Karlsruhe Institute of Technology

Software Design and Quality

Preventing Automatic Code Plagiarism Generation Through Token String Normalization

Bachelor's Thesis of

Moritz Brödel

at the Department of Informatics
KASTEL – Institute of Information Security and Dependability

Reviewer:	Prof. Dr. Ralf Reussner
Second reviewer:	Prof. Dr.-Ing. Anne Koziolek
Advisor:	M.Sc. Timur Sağlam
Second advisor:	M.Sc. Sebastian Hahner

24. November 2022 – 21. April 2023

Abstract

Code plagiarism is a significant problem in computer science education. Token-based plagiarism detectors, which represent the state-of-the-art in code plagiarism detection, excel at identifying manually plagiarized submissions. Unfortunately, they are vulnerable to automatic plagiarism generation, particularly when statements are inserted or reordered. Therefore, this thesis introduces token string normalization, which makes the results of token-based plagiarism detectors invariant to statement insertion and reordering. It inherits token-based plagiarism detectors' high language independence and utilizes a program graph. We integrate token string normalization into the state-of-the-art token-based plagiarism detector JPlag. We show that this prevents automatic plagiarism generation using statement insertion and reordering. Additionally, we confirm that JPlag's existing capabilities are retained.

Contents

List of Figures

List of Tables

1. Introduction

In computer science education, students commonly receive take-home programming assignments to complete. These make for a more practical test of their skills than classical exams. However, there is no way to check who wrote the submitted programming code. As a result, plagiarizing someone else's work[1] is a trivial task, and many students do so [2, 9]. Since there are more and more computer science students [5], class sizes are often huge [30], which makes comparing every pair of submissions by hand infeasible for instructors [3]. For example, in a class of 500, there are more than 100.000 submission pairs. In this scenario, even a spot check of thousands of pairs is unlikely to deter students from plagiarizing, as the chance of getting caught is still relatively small.

Plagiarism detectors present a countermeasure to plagiarism. Despite their name, they do not directly detect plagiarism cases but merely assist instructors in finding them. They do this by computing a similarity score between every pair of submissions. As plagiarism is strongly associated with a high similarity score, instructors can catch most cases of plagiarism by only inspecting the relatively few pairs with high scores. Token-based plagiarism detectors present the current state-of-the-art in plagiarism detection [24, 33]. Before comparison, token-based plagiarism detectors convert source code into an intermediate representation through a process called tokenization. This representation is called the token string; it is language-independent and more abstract than the source code.

Students routinely modify plagiarized submissions to hide their plagiarism. They typically alter the code's representation but leave its structure intact [13, 12]. Examples include changing whitespace and variable names. Through tokenization, token-based plagiarism detectors only consider the code's structure and are unaffected by such shallow modifications. Students can also alter the code's structure. Examples include changing control structures and inlining methods. Token-based plagiarism detectors are vulnerable to these kinds of attacks. However, employing them proficiently enough to a plagiarized submission to reach a low *Similarity Score to the Original (SSO)* requires effort and knowledge of programming on the student's part [26]. A widespread assumption is that creating an undetected plagiarized submission is harder than simply solving the assignment [15]. Students would then have no incentive to commit this type of plagiarism.

This assumption relies on the implicit premise that students must create plagiarized submissions by hand. However, this is not the case, as proven by the development of MOSSAD [11], a tool that automatically creates plagiarized submissions undetected by token-based plagiarism detectors using dead code insertion. Students can use automatic plagiarism generators such as

[1] We regard forbidden collaboration as a form of plagiarism.

Mossad to create plagiarized submissions without any effort or knowledge of programming. Therefore, they pose a significant threat to academic integrity. We will seek to devise a defense mechanism. We will focus on the two modifications most suitable for automatic plagiarism generation. Their abstract nature allows them to be employed universally with little regard for a submission's language or features. The first is inserting dead code, which is what Mossad does; we will refer to this as simply insertion. The second is reordering subsequent statements which do not affect each other; we will refer to this as simply reordering.

Our approach is a form of normalization that makes token-based plagiarism detectors invariant to these two modifications. To retain token-based plagiarism detectors' high language independence, we normalize the token string rather than the code directly. First, we construct a program graph from the token string. We then use this graph to remove dead code. Finally, we turn the graph back into a token string; in the process, we impose a fixed order on the statements. The resulting token string is unaffected by insertion and reordering of the original code. We integrate token string normalization into the state-of-the-art [33] token-based plagiarism detector JPlag [27]. We construct an automatic plagiarism generator using insertion and reordering ourselves and show that while the original version of JPlag is vulnerable, ours is not. Additionally, we confirm that we retain JPlag's existing capabilities regarding runtime and (lack of) false positives.

2. Foundations

2.1. Code Plagiarism

We already mentioned that some students attempt to hide their plagiarism by modifying plagiarized submissions. If they explicitly intend to deceive a plagiarism detector, we refer to such an action as an *obfuscation attack* on the detector. Students can alter either the code's representation or its structure. We call the first type of modification lexical and the second structural [15]. Lexical modifications are largely language-independent, requiring little semantic knowledge about the program. Structural modifications, in contrast, are language-specific and usually require much more semantic knowledge about the program. Generally, lexical modifications are comparatively straightforward, making students more likely to use them. Relatedly, development tools like IDEs typically have mature support for automating lexical modifications, such as renaming all occurrences of a variable. On the other hand, automation of structural modifications is usually much more limited.

Now, we will look at a typical lifecycle for a plagiarized submission. First, submissions are automatically tested as soon as they are handed in. If we assume the original submission's correctness, the plagiarized submission fails this step if its functionality differs. Therefore, a plagiarized submission must have the same functionality as the original. No modification may change it. After, as soon as all submissions have been handed in, an automatic plagiarism detector is run. The instructors manually inspect each pair with a high score. If the plagiarized submission has a high SSO, the instructors manually inspect the pair and discover that the student plagiarized. The plagiarized submission must have a low SSO to prevent this. Submissions plagiarized by hand usually fail this way. Finally, a reviewer looks at the submission in isolation to grade it. If they consider it suspicious, they may check its similarity to other submissions by hand, eventually discovering that the student plagiarized. Therefore, a plagiarized submission must look unsuspicious on its own. In conclusion, an obfuscation attack must result in plagiarized submissions with unchanged functionality, a low SSO, and high readability to be successful.

2.2. Token-based Plagiarism Detection

Since students often apply lexical modifications to plagiarized submissions to obfuscate their plagiarism, a good plagiarism detector should not be affected by them. Token-based plagiarism detectors work by this principle: Through tokenization, they strip the code of its representation, only extracting its structure. All submissions with the same structure result

Source Code	**Token String**
`void printSquares() {`	METHOD_BEGIN
`    int i = 1;`	VAR_DEF
`    while (i <= 10) {`	WHILE_BEGIN
`        int square = i * i;`	VAR_DEF
`        System.out.println(square);`	APPLY
`        i++;`	ASSIGN
`    }`	WHILE_END
`}`	METHOD_END

Table 2.1.: An example of JPlag's tokenization. We will use the method on the left as our running example throughout the thesis.

in the same token string, regardless of representation. We can see this as a sort of "bucket principle", as all submissions with the same structure are put in the same metaphorical bucket. Lexical modifications do not change the bucket a submission lands in and are thus ineffective. The two plagiarism detectors most cited in the literature, JPlag and Moss, use tokenization [24]. They differ in how they compare the resulting token strings, however. JPlag uses string tiling [27], while Moss uses fingerprinting [29].

Since we will be working with JPlag, we will describe it in more detail in the following. JPlag starts by turning the input submissions into token strings through tokenization. Depending on the language, tokenization is done by either a parser or a scanner. It is the only part of JPlag that is language dependent. See Table 2.1 for an example. Note that generally, a statement may result in more than one token. After tokenization, JPlag compares all pairs of token strings using Running Karp-Rabin Greedy String Tiling [36]. This efficient string comparison algorithm essentially tries to cover as much as possible of one string with the other. JPlag then outputs the coverage, meaning how much was covered as a percentage. A parameter called the *minimum match length* influences this calculation: JPlag only considers covered sequences longer than the minimum match length when calculating the coverage. The coverage is interpreted as the similarity score between two submissions. The results can be viewed in a GUI. There, pairs of submissions are shown in a side-by-side viewer. Matches are highlighted. [27]

Given that the original submission's functionality needs to be preserved, the primary mechanism by which obfuscation attacks succeed against token-based plagiarism detectors is by breaking up longer matches into multiple shorter ones. All resulting matches shorter than the minimum match length are then ignored. The SSO of the plagiarized submission is thus lowered. If this is done often enough, the SSO will eventually be relatively low, and the pair will likely evade manual inspection and remain undetected.

2.3. Program Dependence Graph

The program dependence graph (PDG) [14] is a standard program representation. It can be used for optimization and program slicing [34]. In this graph, the statements of a program constitute the nodes. The edges represent dependencies between statements. They are split into data and control dependencies.

A statement T is data dependent on a statement S iff S can directly control the value of a variable T uses. More precisely, all the following conditions have to be met:

- T may be executed after S.
- There exists some variable which S writes and T reads.
- There is no statement S' always executed between S and T with the above properties.

A statement T is control dependent on a statement S iff S directly controls whether T is executed. More precisely, all the following conditions have to be met:

- T may be executed after S.
- T is not always executed after S.
- There is no statement S' always executed between S and T with the above properties.

For an example of a PDG, see Figure 2.1. Note that the PDG does not preserve information about statement order.

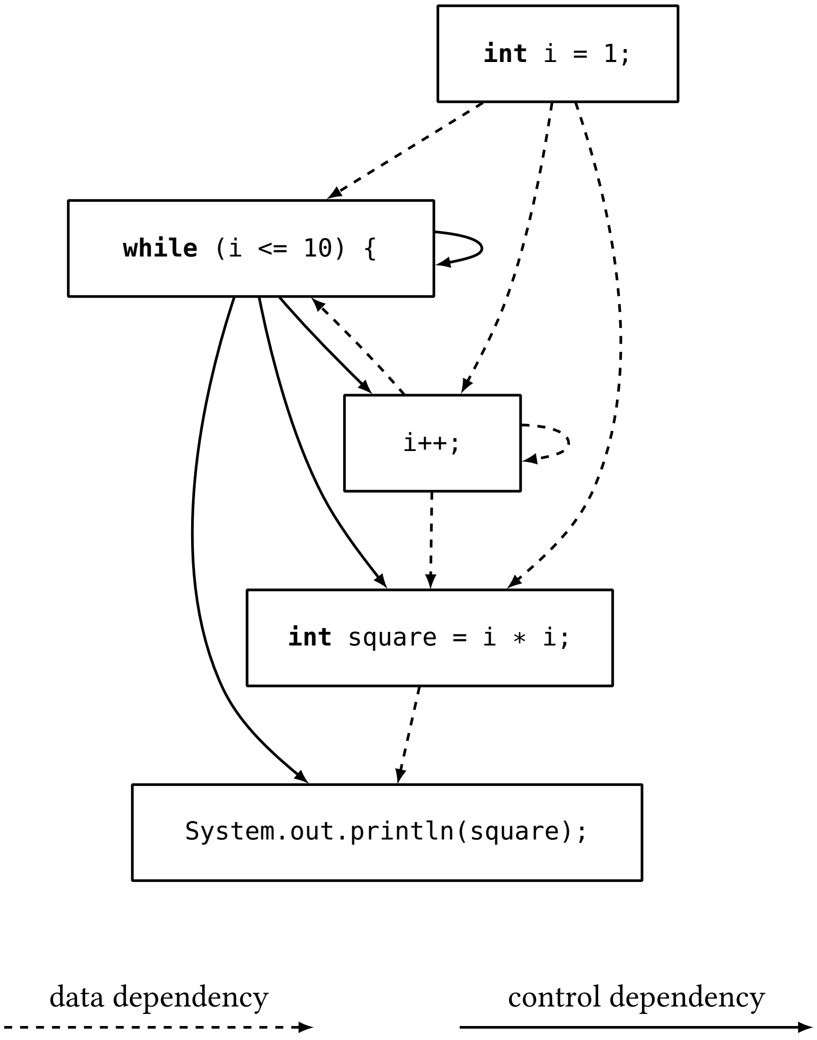

Figure 2.1.: A PDG of the example program from Table 2.1.

