarism detection; the algorithm was was specifically designed to compare sequences in the presence of additions, deletions, and modifications. The use of these refinements, in conjunction with the Smith-Waterman algorithm, leads to a plagiarism detection solution which is accurate *by design*.

While our approach is syntax-aware, it is not tied to the syntax of any particular language. We use ANTLR [7] grammars for defining program syntax, which are commonly used when defining language parsers for compilers. Most languages can be defined with ANTLR grammars, and many already have ANTLR grammars available, making our approach applicable to most languages. While our filtering and abstraction refinements require an additional time investment to specify, this investment needs to be performed only once per language, making the specification burden overall minute.

To evaluate the accurracy of our approach, we compare it against multiple existing plagiarism detection approaches (namely, MOSS [12], JPLAG [10], and Zhang and Liu [8]) on Java programs. While designing our evaluation, we discovered that many evaluations are based on simulated plagiarized programs written by the very authors of the corresponding plagiarism detection technique (e.g., [3–5, 8, 15]). We observe that subtle biases can be introduced with this evaluation approach, as it is possible for an author to subconsciously write programs which are more liable to be caught by their own technique. To reduce such bias, we base our evaluation on randomly-generated Java programs which have been automatically obfuscated using plagiarism obfuscation approaches observed in the wild (e.g., those in Martins et al. [4]). Our evaluation shows that


*ITiCSE '19, July 15–17, 2019, Aberdeen, Scotland UK*

ACM ISBN 978-1-4503-6301-3/19/07…$15.00
https://doi.org/10.1145/3304221.3319789

our approach is superior to that of all competing approaches, in terms of true/false positives/negatives discovered.

Overall, our contributions are as follows:

- We introduce a novel syntax-aware plagiarism detection approach. We explain our technique in Section 3, and provide an example in Section 4.
- We introduce a new evaluation approach based on random program generation and obfuscation, and use this evaluation approach to evaluate our plagiarism detection approach. We find that our plagiarism detection approach offers superior accurracy to that of all competitors considered. Section 5 provides further details.
- We make our plagiarism detection and evaluation approaches, along with their corresponding source code, freely available[1].

## 2 BACKGROUND AND RELATED WORK

Plagiarism is associated with malicious intent, and students who plagiarize code will often obfuscate it to avoid detection [4]. Plagiarism detection tools (hereafter referred to as "detectors") must see through these obfuscations.

Similarity-based detectors give scores to all possible pairs of programs, where higher-scoring pairs are more similar to each other (and more indicative of plagiarism) than lower-scoring pairs. Exactly what constitutes a "high score" is relative to the listing of scores; in practice, instructors would look at the code corresponding to some of the highest-scoring pairs, in order to make a judgement call on whether or not plagiarism occurred. While this still requires instructors to perform manual code inspection to find plagiarism, it dramatically reduces the number of pairs to consider, going from hundreds to thousands of pairs to perhaps several. With this in mind, the purpose of detectors in practice is to eliminate unlikely cases of plagiarism, leaving only likely cases.

### 2.1 Evaluating Plagiarism Detectors

Effective detectors will consistently give high scores to plagiarism and low scores to non-plagiarism. This suggests an evaluation strategy: see how a given detector scores known cases of plagiarism and non-plagiarism, and measure how close these scores are to expectations. Ideally, this evaluation would involve real student assignment submissions, reflecting how detectors are intended to be used. However using real submissions has a problem: students must *honestly tell us* whether or not they plagiarized. Given the negative consequences of plagiarism, along with the fact that plagiarism is intentionally obfuscated to avoid detection, students cannot be relied upon to provide this information. As such, alternative evaluation strategies are frequently used.

A common alternative strategy is to manually create benchmarks which intentionally plagiarize code [3–5, 8, 15]. We argue that this is prone to bias, as authors may subconsciously write benchmarks which behave differently on their own detector. In contrast, we generate random programs and randomly perturb them in a manner consistent with plagiarism.

[1] https://github.com/lawtonnichols/plagiarism-detector

### 2.2 Related Work on Plagiarism Detection

Only two of the papers mentioned in a recent plagiarism detection survey [4] are open source, and each lacks widespread language support. We believe our approach fills this gap.