3. Related Work

3.1. Plagiarism Detection

In his bachelor's thesis, Krieg [19] sets out to make token-based plagiarism detectors resilient to attacks by Mossad. To this end, he introduces three mechanisms to prevent insertion from lowering the SSO. The first is generic token filtering, which checks whether two token strings that are not similar could become similar by removing some tokens from one string. The second mechanism is deleting unused variables, which he accomplishes using a compiler. The final mechanism is deleting unreachable code using a state machine that works on the token string. Token string normalization promises to be a more general and language-independent method to remove dead code than the latter two mechanisms. It can also guard against obfuscation attacks using reordering.

In a prescient 2006 paper, Liu et al. [20] point out the problems with existing plagiarism detectors, stating that "disguises like statement reordering, replacing a while loop with a for loop, and code insertion can effectively confuse these tools". They even go so far as to provide a "Plagiarism Recipe", noting that it may be possible to automate the process. This can be seen as a precursor to Mossad. They set out to develop a more robust tool. Their approach involves constructing PDGs from the source code and comparing them using subgraph isomorphy. As determining subgraph isomorphy is NP-complete, it seems plausible that GPlag's runtime is too high for practical use despite the search space pruning described in the paper. Unfortunately, GPlag is not publicly available, so no independent assessments exist. Our approach differs in two crucial ways. For one, we do not construct the PDG directly from the source code as in GPlag, providing much greater language independence. Secondly, we do not compare the graphs directly, leading to a (presumably) much faster runtime. Token string normalization using graphs can be seen as a "best of both worlds" approach combining the robustness of graph-based approaches with the fast runtime and language independence of token-based approaches.

Program behavior may be the aspect most robust to obfuscation attacks. Several papers attempt to use this to their advantage. In the most recent one, Cheers et al. [8] present the plagiarism detector BPlag. It uses symbolic execution to extract behavior from source code. BPlag calculates a similarity score by comparing graphs representing the extracted behavior. Symbolic execution and graph comparisons are both computationally expensive. As a result, BPlag's runtime is too high for practical use, much like GPlag's. Along with BPlag's implementation, the authors distribute a notice acknowledging this: "BPlag is computationally complex, requires lots of RAM and disk space, and does not scale to large data set sizes. It should not be used on conventional computers - HPC workstations only." [7] Again, as with GPlag, our approach differs as we do not compare graphs directly. We also do not consider program behavior nor work with bytecode.

A program's bytecode may better reflect its behavior than the source code. For this reason, Karnalim [18] uses bytecode to identify code plagiarism in an academic context. He uses techniques that could be seen as normalization forms, such as linearizing method contents. However, he does not use graphs as we do and works with bytecode, which we do not. In a business context, program source code is often unavailable, and plagiarism detectors have no choice but to work on bytecode. Chae et al. [6] attempt to tackle this problem by comparing the sequence and frequency of API calls in bytecode. They do so with a novel graph which they turn into a vector by randomly walking the graph. Token string normalization is somewhat reminiscent of this approach; we also construct a program graph and reduce it for efficient comparison. However, the similarities end there, as we neither work with bytecode nor consider API calls. Our program graph is also entirely different. Zhang et al. [37] consider the same problem. Their approach to solving it, which they call *LoPD*, defies the prevailing plagiarism detection paradigm. Instead of checking for similarities between programs, they check for dissimilarities. If none can be found, one of the programs is likely plagiarized. They check for dissimilarities by comparing the computation paths for specific inputs. While paths are related to graphs, LoPD differs entirely from token string normalization. Again, token string normalization does not consider bytecode; more importantly, we do not consider program input or runtime behavior.

3.2. Clone Detection

Similar code sections, or clones, are common in large coding bases. Programmers create them by accident or through copy-pasting. Juergens et al. [16] confirm that such code clones significantly impede modern software development. They do so by finding inconsistent clones. Code clone detection is a related but distinct field to plagiarism detection [22]. While in both, the aim is to identify similar code sections, there are fundamental differences. In plagiarism detection, the aim is to quantify how likely a code base is to have been plagiarized from another to protect academic integrity. An adversary may intentionally use obfuscation attacks to create the plagiarized version. In contrast, code clone detection aims to point out similar code sections within a code base created unintentionally to ease development.

Wang et al. [31] seek to specifically detect what they call *large-gap* clones, meaning clones with big differences. The tool they present, CCAligner, does this by tokenizing source code and finding fuzzy matches through a novel *e-mismatch index*. CCAligner works similarly to JPlag, though JPlag only considers exact matches. Of course, the main difference is that CCAligner aims to detect clones, whereas JPlag aims to detect plagiarism cases. The same applies to token string normalization. Additionally, our procedure is entirely unlike CCAligner's. White et al. [35] present a machine-learning approach to clone detection. They use a neural network to turn source code into vectors; similar vectors correspond to code clones. Our approach differs because we do not use machine learning or aim to detect clones. In his master's thesis, Ly [21] aims to improve clone detectors by normalizing source code using PDGs. Procedurally, token string normalization works the same. However, there are crucial distinctions. For one, again, the objective is entirely different. Furthermore, we normalize the token string rather than the source code. Token string normalization is much more language-independent than his approach as a result.

3.3. Normalization

While Ly's normalization comes closest to our approach, normalization is a standard pattern in code comparison. In general, we can differentiate two types of normalization.

The first type, called *lexical normalization*, makes code comparisons invariant to lexical modifications. It is relatively straightforward, as program semantics need not be considered. Roy and Cordy [28] use lexical normalization to detect code clones. They "ignore editing differences" so that lines of code can be compared textually; they rename all identifiers to `id`, for example. Allyson et al. [1] seek to improve a text-based plagiarism detector for code plagiarism detection using lexical normalization. They do this with several preprocessing techniques, removing whitespace, for example. JPlag's tokenization step is a form of lexical normalization. For this reason, token string normalization does not need to consider lexical modifications, markedly differentiating it from the works above.

Instead, token string normalization is a form of what we call *structural normalization*. This type of normalization makes code comparisons invariant to structural modifications. Program semantics must be considered here, so structural normalization is generally more complex than lexical normalization. It may be more accurate to view structural normalization as a type of program rather than code normalization, as code is not directly considered. Wang et al. [32] use structural normalization with the aim to simplify program analysis by bringing code into a consistent form. They use the system dependence graph, a generalization of the PDG. They consider this graph the result of the normalization and do not turn it back into source code. This presents the first difference to token string normalization. Another difference is, once again, that we do not construct our graph directly from the source code. We also have the explicit goal of detecting plagiarism cases. Besides detecting plagiarism and clones, program comparison is also used to detect malware. Like plagiarism detection in a business context, malware detection must work with bytecode, as source code is usually unavailable. Bruschi et al. [4] use structural normalization for malware detection. They consider their normalization a form of optimization. They apply several normalizing transformations to programs, such as dead code removal. From the resulting normalized programs, they construct graphs. Like previously mentioned approaches, they compare graphs directly, which serves to differentiate token string normalization. Once more, token string normalization is also different because we do not work with bytecode.

4. Automatic Plagiarism Generation

In a 2020 paper, Devore-McDonald and Berger [11] outline MOSSAD, an automatic plagiarism generator inspired by genetic programming using dead code insertion. As its inputs, MOSSAD takes a submission, an entropy file containing statements to insert, the plagiarism detector that should be tricked, and a target SSO for the plagiarism detector. First, the tool randomly chooses a statement from either the entropy file or the submission itself. MOSSAD then inserts this statement in a random position in the submission. Next, the tool compiles the submission to bytecode with optimizations to check whether the inserted statement is dead. Because the compiler removes dead code when optimizing, this is the case if the resulting bytecode has not changed. If it is different, or the submission does not compile, the statement insertion is unsuccessful, the submission containing it is discarded, and MOSSAD tries again. Otherwise, the tool considers the statement insertion successful. MOSSAD proceeds by calculating the plagiarism detector's SSO. If it is below the target SSO, MOSSAD outputs the plagiarized submission. In this case, it saves all the successfully inserted statements to the entropy file. Otherwise, the tool repeats the process. There is a timeout period after which MOSSAD terminates even if the target SSO has not been reached. In this case, it has failed to produce a plagiarized submission.

MOSSAD takes advantage of the fact that many plagiarism detectors are publicly available. It can thus ensure a low SSO for the plagiarized submissions it produces. Additionally, MOSSAD never changes a submission's functionality. As a result, MOSSAD can fail only in one of the three ways described in section 2.1: If the reviewer is suspicious of the plagiarized submission when looking at it in isolation. For MOSSAD, we can imagine a simple heuristic for when this might be the case. If the plagiarized submission is more than twice as long as the original, over half the statements are inserted dead code. As these statements are strewn randomly throughout the submission, there is no reasonable explanation. Any reviewer would be suspicious. It can also occur that MOSSAD does not manage to produce any plagiarized submission. This presents a distinct fourth type of failure. A student may fail this way by giving up on plagiarizing.

In section 2.2, we noted that obfuscation attacks on token-based plagiarism detectors aim to break up matching code segments. This can only be done with structural modifications, as token-based plagiarism detectors' results are invariant to lexical modifications. Most structural modifications work only narrowly and are dependent on the language. One example of this is changing a **`for`** loop to a **`while`** loop. This modification confuses JPlag but is constrained to languages with **`for`** and **`while`** loops. More importantly, employing this attack alone is unlikely to break up many code blocks. Instead, it must be part of an expansive, language-dependent repertoire of structural modifications that would be applied together to break up most of a submission's code blocks. Devising such a repertoire would require considerable effort and would likely be an unattractive option for anyone creating an automatic obfuscation tool. It is also outside the scope of this thesis. Instead, we will constrain ourselves to more abstract modifications, which restructure submissions on a high level, leaving them unchanged for the most part. There are two such modifications. One is to insert statements that do not affect program functionality. We will refer to such statements as dead code. The other is to reorder subsequent program statements. In order to preserve program functionality, these statements must not affect each other. We will refer to them as independent (of each other).

4.1. Dead Code Insertion

Here, blocks are broken up by inserting small blocks of dead code. See Figure 4.1 for an example of how insertion works. MossAD works by using insertion. Ideally, the blocks consist of a single statement since that is enough to break up code segments, and more statements are more suspicious. Unfortunately, MossAD does not abide by this principle and often inserts statements subsequently. We will refer to this behavior as *bloating*, as it does not meaningfully reduce the similarity but merely increases the program's size. MossAD could be improved in this regard.

Any statements MossAD inserts come from the current submission, a previously plagiarized submission, or a small pool of typical statements. Especially with its memory regarding past runs, this approach does well in inserting diverse statements into the submission. However, all three statement sources have their downside.

- Statements from the current submission have a high chance of appearing relevant in the spot they are inserted. However, they are prone to changing the submission's functionality, making them relatively difficult to insert successfully. They are also relatively small in number, which makes it hard to insert enough of them.

- Statements from previously plagiarized submission are less prone to changing the submission's functionality and are much more numerous. However, they are far less likely to appear relevant, making them generally more suspicious. For example, a previously plagiarized submission using a graph might contain the statement `int ingoingEdgeCount = 0`. This statement might be inserted into the current submission, which has nothing to do with graphs. It would appear suspicious to a reviewer and might cause them to raise the alarm when a more relevant-seeming statement would not.

- Statements from a small pool of typical statements are both unlikely to change the submission's functionality and likely to appear relevant. It is relatively easy to devise a countermeasure against their insertion, however. If a plagiarism detector were to get a hold of the pool, it could simply count their occurrences when analyzing a submission and raise the alarm if there are too many.