The most successful detector is MOSS [12], and we compare our tool against it in our evaluation. The algorithm behind MOSS (namely, Winnowing [12]) is freely available, though the MOSS tool itself is closed source. While Winnowing is language-agnostic, MOSS has language-specific modes, and setting these modes properly has dramatic impact on the results (see Section 5.3). As such, MOSS's approach is neither widely applicable nor freely available, unlike our approach. MOSS, Nayayanan et al. [6], and Chilowicz et al. [1] are all based on fingerprinting at various granularity levels, with MOSS using files, Nayayanan et al. using token sequences, and Chilowicz et al. using syntax trees.

JPLAG is a Java-specific detector which applies a string tiling algorithm at the token level [10]. Son et al.'s [14] approach, like ours, is based on comparing parse trees, though their comparison is based on convolution kernels instead of sequence alignment.

Tahaei and Noelle [16] detect plagiarism by observing multiple submissions of code and comparing the differences between those submsisions via logistic regression. In contrast, our work does not require multiple submissions of student programs. Their evaluation was done on student code labeled by the instructor.

Fu et al. [2] use a version of the TF-IDF statistic to find the most "surprising" differences on abstract syntax trees. Their evaluation was on short programs, and cases of plagiarism were generated from starter code using generators made by multiple people. This evaluation is similar to our own, but the types of program transformations performed by the generators are unclear.

Prado et al. [9] introduce a detector which uses static and dynamic analyses to perform plagiarism detection, making it reliant on existing tools and analyses for the languages that it supports. In contrast, our method only requires an ANTLR grammar and minimal work to add a new language.

Sulistiani and Karnalim [15] compare token sequences, filtering them using cosine similarity for efficiency. We observe that token sequences remove some of the underlying structure of a program, and so our detector instead works with parse trees. Their evaluation consists of handmade cases of plagiarism.

Zhang and Liu's [8] approach is arguably the most similar to our own, as they also make use of the Smith-Waterman algorithm [13]. However, Zhang and Liu's approach has several key differences from that of our own: to the best of our knowledge, they convert trees to sequences via a preorder traversal instead of a postorder traversal (Section 3.1), their scoring function is less general than that of our own (Section 3.5), they do not perform sorting (Section 3.2), and most importantly, they lack our filtering and abstraction refinements (see Section 3.3). For these reasons, we have found that Zhang and Liu's approach cannot detect plagiarism as accurately as our proposed method, as our evaluation shows (Section 5.3).

## 3 OUR METHOD

Figure 1 provides a graphical view of our process. In the rest of this section, we cover each part in more detail.

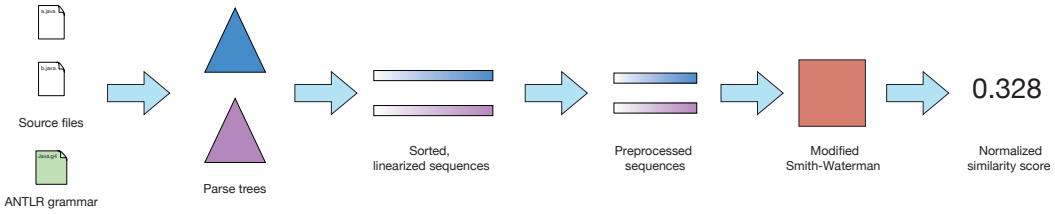



**Figure 1: A graphical overview of our method.**

## 3.1 Parsing: ANTLR Grammars to Sequences

Our method requires as input the parse tree of a program—because parse trees can be made for any programming language, every further step works exactly the same regardless of the initial language. We assume that input programs are syntactically well-formed, which is reasonably ensured via a course policy which states that only compiling programs may receive credit. To get parse trees, we use the ANTLR parser generator [7], which has existing grammars for several popular languages.

*Adding support for a new language.* With this in mind, to apply our approach to a new language, a user needs only to: (1) find (or create) an ANTLR grammar describing the language; (2) use the ANTLR tool to create a corresponding parser for the grammar; (3) write fairly straightforward boilerplate code to attach the parser to our tool; (4) provide minimal information for filtering (see Section 3.3). To keep things concrete, we focus on Java for the remainder of this paper, but any other language could be substituted in its place.

The parser from ANTLR produces parse trees corresponding to input programs. Once we obtain a parse tree for two given programs, we *linearize* the two trees by performing a postorder traversal and outputting the labels of each node traversed. Once we have the two programs that we wish to compare in this linearized form, we then preprocess them before performing sequence alignment on them. The first preprocessing step is sorting the functions.

## 3.2 Sorting

Because we are comparing entire files at a time, we must be wary of a common plagiarism operation: function rearrangement. The order of the functions should not matter when evaluating cases of plagiarism—it makes no difference if `foo()` comes before or after `bar()` if their contents are plagiarized.

The heuristic we use is to sort functions in each file by size (i.e., the number of nodes in the parse tree) before we linearize each parse tree. We found that this gives good results while avoiding the exponential blowup that would occur with trying every possible combination of functions.

After the linearized sequences are in this sorted order, we can further prune and enhance the information contained within them.

## 3.3 Filtering, Abstraction, and Weight

In this step, we determine which parse tree node labels to keep, what equivalence class they belong to (if any), and provide a relative score that indicates how important a given label is. Users must provide information specifying how labels are grouped, once per language. We explain each case along with its justification.

*Filtering.* Another common plagiarism operation is renaming variable names and other identifiers, so we cannot trust any such node in a parse tree. We have a filtering phase to get rid of these and other parse tree labels that we deem unsuitable for comparison. In fact, we found the ratio of useful to interesting nodes to be so small that we only require a list of nodes to *keep*.

As an example, consider the Java grammar rule for expressions:

```
expression
    : primary
    ...
    | methodCall
    | NEW creator
    | '(' typeType ')' expression
    | expression postfix=('++' | '--')
    | prefix=('+'|'-'|'++'|'--') expression
    ...
```

This is essentially a catch-all rule, and almost every line of Java code will contain an `expression` node in its sub-parse tree. As such, there no useful information in `expression`, and so we omit it from our output. There are many similar rules in the Java grammar.

*Abstraction.* Several nodes act in a similar way, and should be considered as such. For example, several `if`/`else` statements may be transformed into one `switch` statement, and vice versa. We therefore consider parse tree node labels for `if`/`else` and `switch` to be in the same equivalence class, and we consider them to "match" in our Smith-Waterman algorithm. Different kinds of loops also belong together in the same equivalence class.

This concludes our preprocessing steps. We are now ready to compare two program sequences with a modified Smith-Waterman algorithm. Before introducing our algorithm, we first discuss sequence comparison in general.

## 3.4 Comparing Sequences: False Start

Smith-Waterman is a relatively unfamiliar algorithm to most computer scientists, so we will briefly compare it to the more familiar and similar concept of string edit distance. Levenshtein distance is the most common string edit distance measurement algorithm.

Levenshtein distance has notions of insertion, deletion, and mismatch "costs", and the final answer is the minimized cost of performing these different string-altering operations. Overall, Levenshtein distance determines the most cost-effective way to turn one string into another. Smith-Waterman has similar computational costs, but matching is more customizable; for example, Smith-Waterman allows a *matrix* of scores, which allows a programmer to give different match/mismatch scores to different pairs of elements of a sequence. In this work, we assign higher scores to programs that match at certain key nodes, such as loops and if statements.

Levenshtein distance also computes a cost over the entire string, in what is known as a global alignment. For example, comparing the strings `A = ''cat''` and `B = ''␣␣␣cat␣␣''` results in a score of 5, because the entire string A must be transformed into the entire string B. In contrast, Smith-Waterman performs a *local* alignment and finds the largest substrings of each string that match. Local alignment is important for our purposes because we want to give a high similarity score to a pair of programs when a *portion* of one is found in the other; our method would be less accurate if we only gave high similarity scores to program pairs when they have almost everything in common.

### 3.5 Comparing Sequences: Smith-Waterman

The Smith-Waterman algorithm [13] is a dynamic programming-based sequence alignment algorithm. It computes a numerical value representing how similar or dissimilar two sequences are. Originally implemented with bioinformatics in mind, it is used to compare biological sequences (e.g., RNA and proteins) and to determine how closely two such sequences overlap each other. The algorithm works for sequences of any kind, not just sequences of characters; in this work, we align sequences of abstracted parse tree nodes.

Smith-Waterman is parameterized by a gap score (for insertions and deletions) and a scoring matrix (for matching and mismatching). This matrix is consulted each time nodes are compared to determine how similar they are, and so tuning this matrix has a major impact on the results. We simplify the process of constructing a scoring matrix by: (1) providing all nodes that match a default match score, and all nodes that do not match a default mismatch score; and (2) assigning relative weights to certain matches.