Source Code after Insertion

```
void printSquares() {
    int i = 1;
    while (i <= 10) {
        boolean done = false;  // inserted
        int square = i * i;
        System.out.println(square);
        i++;
    }
}
```

Token String after Insertion

Method_Begin
Var_Def
While_Begin
Var_Def
Var_Def
Apply
Assign
While_End
Method_End

Original Token String

Method_Begin
Var_Def
While_Begin
Var_Def
Apply
Assign
While_End
Method_End

Figure 4.1.: An example of how insertion breaks up blocks. The highlighted matches are now much shorter than before.

An inserted statement would ideally be unlikely to change the submission's functionality, appear relevant, and not be taken from a pre-determined pool. An advanced plagiarism generator could generate such statements by analyzing the current submission and mutating its statements. Exploring this approach is best left for future work, however, as our focus is not on how suspicious a submission might appear to a reviewer. For now, combining these three sources of statements looks like the best option. As they all have different weaknesses, they complement each other.

Reviewer suspiciousness aside, insertion works very reliably. Even if we accept the restriction that there may be no two subsequent inserted statements, statement insertions are all but unlimited. If we randomly choose two subsequent statements from a program, there likely exists some statement whose insertion between the two does not change the program's functionality. For Java, there are some notable exceptions, however. Firstly, the compiler forbids unreachable code. For example, nothing may come directly after a `return`. However, this is of little consequence since inserting a statement before a `return` has almost the same effect as inserting one after. The same cannot be said for the second exception, control statements without braces. When omitting the braces on a control statement, only the following statement is affected by it. This case is more important than the first since it can lead to long sections where no statements may be inserted. It can occur with nested `for` loops:

```
for (int row = 0; row < matrix.length; row++)
  for (int col = 0; col < matrix[0].length; col++)
    System.out.println(matrix[row][col]);
```

It can also occur when `if`/`else` is used:

```
if (cond)
  foo()
else if (otherCond)
  bar()
else
  qux()
```

For plagiarism generators, such sections present a problem. They cannot be broken up using statement insertion, so plagiarism detectors will consider them the same between the original and the plagiarized submission. As a result, the pair will have a suspiciously high similarity score if there are enough such sections, leading to manual inspection and discovery. A possible workaround is to force braces on control statements in a pre-processing step, as some linters can do.

An upside of using insertion is that little semantic knowledge of the program is needed. Plagiarism generators only need to remove dead code from the plagiarized submission and check if the functionality has changed. Of course, the more sophisticated the dead code detection is, the less noticeable the insertions will be. However, in principle, it works even if a tool can only detect dead code rudimentarily. In practice, plagiarism generators outsource the dead code detection to a compiler. Dead code removal is a common optimization feature. For example, MOSSAD uses the GNU Compiler Collection (GCC) with optimizations enabled to detect dead code.

4.2. Independent Statement Reordering

Here, blocks are broken up by reordering subsequent statements that do not affect each other, or independent statements for short. See Figure 4.2 for an example of how reordering works. We can limit ourselves to only considering swaps of two subsequent statements. Through concatenation, any reordering of an arbitrary number of subsequent statements may be produced this way. To the author's knowledge, this method has not been investigated before in the context of automatic plagiarism generation. Its notable upside is that it is all but unsuspicious to the plagiarized submission's reviewer. Statement order is often a stylistic choice, and no permutation may be more suspicious than any other. The price to pay for this quality is that generating plagiarized submissions through reordering is relatively tricky. For one, the number of possible swaps is quite limited. Swapping two randomly chosen subsequent statements of a program will usually change the program's functionality. In addition, deciding if two statements are independent of each other requires a great deal of semantic knowledge about the program. Like dead code detection, this can be done rudimentarily, and the method still works in principle. However, doing so further limits the number of possible swaps, which was relatively small in the first place. There are also no known existing tools that can accomplish this task, further complicating matters. Compilers will typically leave the statements of a program in their original order and will only reorder lower-level instructions for the sake of optimization.

Source Code after Reordering

```
void printSquares() {
    int i = 1;
    while (i <= 10) {
        int square = i * i;
        i++;  // reordered
        System.out.println(square);  // reordered
    }
}
```

Token String after Reordering	**Original Token String**
METHOD_BEGIN	METHOD_BEGIN
VAR_DEF	VAR_DEF
WHILE_BEGIN	WHILE_BEGIN
VAR_DEF	VAR_DEF
ASSIGN	APPLY
APPLY	ASSIGN
WHILE_END	WHILE_END
METHOD_END	METHOD_END

Figure 4.2.: An example of how reordering breaks up blocks. The highlighted matches are now much shorter than before.

5. Token String Normalization

The results of token-based plagiarism detectors are invariant to lexical modifications of the input submissions. This is due to tokenization, as all variants produced through lexical modifications result in the same token string. In section 2.2, we called this the "bucket principle": The submissions are put into (metaphorical) buckets; all submissions with the same structure land in the same bucket. We will expand upon this concept to defend against automatic plagiarism generation using insertion and reordering. In particular, we will make a submission's token string invariant to insertion and reordering by removing dead code and putting subsequent independent statements in a fixed order. As a result, automatic plagiarism generators using insertion and reordering can no longer generate plagiarized submissions with a low SSO. We call this process *token string normalization.*

We normalize the token string in an additional step between tokenization and comparison. As the input, we receive a submission's token string representation and some additional semantic information about it. We generate this semantic information during tokenization along with the tokens. We use it to construct a program graph. Each node in this graph represents one of the submission's statements. Note that throughout the rest of the thesis, we will use "statement" to refer to the smallest unit of a program. Certain structural elements qualify as statements in our usage, such as a brace that closes a block. After construction, we remove nodes representing dead code from the program graph. We then turn the it back into a token string. During this process, we sort all subsequent independent statements by their tokens to put them in a fixed order. Generating the additional semantic information required to construct the program graph is the only part dependent on the language. In the following, we will first detail this information. After, we will specify the program graph and describe precisely how it is used. Finally, we will normalize an example token string to illustrate the process.

5.1. Required Semantic Information

A submission's token string carries relatively little semantic information about the program. It only tells us what types of tokens were produced from the submission as well as their order. This information is insufficient to construct a meaningful program graph. For this purpose, more semantic information about the program is required. We extract this information along with the tokens during tokenization. This way, we can utilize the existing parser backend that generates the tokens.

We need six pieces of semantic information about the program to normalize the corresponding token string:

Token-Statement Grouping We need to know which tokens belong to the same statement. The statements are the smallest unit we will be working with during normalization. The tokens within each statement never change. The statements constitute our program graph's nodes.

Statement Sequence We need to know which order the statements had in the original program. This information is required to generate the graph's edges.

Statement Cruciality We need to know which statements are crucial to the program's functionality, or just *crucial* for short. We define the cruciality of a statement S as follows: Imagine we replace S with another statement that writes[1] variables the same but otherwise does nothing. If this changes the program's functionality, S is crucial; otherwise, it is not. Colloquially, we might say that crucial statements do "more" than only write variables. They directly affect the program's functionality. Examples of crucial statements are (most) class declarations, method calls, and control structures such as `if` conditions and `while` loops. Some of these may not be crucial, an empty `while` loop, for example, but such exceptions are relatively rare. Examples of statements that are not crucial are declarations of local variables and constant assignments to them. We use statement cruciality to remove dead code.

Statement Position Significance We need to know in which way the position of each statement relative to other statements is *significant*. Two statements' relative position is significant if changing it alters the program's functionality. Again, as with statement cruciality, we do not consider variables. For each statement, there are three options for its position significance:

- **None** The statement's position relative to other statements is not significant. Examples of statements with position significance "None" are attribute declarations.
- **Partial** The statement's position relative to other statements with partial position significance is significant. Examples of statements with position significance "Partial" are method calls that never result in an exception.
- **Full** The statement's position to all other statements is significant. Examples of statements with position significance "Full" are method signatures and `return` statements.

We use statement position significance to construct the graph's position significance edges. They allow us to turn the graph back into a token string.

Statement Variable Accesses We need to know which variables each statement accesses and what the access types are. There are two access types, read and write[1]. Each variable

[1] We regard declaring a variable as writing it.

needs a unique identifier so we can trace it throughout the program. This identifier needs to be generated during tokenization. The variable's name is insufficient as there may be different variables with the same name in a program. Statements that write variables are most commonly assignments. However, method calls may also write variables. In contrast, variable reads can occur in almost any context. One example is an `if` condition comparing a variable with a constant. We use statement variable accesses to construct the graph's variable edges. They allow us to remove dead code and turn the program graph back into a token string.

Bidirectional Blocks We need to know which statements belong to the same *bidirectional block*. A bidirectional block is a block of statements within which the execution order of statements can differ from their order in the statement sequence. In contrast, statements not in bidirectional blocks are always executed in sequence. Bidirectional blocks are typically loops. We use information about bidirectional blocks to remove dead code.

We can infer some of this information from the token string, most notably the statement sequences. Additionally, we could use the token types to infer information. We cannot infer everything, however, especially statement variable accesses. Therefore, we must explictly pass some semantic information along with the token string to the normalization.

One of our priorities with token string normalization is retaining token-based plagiarism detectors' existing capabilities. For this reason, we want to ensure that each submission's functionality is unchanged by normalization. In particular, we only remove code that we are certain is dead and only put subsequent statements in a fixed order that we are certain are independent. We generate the semantic information about the program in a way that enables this behavior. Specifically, we interpret the above conditions through the lens of possibility. For example, any statements that could be crucial we mark as crucial. Similarly, if a statement could have full position significance, we mark it as having full position significance. These expected inaccuracies allow us to generate the required semantic information about the program relatively seamlessly during tokenization. Indeed, this is one of the strengths of our approach.

5.2. Normalization Graph

We use a novel program graph called the *Normalization Graph (NG)* to normalize the token string. It extends the program dependence graph, which already supports dead code removal. The NG introduces additional edges, allowing us to turn it back into a token string.

5.2.1. Definition

A program's statements constitute the NG's nodes. The NG's edges are directed. They are divided into variable edges and position significance edges. In the following, S and T are different random statements. Variable edges are related to the variable accesses in statements. They are either variable flow edges or variable order edges.

- Variable flow edges are similar to the data dependencies in the PDG. We use them to remove dead code. There is a variable flow edge from S to T iff S sets the value of a variable T uses. To be more precise, there must be some variable S writes and T reads. Additionally, S must either come after T in the sequence of statements, or S and T must be in the same bidirectional block.

- Variable order edges do not have an equivalent in the PDG. We use them to turn the NG back into a token string. There is a variable order edge from S to T iff changing the order of S and T would change the program's functionality because of a variable. To be more precise, a variable must be accessed in S and T, and at least one of these accesses must be a write. Additionally, T must come after S in the sequence of statements.

Position significance edges are related to statements' position significance. They can be seen as a generalization of the PDG's control edges. Like variable order edges, we use them to turn the NG back into a token string. Position significance edges are either full position significance edges or partial position significance edges.

- There is a full position significance edge from S to T iff the relative position of S and T must stay the same because of full position significance. To be more precise, either S or T must have full position significance. Additionally, T must come after S in the sequence of statements. There must also be no statement between S and T in the sequence of statements with full position significance.

- There is a partial position significance edge from S to T iff the relative position of S and T must stay the same because of partial position significance. To be more precise, both S and T must have partial position significance. Additionally, T must come after S in the sequence of statements. There must also be no statement between S and T in the sequence of statements with partial position significance.

Note that if S comes after T in the sequence of statements, any edges from S to T are variable flow edges.

5.2.2. Usage

First, we remove dead code using statement cruciality and variable flow edges. Specifically, we remove all statements from which no crucial statement can be reached using only variable flow edges. Such statements are neither crucial nor write a variable used in a crucial statement. They do not contribute to the program's functionality, neither directly nor indirectly (through a variable). After, we remove all variable flow edges from the NG, as they have served their purpose. The graph is now acyclical, as any edge from S to T implies that T came after S in the statement sequence. Recall that only variable flow edges did not abide by this principle.

Token String		Statements
Method_Begin	1	`void printSquares() {`
Var_Def	2	`    int i = 1;`
Var_Def	3	`    boolean debug = false;  // inserted`
While_Begin	4	`    while (i <= 10) {`
Var_Def	5	`        int square = i * i;`
Assign	6	`        i++;  // reordered`
Apply	7	`        System.out.println(square);  // reordered`
While_End	8	`    }`
Method_End	9	`}`

Table 5.1.: The example token string and the statements the tokens represent. For the original submission see Table 2.1.