*Weights.* Experimentally, we found that some label classes are more important than others. For example, it is quite likely that several `if` statements will match in a plagiarized pair of programs, whereas assignment statements do not closely correspond. For this reason, we assign relative weights to certain equivalence classes of labels. Just like the rest of our method, these weights are completely customizable to fit any instructor's needs. The relevant portion of our weight/equivalence class file follows:

```
{"conditional": 5, "loop": 5, "return": 5,
"functioncall": 5, "assign": 2}
```

This syntax specifies that conditional statements (e.g., `if`/`else`, `switch`) that match are to be given 5× the match score. If such a node does not match, we perform the same multiplcation against the mismatch score.

*Final steps.* After obtaining a similarity score from the Smith-Waterman algorithm, we normalize the scores to be between 0.0 and 1.0. Without such a normalization, larger plagiarized program pairs (which naturally have more similar portions than shorter plagiarized program pairs) would have larger scores, rendering sorting by score useless.

## 4 COMPLETE EXAMPLE

This section goes through an in-depth example of our method.

*Description of the programs.* Figure 2 contains three example programs: A, B, and C. Programs A and B have been plagiarized using several common operations (e.g., identifier renaming, manipulation of spacing, etc.) [4], while program C is independent of A and B. We use this example to demonstrate our method, and we further elaborate on plagiarism operations in our evaluation (Section 5).

*Parsing.* We first parse the three programs, creating the parse trees shown in Figure 3. None of these parse trees look similar to each other, with differences in both the number of nodes and node structure. Our goal is to get Programs A and B into a format where they both "look" similar; it is for this reason that we preprocess and linearize the parse tree nodes.

*Sorting.* We then extract functions and sort them by their size, in terms of the number of parse tree nodes. In this example, sorting will not change the function order in Program A. However, in Program B, `func2` will be moved to come *before* `func1`. This sorting heuristic thus puts the plagiarized functions in the same order.

*Linearization.* We then perform a postorder traversal of the parse trees before we prune further. The contents, however, are the important part, and we want to get Programs A and B into a format where they have a lot in common. The first few nodes of Program A's `bar` function are (bold font indicating common elements):

**TerminalNodeImpl**, **IdentifierNonterminal**, **(**, **TerminalNodeImpl**, **)**, **TerminalNodeImpl**, FormalParameters, **{**, **TerminalNodeImpl**, **int**, **TerminalNodeImpl**, **PrimitiveType**, **TypeType**, sum, **TerminalNodeImpl**, **VariableDeclaratorId**, **=**, **TerminalNodeImpl**, 0, **TerminalNodeImpl**, Literal, Primary, PrimaryExpression, VariableInitializer, VariableDeclarator, VariableDeclarators, LocalVariableDeclaration, . . .

The first few nodes of Program B's `func1` function are:

**TerminalNodeImpl**, **IdentifierNonterminal**, **(**, **TerminalNodeImpl**, int, **TerminalNodeImpl**, PrimitiveType, TypeType, dummy1, **TerminalNodeImpl**, VariableDeclaratorId, FormalParameter, TerminalNodeImpl, **int**, **TerminalNodeImpl**, **PrimitiveType**, **TypeType**, dummy2, **TerminalNodeImpl**, **VariableDeclaratorId**, FormalParameter, FormalParameterList, **)**, **TerminalNodeImpl**, FormalParameters, **{**, **TerminalNodeImpl**, int, . . .

While there are similarities at the beginning, they do not last long. `func2`'s extra parameters and names, as well as the extra statements at the beginning, are taking up a substantial amount of space, so we want to remove them. Similarly, any identifier names are naturally untrustworthy, and are thus ripe for removal. Not shown are the nodes for the `while` and `for` loops, which are currently considered different; we want to treat these as being similar to each other.

*Pruning and Abstraction.* Beforehand, we have compiled a set of "useful" parse tree nodes which we want to keep; it is only necessary to do this once per language. In this step, we prune out any node that is not in our useful set. After performing this pruning, we are left with sequences that are typically ~33% as large as the originals. Pruning reduces both the amount of input to process downstream (improving performance), as well as the amount of irrelevant "noise" in the input (improving accurracy/precision). The final nodes of Program A's `bar` function are (bold font indicating common elements):

```java
public class A {
    public void foo() {
        for (int i = 1; i < 10; i++) {
            System.out.println(i);
        }
    }

    public int bar() {
        int sum = 0;
        for (int i = 0; i < 10; i++) {
            for (int j = 0; j < 10; j++) {
                sum += i + j;
            }
        }
        return sum;
    }
}
```

(a) Program A

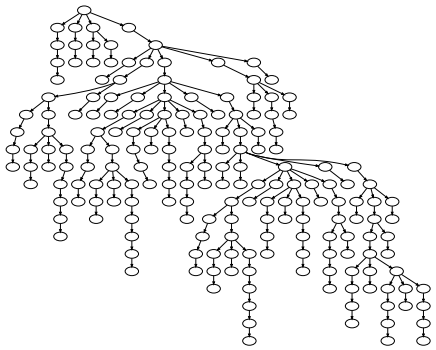
```java
public class B
{public int func1(int dummy1, int dummy2)
    {int extraStmt1 = 5; int extraStmt2 = 42;
     int s = 0; int i = 1;
        while (11 > i)
        { int j = 1; while (11 > j) {
                s += (i - 1) + (j - 1);
                j = j + 1;
            }
            ++i;
        }
        return s;
    }
    public void func2()
    {int z = 1; while (10 > z)
    {System.out.println(z); z++;}}
}
```

(b) Program B

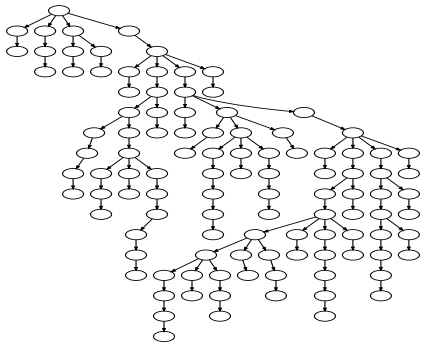
```java
public class C {
    public void hello() {
        System.out.println("Hello, world!");
    }

    public int sum2(int x) {
        int a = 0;
        for (;;) {
            if (x <= 0) break;
            a += x;
            x--;
        }
        return a;
    }
}
```

(c) Program C

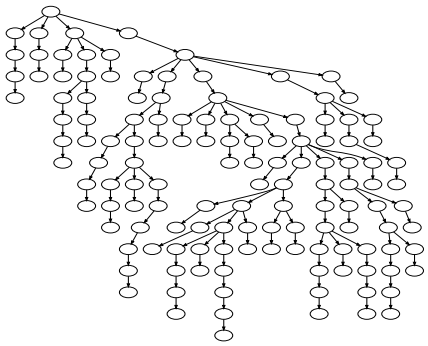
**Figure 2: Three contrived example programs: A and B for a plagiarized pair, while C has no plagiarized information taken from A or B.**



(a) Parse tree for Program A



(b) Parse tree for Program B



(c) Parse tree for Program C

**Figure 3: A bird's eye view of the parse tree structure for Programs A, B, and C.**

. . . , Primary, **Primary**, **AdditiveExpression**, **AssignExpression**, **ForLoop**, **ForLoop**, **Primary**, **ReturnStatement**

The last nodes of Program B's `func1` function are:

. . . , Literal, **Primary**, **AdditiveExpression**, **AssignExpression**, **WhileLoop**, Primary, PrefixIncDecNegPlus, **WhileLoop**, **Primary**, **ReturnStatement**

These pruned sequences share several similar elements, and sequence alignment will give a larger score on these than the originals. One final step that is not shown is abstraction, where we group similar nodes. In this example, we will end up grouping `WhileLoop` and `ForLoop` into the same equivalence class, and so we will consider them to represent the same node; this allows sequence alignment to return a larger score for these plagiarized programs.

*Smith-Waterman.* Running Smith-Waterman on the abstracted sequences is the last step. We run the algorithm for each pair of input programs with the scoring matrix generated by the supplied node weights. The scores are relative to each other, and are not meaningful on an absolute scale. Higher-scoring pairs are more indicative of plagiarism than lower-scoring pairs, and in practice users need only look at the several highest-scoring pairs to find plagiarism.

*The final scores.* After all this work has been done, we are left with a similarity score between 0 and 1 for each pair of programs:

- Program A and Program B: 0.299
- Program A and Program C: 0.193
- Program B and Program C: 0.120

In this case, our method has correctly scored the plagiarized program pair higher than the non-plagiarized program pairs.