We now use topological sorting [17] to turn the NG back into a token string. Specifically, we put the nodes in the order of their distance from a root. A root is a node without ingoing edges. As the NG is now acyclical, at least one such root exists. Nodes with the same distance represent subsequent independent statements. We sort them by the tokens they contain. It does not matter how exactly, only that the sorting is deterministic. We can now assemble the normalized token string by going through the nodes in order and concatenating the tokens from each. Note that if two subsequent independent statements contain the same tokens, their order does not affect the normalized token string; therefore, we can consider them equal in order.

5.3. Example

To illustrate the normalization process, we will normalize the example token string shown in Table 5.1. It represents a plagiarized submission created by an automatic plagiarism generator. One statement has been inserted and two reordered. Note that token string normalization cannot directly access program statements; we only show them for a better understanding.

We will first specify the semantic information for this example submission. Statement-token grouping and statement sequence are apparent from Table 5.1. Statement 7 is crucial because a method is called in it. Statements 1, 4, 8, and 9 are crucial and have full ordering because they begin or end a code block. Statements 2 and 6 write the variable `i` and statements 4, 5, and 6 read it. Statement 5 writes the variable `square` and statement 7 reads it. Statements 4 through 8 are in the same bidirectional block.

Using this information, we construct the Normalization Graph; see Figure 5.1. We remove the node representing statement 3 as no crucial statement can be reached from it using only variable flow edges. After, we remove all variable flow edges to make the graph acyclical; see Figure 5.2. Each node's height represents its distance from a root. In this case, the only

Statements in Order	**Normalized Token String**
`void printSquares() {`	METHOD_BEGIN
`    int i = 1;`	VAR_DEF
`    while (i <= 10) {`	WHILE_BEGIN
`        int square = i * i;`	VAR_DEF
`        System.out.println(square);`	APPLY
`        i++;`	ASSIGN
`    }`	WHILE_END
`}`	METHOD_END

Table 5.2.: The statements in the order the nodes representing them have in the normalization graph after dead code removal and the normalized token string assembled from them.

root is the node representing statement 1. The only two nodes with the same distance from it are those representing statement 6 and 7, respectively. We sort them alphabetically by their token types, so the node representing statement 6 comes after the one representing statement 7. Finally, we go through the nodes in order and get the tokens from each to receive the normalized token string; see Table 5.2. Note that the normalized token string from the plagiarized submission is the same as the token string from the original submission (compare with Table 2.1).

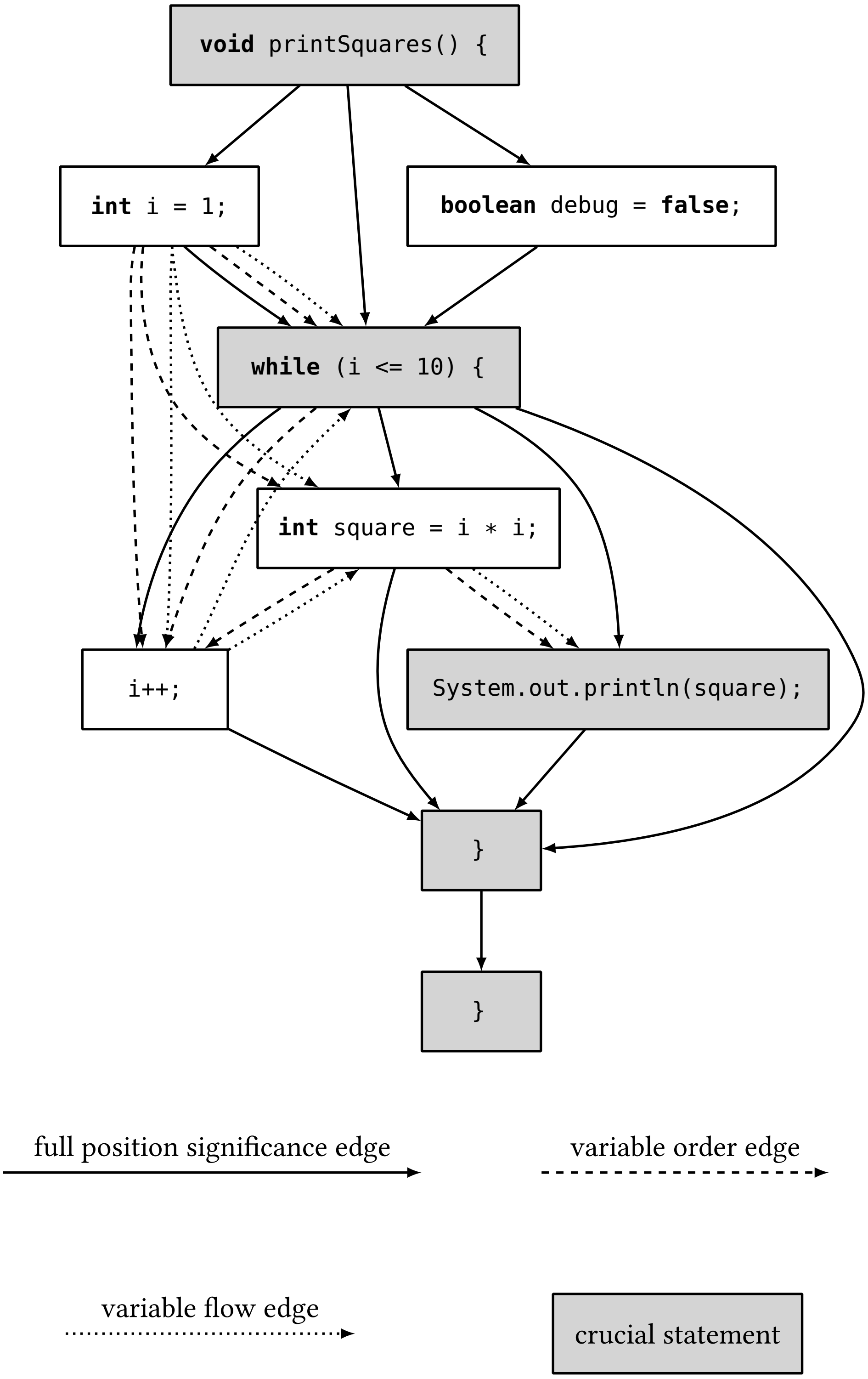

Figure 5.1.: The Normalization Graph created from the token string in Table 5.1 after construction. Note that the graph does not directly contain program statements; we only show them for a better understanding.

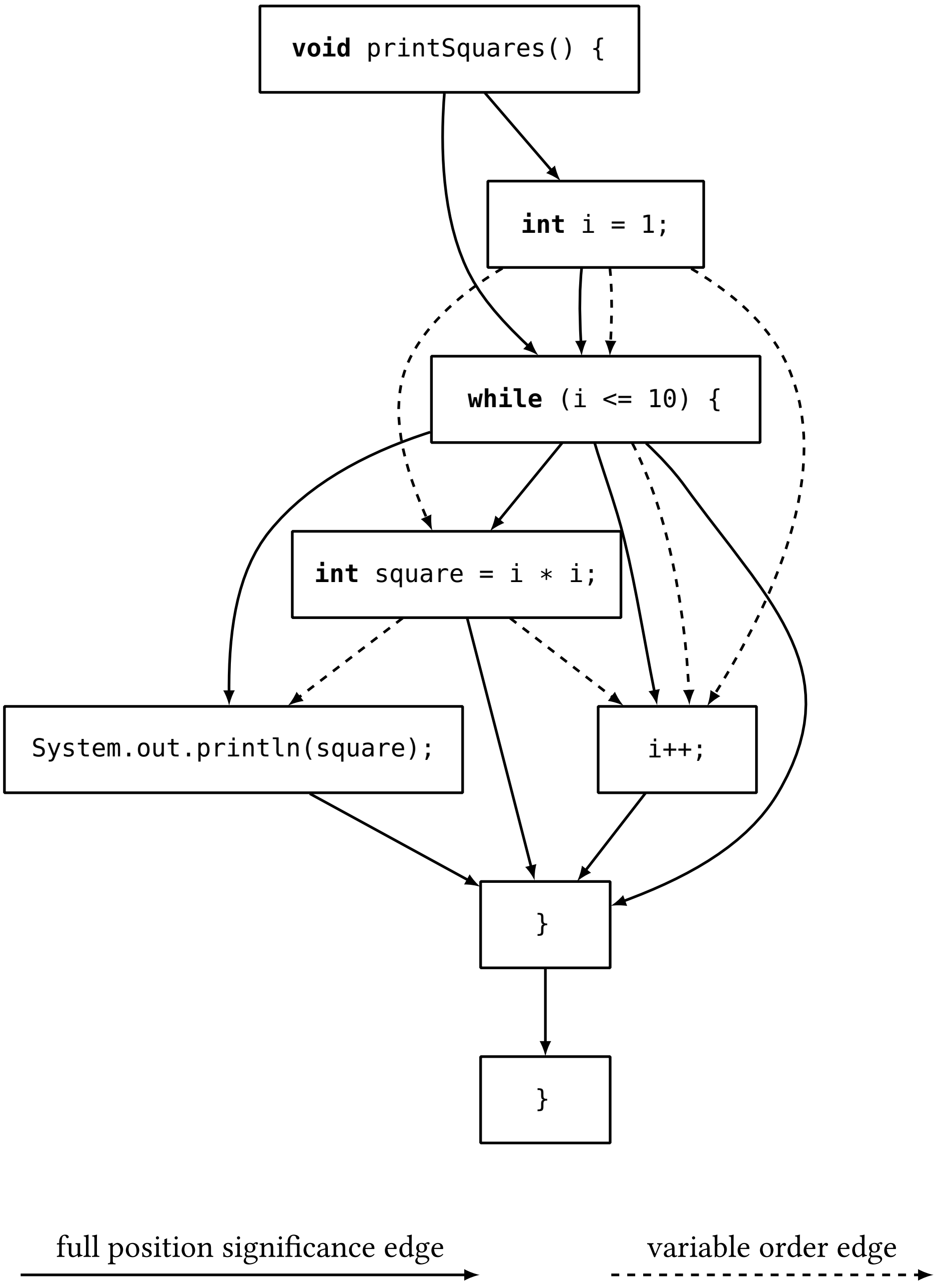

Figure 5.2.: The Normalization Graph created from the token string in Table 5.1 after dead code removal. Note that the graph does not directly contain program statements; we only show them for a better understanding.

6. Implementation

We will implement the token string normalization into JPlag for the Java programming language. JPlag represents the current state-of-the-art in code plagiarism detection [33]. Java is the language JPlag was mainly developed for (as evidenced by its name). Java is also one of the most common languages for introductory programming courses [23, 10]. The author chose it for these reasons. Due to the language-independent nature of the normalization, supporting additional languages would require relatively little additional effort. Doing so would be outside the scope of the thesis, however. The implementation is split into three components. The first component is a generic interface to add semantic information to the tokens during tokenization. The second component uses this interface to add semantic information to Java tokens. The third and final component uses this information to perform token string normalization. Note that only the second component is language-dependent.

6.1. Generic Interface for Semantic Information

We add the generic interface to the `language-api` package as a new subpackage called `semantics`.

6.1.1. Code Semantics

We model the semantic information as a class called `CodeSemantics`. It contains information about the semantics of a code fragment. Specifically, the class contains a code fragment's cruciality and position significance as an enum. It also contains information about bidirectional block depth. More precisely, it contains how much the code fragment causes the depth of bidirectional blocks to change as an integer. We can use this information to calculate which code snippets belong to the same bidirectional block. The variables the code fragment reads and writes are each contained as sets. We add a `CodeSemantics` instance to each token as part of token initialization. We later supplement the variable accesses.

In JPlag, each token already contains a line number. We use it to discern the statement sequence and to group the tokens into statements. For this to work, each line must consist of exactly one statement. We achieve this by formatting the code beforehand. First, we remove comments using the JavaParser library, as a single statement can be broken up into multiple lines through comments. We then format the code using the Eclipse CLI with a particular configuration file. Most importantly, the maximum line width is set to 9999 so that statements are not broken up because of it. We generate a `CodeSemantics` instance for each statement by

Category	**Cruciality**	**Position Significance**	**Bidirectional Block Depth Change**
Default	Not Crucial	None	0
Keep	Crucial	None	0
Critical	Crucial	Partial	0
Control	Crucial	Full	0
Loop Begin	Crucial	Full	1
Loop End	Crucial	Full	-1

Table 6.1.: The six categories most code fragments fit into. See Table 6.2 for examples.

Category	**Example Token**	**Example Statement**
Default	VAR_DEF (in method)	`int localCount;`
Keep	VAR_DEF (in class)	`static int classCount;`
Critical	APPLY (no exception)	`System.out.println("hi");`
Control	RETURN	`return false;`
Loop Begin	FOR_BEGIN	`for(;;) {`
Loop End	WHILE_BEGIN	`} do while (true);`

Table 6.2.: Examples for each of the six code fragment categories defined in Table 6.1.

joining its tokens' instances. The joining of multiple initial `CodeSemantics` instances into a single joined instance is defined as follows:

- The cruciality of the joined instance is the disjunction of each initial instance's cruciality. Simply put, the joined instance is crucial iff any of the initial instances are crucial.
- The position significance of the joined instance is the most significant of each initial instance's position significance.
- The bidirectional block depth change of the joined instance is the sum of each initial instance's bidirectional block depth change.
- The variable reads and writes of the joined instances are the union of each initial instance's variable reads and writes.

Most code fragments fit into one six categories regarding their code semantics (ignoring variable accesses); see Table 6.1 for the categories and Table 6.2 for examples. We implement each category as a constructor of the `CodeSemantics` class.

6.1.2. Variable Registry

The interface provides a `VariableRegistry` class to help track variables during tokenization. We use this class to add variable reads and writes to token semantics. Variables are unique instances of the class `Variable`, which contains the variable's name, its scope, and whether its type is mutable. When a variable is declared, we register it with this information. A variable's scope may be file, class, or method. How we save variables in the registry depends on their scope:

- We save variables with a file scope in a simple hash map, which maps each variable's name to the corresponding `Variable` instance. These variables are visible within the entire file after their declaration. To register such a variable, we create a new `Variable` instance and add it to the map as a value with the variable's name as the key. From then on, we can recover this variable from the map using its name. We must not track file scopes, as `VariableRegistry` instances are unique to files.

- We save variables with a class scope in a stack of hash maps. Again, the variable names are the keys, while the values are the corresponding `Variable` instances. These maps are in a stack since classes can contain other classes, but variables from outer classes cannot be accessed from inner classes. The hash map containing the variables of the current class is on top of the stack. We use this map like the one for file variables to register and recover variables. We add a new hash map to the stack when a class is entered. When a class is exited, we remove the topmost map. For this behavior to work, we must call corresponding methods to inform the variable registry whenever a class is entered or exited.

- Variables with a local scope are saved in a hash map which maps each variable's name to a stack of corresponding `Variable` instances. The values are stacks because local scopes can contain other local scopes, and variables from outer local scopes can be accessed from inner local scopes. The only condition is that no variable with the same name was declared after. To register a variable, we create a new `Variable` instance and put it on top of the stack associated with the variable's name. If there is no associated stack, we associate a new empty one beforehand. To recover a variable, we get the top entry from the stack associated with its name.

  There is another data structure for local variables, which is used to track their visibility. This data structure is a stack of sets. Each set contains the names of variables initialized in a specific local scope. When we register a variable, we add its name to the topmost set in the stack. When a new local scope is entered, we add a new empty set on top of the stack. When a local scope is exited, we remove the topmost set in the stack. We go through the variable names in it, and for every name, we remove the topmost variable in the stack associated with that name. These variables are no longer visible from this point on. If a stack is empty after we have removed its topmost variable, we remove it from the map entirely, along with the associated variable name. Due to this, the hash map has the convenient property that the keys are exactly the names of visible local variables. As with class scopes, we must call corresponding methods whenever a local scope is entered or exited.

We can pass a code semantics instance along with a registered variable's name to add an access to that variable to the instance. A variable's scope can be explicitly set to class on access in many languages. In Java, for example, this is done with the **`this`** keyword (**`this`**`.count`, for example). For this reason, we additionally pass a boolean denoting whether the variable's scope is class. If it is set to true, we recover the variable from the data structure for class variables. Otherwise, we try to recover the variable first from the data structure for local variables, then from the data structure for class variables, and finally from the data structure for file variables. If we did not recover a variable, we encountered an access to an untracked variable. Such a variable has not been declared in the file, so it must be global. If the access to the variable is a write, we cannot gauge the effects. For this reason, we mark the passed code semantics as crucial and having full position significance. A code semantics instance can also be passed to add reads to all non-local variables to it. We can set the type of the next variable access. By default, we assume accesses to be reads. The next variable access can be ignored through another method. The variable registry tracks where accesses to mutable variables are (potentially) writes. One such location is the parameter list of a method invocation, as the method may write to mutable variables. Similar to how we track class and local scopes, we must call corresponding methods whenever such a location was entered or exited.

6.2. Adding Semantic Information to Java Tokens

In JPlag, the Java tokens are generated in a class called `TokenGeneratingTreeScanner`, which is contained in the Java language module. This class extends the `TreeScanner` abstract class. `TreeScanner` contains various `visit` methods for the node types of an abstract syntax tree generated by parsing Java source code. The `visit` methods receive the visited node as input. When a node is visited, the nodes under it are also visited. For this reason, we can visit every node by visiting the root. To generate the tokens, `TokenGeneratingTreeScanner` overrides the `TreeScanner` `visit` methods. For example, it overrides `visitMethodInvocation` to generate an Apply token. We expand the `visit` methods to generate code semantics instances as well. We add these instances to the tokens upon token creation. In Java, every token fits into one of five categories listed in subsection 6.1.1: Default, Keep, Control, Loop Begin, and Loop End. We use the category constructors to initialize the code semantics instances. For example, when we generate an Apply token, we initialize a new code semantics instance with the Control category constructor and add it to that token. We keep a reference to the semantics instance to add variable accesses later. We pass this reference to `visit` methods called within the current `visit` method.

We track and add the variable accesses using a variable registry instance. We register attributes as class variables when a class is visited. When a variable declaration is visited, we register the declared variable as a local variable if we are in a local scope. If we are not, this is an attribute declaration we have already registered, so we do nothing. Upon class visit, we register the class as a file variable since it can (mostly) be accessed from anywhere within the file. Since no variables can be explicitly declared outside of classes, Java has no other file variables. Of course, we need to track class and local scopes to use the variable registry. Class scopes are entered and exited when a class is visited. Local scopes are entered and exited in many `visit` methods, for example, when a block, a method, or a for loop is visited. We add variable accesses to semantics instances in two `visit` methods that that were not overridden previously. In one method, (unqualified) identifiers are visited. Here, we pass the code semantics instance, the variable name, and the information that the variable's scope is not explicitly class to the variable registry. In the other method, member selects are visited. Here, we first check whether the qualifier is the **`this`** keyword. If it is, a class variable is accessed. We pass this information to the variable registry, along with (again) the semantics instance and the variable name.

We set the variable access type when variable writes are visited. In Java, these are assignments. Standard assignments only write the variable. Compound (+=) and unary (++) assignments write and read the variable. We set the next variable access type accordingly through the variable registry. Recall that by default, we assume variable accesses are reads. Therefore, only write and read/write access types must be set explicitly. We must also tell the variable registry when accesses to mutable variables may be writes. In Java, this is the case when a method of a mutable variable is called (`mutable.foo()`) or a mutable variable is a parameter in a method call (`foo(mutable)`). We tell the variable registry when this starts and stops being the case when method calls are visited. We pass whether a variable's type is mutable to the variable registry upon registering the variable. We gain this information by checking the variable's type against a static list of non-mutable types (**`int`**, `Boolean`, `String`, ...).

Finally, we need to deal with a quirk of the identifier and member select visits: They visit both variable and method references, despite variables and methods occupying separate namespaces. Since we only track variables, we need some way to ignore method references. To give some examples, we want to track the `foo` in `foo++`, **`this`**`.foo++`, and `foo.bar()`, but not in `foo()` and `bar.foo()`. We accomplish this by using the variable registry's functionality to ignore the next variable access. We ignore the next variable access when a method call is visited (in order to ignore `foo()` and similar cases), and then un-ignore it when a member select is visited (in order not to ignore `foo.bar()` and similar cases).

6.3. Token String Normalization

We add the token string normalization to the `core` package as a new subpackage called `normalization`. We perform the normalization by first constructing an instance of the class `NormalizationGraph` and then turning it back into a token string.

6.3.1. Normalization Graph Construction

Internally, we use the graph library JGraphT. The nodes representing the program statements are instances of the class `Statement`. This class contains the tokens in the statement as a list, the line number of the statement, and the statement's semantics (as a `CodeSemantics` instance). There is a class `Edge` for the edges as well. To construct the graph, we iterate through the tokens in the token string we received as input. Each time the line number changes, we create a new `Statement` instance from all the tokens with the preceding line number. We gain the statement semantics instance by joining the semantics instances in the tokens as described in section 6.1. We then add the statement to the graph as a node along with its ingoing edges. Each edge type described in subsection 5.2.1 is also found in the implementation. We do not fully abide by the definitions, however. Specifically, variable edges work slightly differently. For one, there is only a variable order edge between two statements if there is no variable flow edge between them. Contrast this to the definition of the graph, where every variable flow edge between two statements implies a variable order edge between them. This redundancy allows for an elegant definition but is unnecessary in practice. Secondly, variable flow edges may no longer go from a statement later in the sequence to an earlier statement. A new type of edge, the reverse variable flow edge, replaces variable flow edges with this property. These edges run in the opposite direction as the edges they replace. As a result, all edges run from earlier to later statements. Due to this construction, the graph is acyclical throughout. For this reason, we must no longer remove variable flow edges after dead code removal. This is another efficiency improvement.

When adding statement S to the graph, we process each piece of semantic information individually. We do this as follows.

- First, we process bidirectional block depth change. We keep count of the current bidirectional block depth. We also keep a set of statements in the current bidirectional block. When we add S to the graph, we add its bidirectional block depth change to the count. If the current bidirectional block depth is positive after, we add S to the set of statements in the current bidirectional block. Otherwise, we clear the set.

- After, we process position significance. If S has partial position significance, we add a partial position significance edge from the last statement with partial position significance to S. Adding full position significance edges is slightly more complex. To do so, we keep a set of all the statements since the last statement with full position significance (including it). If S has full position significance, we add full position significance edges from all the statements in the set to S. If it does not, we add a full position significance edge from the last statement with full position significance to S.

- Finally, we process variable accesses. We track variable reads and writes from processed statements each in a hash map. The keys in these maps are variables. The values are collections containing the statements that write and read the variables, respectively. For every variable S reads, we use the variable writes map to get all the previous statements that wrote this variable. We then add a variable flow edge from every such statement to S. Similarly, we iterate through the variables S writes. For every such variable, we again use the variable writes map to get all the previous statements that wrote the variable. We then add a variable order edge from every such statement to S. We also use the variable reads map to get all the previous statements that read the variable. We add an edge from every such statement to S as well. The edge's type depends on whether the statement the edge comes from is in the same bidirectional block as S. We can check this with the set of statements in the current bidirectional block. If the statement is in the same bidirectional block, the edge is a reverse variable flow edge. If it is not, the edge is a variable order edge.

6.3.2. Normalization Graph Usage

After constructing the graph, we remove statements that are dead code. We do this slightly differently as described in subsection 5.2.2. Instead of removing the nodes representing dead code from the graph, we do not include them in the resulting token string. This is more efficient and has the same effect. To determine which statements are dead code, we start a breadth-first search from all crucial statements. We search backward along variable flow edges and forward along reverse variable flow edges. We mark all statements reached this way as not being dead code. Correspondingly, we regard all statements not reached this way as dead code.