## 5 EVALUATION

In this section we compare our method to existing methods.

### 5.1 What We Compare Against

We evaluate our method against MOSS [12] and JPLAG [10], both widely used plagiarism detection tools. We tested MOSS in Java

**Table 1: Results for each method. ↓ indicates that a lower value is better, and ↑ indicates that a higher value is better**

| Method | Cutoff | Time (s) (↓) | True Pos. (↑) | False Pos. (↓) | True Neg. (↑) | False Neg. (↓) | Precision (↑) | Recall (↑) | F-Measure (↑) |
|---|---|---|---|---|---|---|---|---|---|
| Our method | 0.15 | 115.9 | 33 | 16 | 4,884 | 17 | 0.673 | 0.660 | 0.667 |
| Zhang and Liu | 0.58 | 970.3 | 16 | 37 | 4,863 | 34 | 0.302 | 0.320 | 0.311 |
| MOSS, Java Mode | 0.44 | 30.8 | 11 | 33 | 4,867 | 39 | 0.250 | 0.220 | 0.234 |
| MOSS, Text Mode | N/A | 4.1 | 0 | 0 | 4900 | 50 | ∞ | 0.000 | 0.000 |
| JPLAG | 0.17 | 4.7 | 18 | 46 | 4,854 | 32 | 0.281 | 0.360 | 0.316 |

mode and in text mode; we did this to evaluate what would happen if MOSS was run on a language that it does not support. Both MOSS and JPLAG were run with default parameters.

We also evaluate against our implementation of Zhang and Liu's method [8]. Both our method and Zhang and Liu's method require multiple parameters, namely a tree traversal order, a match score, a mismatch score, and a gap score. We experimentally determined optimal values for each of these parameters. For our method, we use postorder traversal, a match score of 1, a mismatch score of -1, and a gap score of -2. For Zhang and Liu, we use preorder traversal, a match score of 1, a mismatch score of -1, and a gap score of -1—we also tried a gap score of -2, but -1 gave better results.

## 5.2 Benchmarks and Methodology

To address the bias problem discussed in Section 2.1, we employ random program generation and random program transformation to create our benchmark suite. We have created 50 pairs of original and plagiarized Java programs for a total of 100 Java programs, and we evaluate by comparing all pairs of those 100 programs. We implemented our generator to perform several popular plagiarism obfuscations observed in the wild [4]:

- Addition of random amounts of whitespace.
- Changing every identifier name (e.g., variable names, argument names, function names, etc.).
- Changing type names to equivalent ones (we overapproximate this by transforming type names into random strings).
- Changing operations to equivalent ones (e.g., replacing $A + B < C$ with $C > B + A$).
- Replacing control structures with equivalent ones (e.g., swapping while with for and vice versa).
- Changing the order of statements and functions.

Our metrics are the standard ones of precision (true positives / (true positives + false positives)) and recall (true positives / (true positives + false negatives)). These numbers are combined via F-measure [11], which provides a single number to compare the relative performance of each method.

## 5.3 Results

Table 1 shows the results for every method that we tried. There were 4,950 possible pairs of programs to compare, and 50 "true" cases of plagiarism; all other pairs are considered to be false positives. Each method produces a score between 0 and 1 indicating the likelihood that a pair was plagiarized. To convert this score to a binary yes/no for whether or not plagiarism was detected, we selected a cutoff value for each technique, where scores ≥ to the cutoff were considered indicative of plagiarism. Cutoff scores were picked to maximize the F-measure for each method.

*Discussion.* Our method has the best F-measure, and the running time is in the middle of the group.

Both ours and Zhang and Liu's methods start with the same parse trees and subsequently run an $O(n^2)$ algorithm. However, our method is nearly 8.4x faster than Zhang and Liu's method. This is because our filtering stage dramatically reduces the input size, minimizing $n$ in the aforementioned time complexity. It is possible that pruning obviously non-matching pairs (e.g., programs of vastly different lengths) could help to further reduce our runtime, but we leave this investigation for future work.

MOSS' text mode returned no matching results, which was to be expected; all identifiers were changed, and so there would be no meaningful matches in any sliding window of results. As such, while Winnowing [12] (the technique MOSS is based on) is language-agnostic, MOSS itself is not. While some effort is needed to apply our technique to a new language, this is fundamentally impossible with MOSS, due to MOSS' closed-source nature.