Finally, we turn the graph back into a token string using topological sorting as described in subsection 5.2.2. We use a priority queue of statements for this purpose. The priority queue requires statements to have an order. It then returns and removes the statement with the lowest order contained in it when called. We implement the `Statement::compareTo` method to impose an order on the statements. This order must only stem from the statements' tokens. We choose the following order for two random statements S and T: S comes before T if it has more tokens. If S and T have equally many tokens, we compare the token types alphabetically to determine the order. If all the token types are equal as well, S and T are equal in order. Initially, we put all the roots, meaning all the statements without ingoing edges, into a priority queue. We call on the priority queue until it is empty to process all the contained statements in order. To process a statement S, we first check whether we previously marked it as not being dead code. We add its tokens to the normalized token string if we did. Then, we go through the successors of S. For each successor T, we remove all the edges from S to T. If this causes T to have no more ingoing edges, we add T to a new priority queue. After processing all the statements in the current priority queue, we check if there are any statements in the new priority queue. If there are, we process them in order, repeating the process. If there are not, we have processed all nodes in the graph. The result is the normalized token string.

7. Evaluation

Throughout the evaluation, we will compare our version of JPlag using token string normalization with the original version. We will refer to the former as *Normalization-JPlag* and the latter as *Original-JPlag*.

7.1. Methodology

7.1.1. Goals, Questions & Metrics

We deem automatic plagiarism generation using statement insertion and reordering a significant threat to academic integrity. For this reason, our primary goal is to make JPlag resistant to such attacks through token string normalization. We consider three variations of automatic plagiarism generation: One using insertion, one using reordering, and one using a combination of both. *Similarity Score to the Original (SSO)* is the central metric here: Plagiarized submissions should have a high SSO, as this means that JPlag detected them successfully. Our secondary goal is to retain JPlag's existing capabilities; they should not be diminished by adding token string normalization. Two aspects are of particular concern: false positives and runtime. We compare the performance of Normalization-JPlag with that of Original-JPlag in those areas. We measure false positives with the similarity score of non-plagiarized submission pairs, which should generally be low.

Primary Goal Make JPlag resistant to automatic plagiarism generation using insertion and reordering.

> **Question** How well does automatic plagiarism generation using insertion and reordering work with Normalization-JPlag?
>
> > **Metric** Normalization-JPlag's SSO of automatically plagiarized submissions.

Secondary Goal Retain JPlag's existing capabilities.

> **First Question** Does normalization make false positives more likely?
>
> > **Metric** Similarity score of non-plagiarized submission pairs of Normalization-JPlag compared with Original-JPlag.
>
> **Second Question** How high are the runtime costs of normalization?
>
> > **Metric** Runtime of Normalization-JPlag compared with Original-JPlag.

7.1.2. Dataset

Our dataset is PROGpedia [25], a collection of submissions from introductory programming courses. Sixteen tasks and student solutions to them are included. Each task may include multiple submissions per course participant. The students could program in one of four languages: C, C++, Java, or Python. The submissions are split into four categories: Correct, Incorrect, Runtime Error, and Compile Error. Submissions in the latter category are not included in the dataset. The dataset was created for research into guiding students toward the correct solution. For this purpose, a code property graph of each submission is included. The code property graph is a generalization of the program dependence graph. We will not use the included graphs despite this relation to token string normalization. The author chose this dataset since it consists of the kind of submissions JPlag is mainly used for.

We pre-process the dataset to prepare it for the rest of the evaluation. First, we merge the submissions from the different categories for each task, as we will not use this information. Then, we remove all the submissions that use a language other than Java. After, we remove all but the latest submission from each student per task. The rationale for this step is that multiple submissions from the same student for the same task are bound to be highly similar. Therefore, we may consider pairs of such submissions to be plagiarized. However, we want each pair of submissions in the prepared dataset to be non-plagiarized. This is because we are only interested in automatically plagiarized submissions. Thus, no manually plagiarized submissions may be left in the dataset. To remove all of them, we must also check for plagiarized pairs of submissions from different students for the same task. We do so manually, with assistance from JPlag, concluding the pre-processing.

The author chose two tasks for the evaluation, partially based on how easy it seemed to eliminate all plagiarized pairs. Other factors considered were the number of submissions and their average line length. The first task is task 56, which is about minimum spanning trees. After removing all plagiarized submissions, there are 28 submissions left, with a mean line count of 85 and a median of 77. The second task is task 19, which is about graph clustering. We remove all plagiarized submissions here as well. Afterward, there are 27 submissions left, with a mean line count of 131 and a median of 106. Since both tasks yield similar results, we will only look at those for task 56 in the evaluation. The results for task 19 can be found in the appendix.

7.2. JPlag-Gen

The author is aware of only one currently existing automatic plagiarism generation tool: Mossad [11], which we already outlined in chapter 4. Mossad works with C/C++ and exclusively uses insertion. We implemented the token string normalization only for Java, however. In addition, we want to consider reordering. For these two reasons, we develop our own tool for automatic plagiarism generation. It is called *JPlag-Gen*. It can produce plagiarized submissions using insertion and reordering. It can also combine these modes by doing the following: First, it produces a plagiarized submission using reordering. It then produces another plagiarized submission from the result using insertion. The rationale for this order is that reordering is the weaker attack, as already noted in section 4.2.

We will not consider how suspicious a plagiarized submission may be to a human in the evaluation. For this reason, JPlag-Gen only looks to attack token-based plagiarism detectors as potently as possible. Specifically, when generating a plagiarized submission, JPlag-Gen tries to lower JPlag's SSO as much as possible. It does this by breaking up all code segments it can. Recall that breaking up code segments is the relevant mechanism by which SSO is lowered. Once all code segments are broken up, we can safely terminate, knowing that any decrease in similarity afterward is superficial. This termination criterium does not depend on a plagiarism detector's score or an arbitrary timeout period, as MOSSAD's does. As a result, JPlag-Gen works entirely independently of plagiarism detectors. For this reason, we can generate a single submission and test multiple plagiarism generators against it. This greatly simplifies comparing plagiarism detectors, as we do not need to re-generate a plagiarized submission for every plagiarism detector we want to test. Such a process would be both cumbersome and introduce noise, as automatic plagiarism generation is usually random. Much like token string normalization, JPlag-Gen relies on statements and lines being equivalent, so we format all the submissions we give it as input beforehand.

7.2.1. Using Insertion

To produce plagiarized submissions using insertion with JPlag-Gen, we use the same basic mechanism as MOSSAD: We insert a random statement in a random position and then use a compiler to check whether this has changed the submission's functionality. More precisely, we compile the submission with optimizations and check if the resulting bytecode has changed. If it has, we remove the inserted statement again. As in MOSSAD, the random statement is from an entropy file, the current submission, or a previously plagiarized submission. There are two crucial differences, however.

One difference is the compiler. MOSSAD uses GCC. However, this compiler only works for C/C++, so we have to use a different compiler for Java. Javac, the standard Java compiler, does not do any optimizations. Instead, we use the Eclipse Compiler for Java (ECJ), which can be configured to remove unused variables. We first compile the submission to class files using ECJ. Afterward, we use the `javap` command with the `-c` and `-p` flags to extract the bytecode from the class files. ECJ is less powerful than GCC for our purposes, as GCC can also remove types of dead code other than unused variables. However, most of MOSSAD's functionality can still be replicated, as MOSSAD mainly relies on inserting unused variables [19]. Furthermore, some dead code insertions MOSSAD uses, but JPlag-Gen does not, can be attributed to the language used rather than the different compilers. For example, unreachable code is allowed in C/C++, while in Java, it is not. Another example is `using namespace` statements in C++, which can be inserted as dead code but have no equivalent in Java.

The other difference comes from JPlag-Gen's termination criterium. We implement it for insertion using a pool of positions where a statement may be inserted. The positions

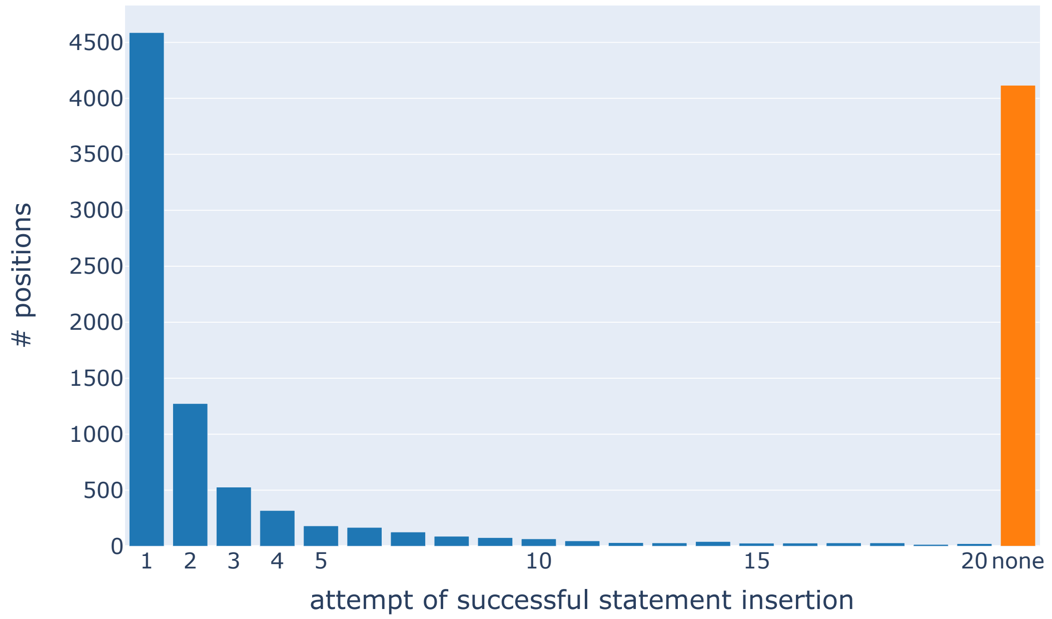

Figure 7.1.: This graph shows for every possible attempt how many positions had a statement successfully inserted into them on that attempt throughout our evaluation. The positions with no statement successfully inserted into them within twenty attempts are shown with the label "none".

where statements are inserted are chosen randomly from this pool. JPlag-Gen terminates when the pool is empty. Initially, this pool consists of every position in the submission. Once a statement has been inserted in a position, we remove it from the pool, as inserting another statement in this position would not break up any more code segments. There may be positions where no statement can be inserted without changing the submission's functionality. We determine such positions by counting for every position how many times we have tried to insert a random statement there. After 20 unsuccessful attempts, we assume no statement can be inserted in the position and remove it from the pool. This assumption is backed up empirically; see Figure 7.1. This figure visualizes the attempts. It shows that a statement is successfully inserted in most positions within the first few attempts. In about 35% of positions, no statement is successfully inserted within twenty attempts. It takes between ten and twenty attempts for only about 3% of positions. From this, we can extrapolate that a successful insertion in a position after more than twenty attempts is improbable. It is much more likely that no statement can be inserted there at all.

7.2.2. Using Reordering

JPlag can also produce plagiarized submissions using reordering. Unfortunately, ECJ is of no use to us here, as it does not change the order of statements during compilation. Instead of relying on a compiler, we use an entirely different approach for reordering. Rather than making a modification and undoing it if it has changed the submission's functionality, as we did for insertion, we only make modifications we know beforehand do not change the submission's functionality. In the context of reordering, this means we determine whether two statements are independent, meaning they do not affect each other. If two subsequent statements are independent, we can swap them without changing program functionality. As the author found no existing tools with this capability, we test whether two subsequent statements are independent ourselves. Statements must fulfill three criteria to pass this test: First, both statements must not begin or end a code block. We check this using statements' indentation. For one, the two statements must have the same indentation. Additionally, the first statement must not have a smaller indent than the statement preceding it; otherwise, the first statement may end a code block. Similarly, the second statement must not have a smaller indent than the statement succeeding it. Secondly, neither statement may affect control flow. To this end, we check whether the statements contain a method call or any of the keywords **`return`**, **`yield`**, **`break`**, **`continue`**, **`throw`**, **`assert`**. Third, if both statements access the same variable, neither may write to it. In order to determine method calls and variable accesses in statements, we re-use the generation of semantic information from token string normalization.

We again implement the termination criterium using a pool of positions. We choose the positions randomly from the pool. After choosing a position S, we check the criteria for the independence of S and the subsequent statement T. If they are met, we swap the two. Additionally, S must have come before T in the original submission. This condition prevents infinite loops where two statements are swapped back and forth forever. Regardless of whether a swap has occurred, we remove the position from the pool, as picking it again would not result in a swap either way. If a swap has occurred, we add the positions before and after to the pool, as a swap can now occur there even if it could not before. Note that the pool is a set, so every position is contained at most once. Again, JPlag-Gen terminates when the pool is empty.

7.3. Effect on Automatic Plagiarism Generation

We check what effect Normalization-JPlag has on automatic plagiarism generation to evaluate our primary goal. We measure this with Normalization-JPlag's SSO of automatically plagiarized submissions; it should be as high as possible. We create such submissions with JPlag-Gen's three modes (insertion, reordering, both). Rather than just measuring once at the end, we measure after every modification JPlag-Gen makes during generation. As the effect of a single modification depends on the submission's size, we track the number of modifications on a submission proportionally to its statement count. To put Normalization-JPlag's SSO into context, we also measure Original-JPlag's SSO.

7.3.1. Using Insertion

First, we check how well Normalization-JPlag fares against JPlag-Gen's insertion mode. We start by looking at ten random individual plagiarized submissions; see Figure 7.2. Insertion effectively reduces Original-JPlag's SSO. Normalization-JPlag's SSO, in contrast, is affected only rarely; most plagiarized submissions still have the maximum SSO of 100% at the end of the process. As we can see by the relatively tight grouping of the individual submissions, the effectiveness of insertion against Original-JPlag is fairly independent of the submission. Now, we look at all plagiarized submissions created using insertion by taking the average SSO; see Figure 7.3. In the end, Original-JPlag's average SSO is 19.4%, while Normalization-JPlag's is 99.5%. From this, we can conclude that Normalization-JPlag is highly resistant to automatic plagiarism generation using insertion.

7.3.2. Using Reordering

Next, we check how well Normalization-JPlag defends against JPlag-Gen's reordering mode. We again first look at ten random individual plagiarized submissions; see Figure 7.4. Reordering is only somewhat effective in reducing Original-JPlag's SSO. In contrast to insertion, the effectiveness of reordering against Original-JPlag is highly dependent on the submission, as we can see by the wide range of outcomes for individual submissions. For some submissions, reordering does not affect Original-JPlag's SSO, meaning it remains at 100%. In these cases, JPlag-Gen swapped only statements corresponding to the same tokens, attribute declarations being a typical example. For other submissions, reordering accomplishes more, bringing Original-JPlag's SSO below 90% or even below 80%. Still, this data confirms the notion that reordering is overall less effective than reordering. Again, Normalization-JPlag's SSO remains unchanged for the most part. Once more, we transition to looking at all plagiarized submissions created using reordering by taking the average SSO; see Figure 7.5. In the end, Original-JPlag's average SSO is 92.2%, while Normalization-JPlag's average SSO is 99.6%. Compared to Original-JPlag, this is a relatively weaker result for Normalization-JPlag than when insertion was used, though its average SSO is still well above the 99% mark. We can conclude that Normalization-JPlag also has a high resistance to automatic plagiarism generation using reordering.

7.3.3. Using a Combination

Finally, we check how well Normalization-JPlag fares against JPlag-Gen's mode combining reordering and insertion. Recall that this mode works by first creating a plagiarized submission using reordering and then creating the final plagiarized submission from it using insertion. As we already looked at submissions automatically plagiarized using reordering, we will

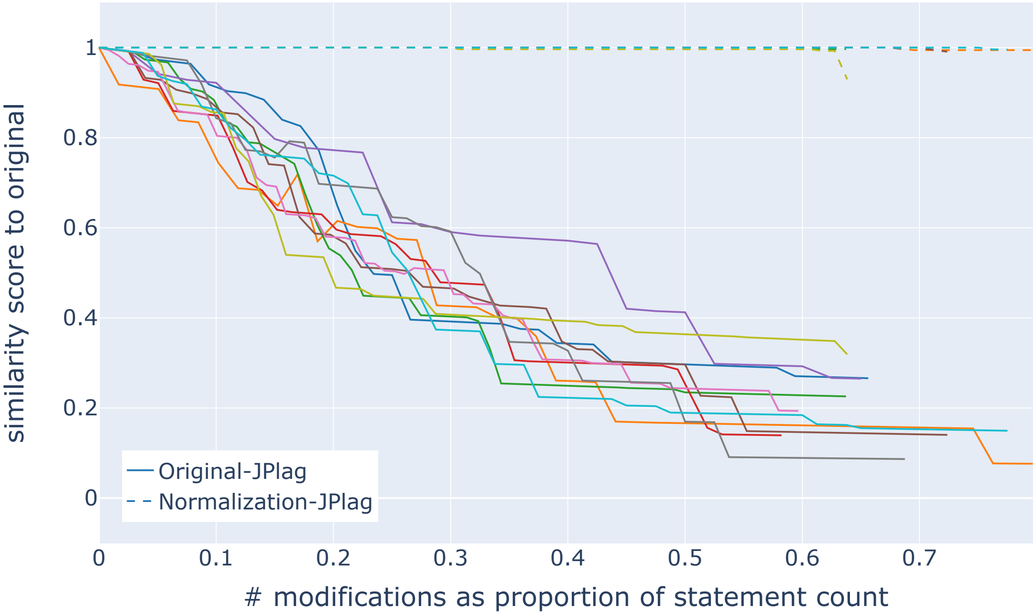

Figure 7.2.: The SSO during the process of creating ten random plagiarized submissions from task 56 using insertion.

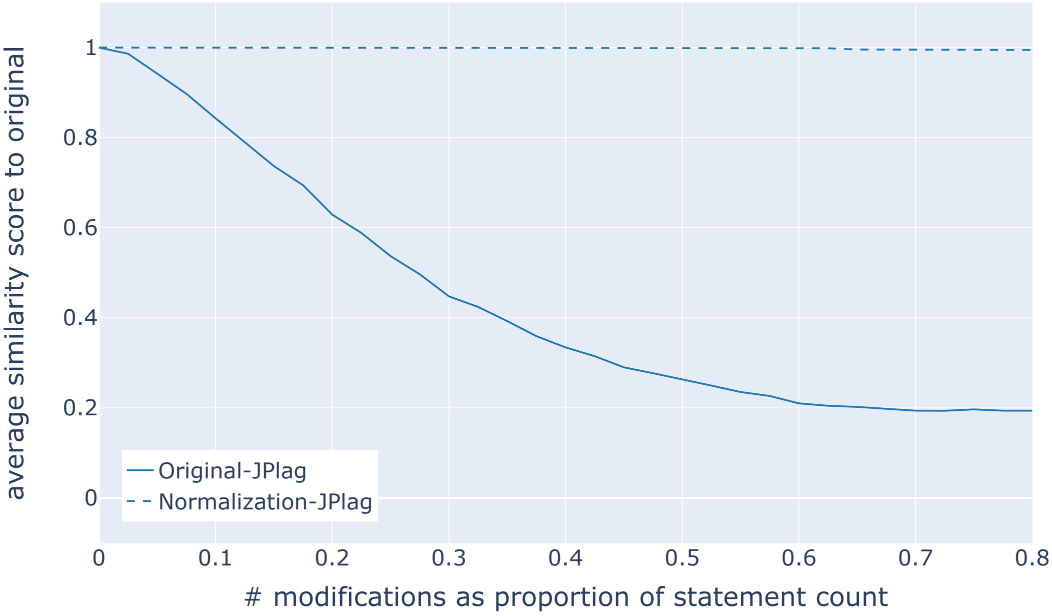

Figure 7.3.: The average SSO during the process of creating all plagiarized submissions from task 56 using insertion.

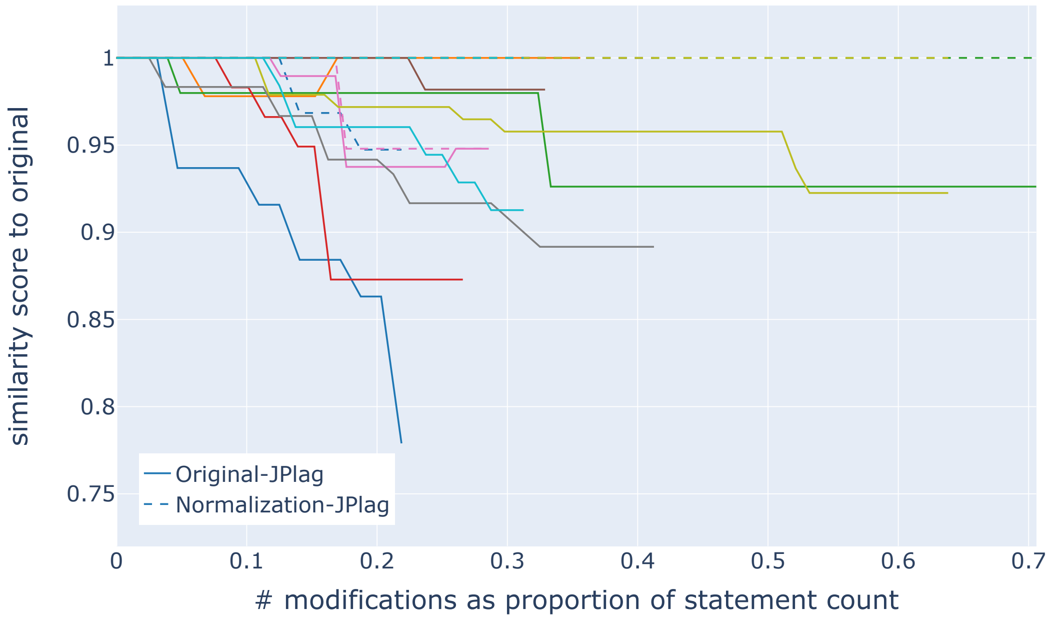

Figure 7.4.: The SSO during the process of creating ten random plagiarized submissions from task 56 using reordering.

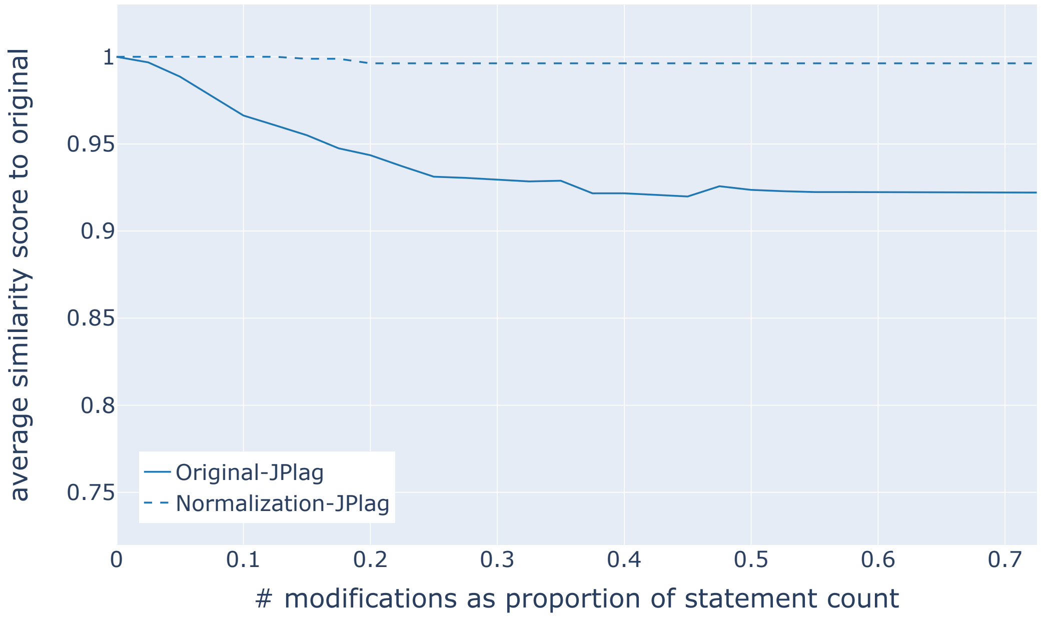

Figure 7.5.: The average SSO during the process of creating all plagiarized submissions from task 56 using reordering.