MOSS' Java mode and JPLAG both had similar results. MOSS' time includes sending the files over the network and waiting for a response from the server, so it is possible that the running time of the main plagiarism detection algorithm is closer to that of JPLAG.

*Threats to Validity.* Our method cannot detect every kind of plagiarism. For example, it cannot currently handle obfuscation where functions are broken down into many small functions, though it would be possible to mitigate this issue (at the expense of increased runtime) by comparing all function pairs. Semantic similarity is also a research-worthy challenge: it is always possible to trick a syntax-based method by swapping the code with completely different code that does the same thing (e.g., insertion sort with quicksort).

# 6 CONCLUSION

In this paper we presented a plagiarism detection method based on the Smith-Waterman sequence alignment algorithm. Our method works for any language with a grammar, and converts parse trees to a linearized form. We demonstrate that preprocessing these sequences is key; through filtering, abstracting, and sorting we are able to find more true cases of plagiarism than popular, existing methods such as MOSS. Our evaluation utilizes random program generation and plagiarism obfuscations, minimizing biases subconsciously introduced when working with handwritten benchmark suites. Program generation also allows us to obtain exact ground truth, which is not obtainable from student code. We have made our tool and its source code freely available.

In the future, we would like to investigate skipping the linearization phase; there are existing *tree* alignment algorithms, and it remains to be shown whether they are able to scale and perform as well as our current method. We may also be able to improve the runtime of our method via further input pruning.


# Acknowledgments

This work was supported by NSF CCF-1319060.

# REFERENCES


[1] Michel Chilowicz, Etienne Duris, and Gilles Roussel. 2009. Syntax tree fingerprinting for source code similarity detection. In *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*. IEEE, 243–247.

[2] 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 (2017).

[3] David Gitchell and Nicholas Tran. 1999. Sim: a utility for detecting similarity in computer programs. In *ACM SIGCSE Bulletin*, Vol. 31. ACM, 266–270.

[4] Vítor T Martins, Daniela Fonte, Pedro Rangel Henriques, and Daniela da Cruz. 2014. Plagiarism detection: A tool survey and comparison. In *OASIcs-OpenAccess Series in Informatics*, Vol. 38. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[5] Lefteris Moussiades and Athena Vakali. 2005. PDetect: A clustering approach for detecting plagiarism in source code datasets. *The computer journal* 48, 6 (2005), 651–661.

[6] Sandhya Narayanan and S Simi. 2012. Source code plagiarism detection and performance analysis using fingerprint based distance measure method. In *Computer Science & Education (ICCSE), 2012 7th International Conference on*. IEEE, 1065–1068.

[7] Terence J. Parr and Russell W. Quong. 1995. ANTLR: A predicated-LL (k) parser generator. *Software: Practice and Experience* 25, 7 (1995), 789–810.

[8] Li ping Zhang and Dong sheng Liu. 2013. AST-based multi-language plagiarism detection method. In *Software Engineering and Service Science (ICSESS), 2013 4th IEEE International Conference on*. IEEE, 738–742.

[9] Bruno Prado, Kalil A Bispo, and Raul Andrade. 2018. X9: An Obfuscation Resilient Approach for Source Code Plagiarism Detection in Virtual Learning Environments.. In *ICEIS (1)*. 517–524.

[10] Lutz Prechelt, Guido Malpohl, and Michael Phlippsen. 2000. JPlag: Finding plagiarisms among a set of programs. (2000).

[11] Yutaka Sasaki et al. 2007. The truth of the F-measure. (2007).

[12] Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. 2003. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, 76–85.

[13] T.F. Smith and M.S. Waterman. 1981. Identification of common molecular subsequences. *Journal of Molecular Biology* 147, 1 (1981), 195 – 197. https://doi.org/10.1016/0022-2836(81)90087-5

[14] Jeong-Woo Son, Seong-Bae Park, and Se-Young Park. 2006. Program Plagiarism Detection Using Parse Tree Kernels. In *PRICAI 2006: Trends in Artificial Intelligence*, Qiang Yang and Geoff Webb (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1000–1004.

[15] 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.

[16] Narjes Tahaei and David C Noelle. 2018. Automated Plagiarism Detection for Computer Programming Exercises Based on Patterns of Resubmission. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*. ACM, 178–186.