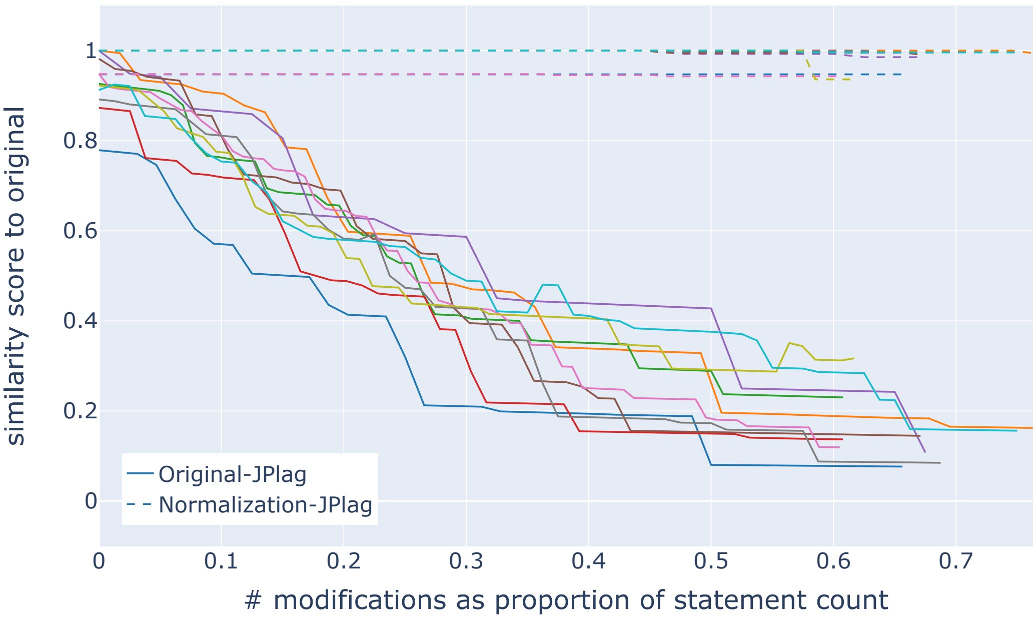

Figure 7.6.: The SSO during the process of creating ten random plagiarized submissions from task 56 using reordering followed by insertion. Only the insertion stage is shown, for the reordering stage, see Figure 7.4.

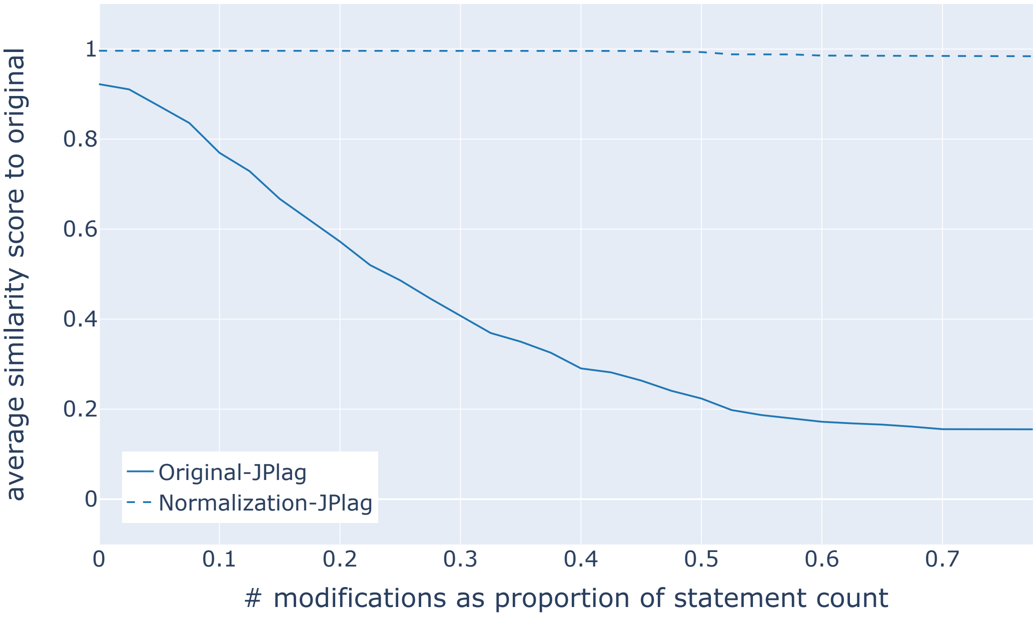

Figure 7.7.: The average SSO during the process of creating all plagiarized submissions from task 56 using reordering followed by insertion. Only the insertion stage is shown, for the reordering stage, see Figure 7.5.

now only look at the insertion stage that has them as input. For ten individual plagiarized submissions, see Figure 7.6; for the average of all plagiarized submissions, see Figure 7.7. Overall, the results are similar to when only insertion was used, albeit both Original-JPlag's and Normalization-JPlag's scores are slightly lower throughout. In the end, the average SSO is 15.5% and 98.4%, respectively. Still, we can conclude that Normalization-JPlag is also highly resistant to automatic plagiarism generation using reordering followed by insertion.

7.4. Effect on Existing Capabilities

We compare false positives and runtime between Normalization-JPlag and Original-JPlag to evaluate our secondary goal. False positives are a concern because many more submissions result in the same token string in Normalization-JPlag than in Original-JPlag. Normalization-JPlag may produce generally higher similarity scores than Original-JPlag for this reason. In the extreme case, it may even produce a very high similarity score for a plagiarized pair; we would refer to this as a false positive. Even in less extreme cases, higher scores for non-plagiarized pairs would be bad, making distinguishing them from plagiarized pairs harder. Therefore, we will investigate the similarity scores of non-plagiarized pairs.

We will also examine the runtime impact of normalization. This is a concern because graph-based approaches to plagiarism detection are often very slow. Mostly, this is because they attempt to solve the NP-complete problem of graph isomorphism, which Normalization-JPlag does not. Still, we want to empirically confirm that Normalization-JPlag is fast enough to be usable in practice.

7.4.1. False Positives

Due to our processing, all submissions pairs in both tasks are non-plagiarized. There are about 750 of them per task. See Figure 7.8 for their similarity scores. The scores are mostly unchanged between Original-JPlag and Normalization-JPlag, with an upper bound of 8.8% and 8.6%, respectively. The upper whisker has also decreased slightly, from 20.6% to 20.3%. Over 95% of scores increase not at all or by less than 1%. The single largest increase is less than 10%. We can conclude that false positives are not a problem with Normalization-JPlag.

To further illustrate this, we compare the scores of non-plagiarized pairs with those of plagiarized pairs. For plots of their distribution, see Figure 7.9 for insertion, Figure 7.10 for reordering, and Figure 7.11 for both. We can see that normalization causes a miniscule change in the distribution of scores of non-plagiarized pairs. However, there is a massive change in the distribution of scores of plagiarized pairs with all three obfuscation methods. Additionally, we can see that insertion causes a significant overlap between Original-JPlag's scores of plagiarized and non-plagiarized pairs. This overlap disappears with Normalization-JPlag. Instead, there is a gigantic gap, spanning almost the entire range of possible scores.

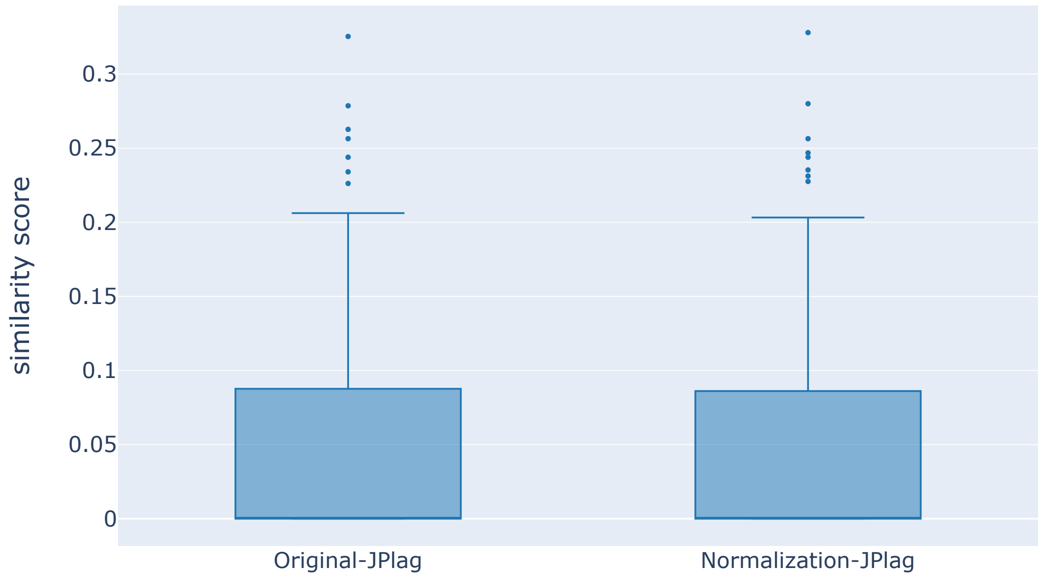

Figure 7.8.: The similarity scores of the submission pairs in task 56, which are all non-plagiarized.

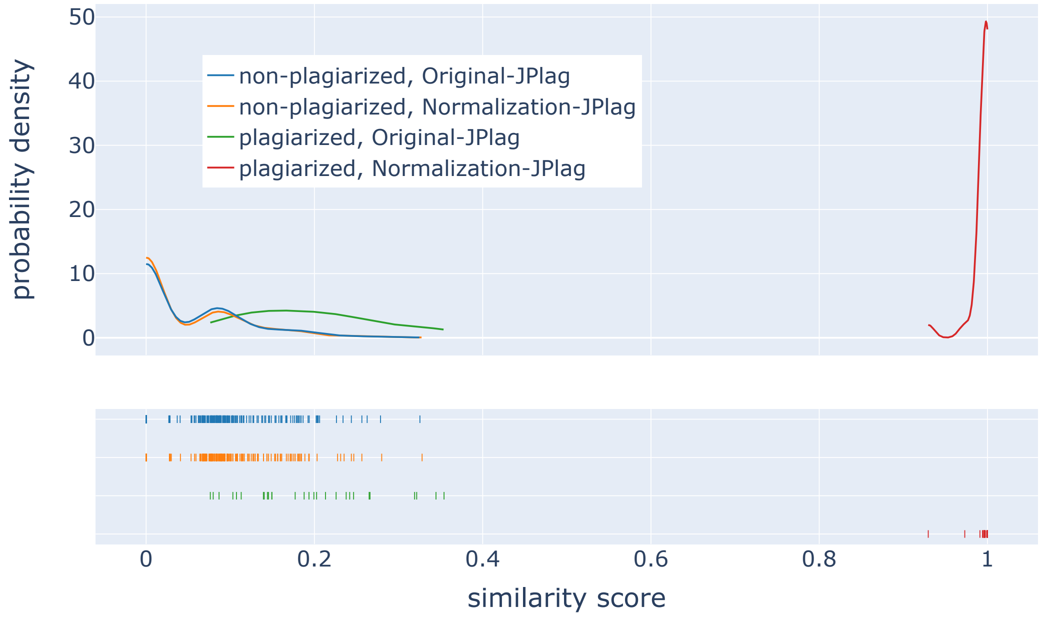

Figure 7.9.: The similarity score distribution of the non-plagiarized and the plagiarized pairs from task 56. The plagiarized pairs each consist of an original and a plagiarized from it using insertion.

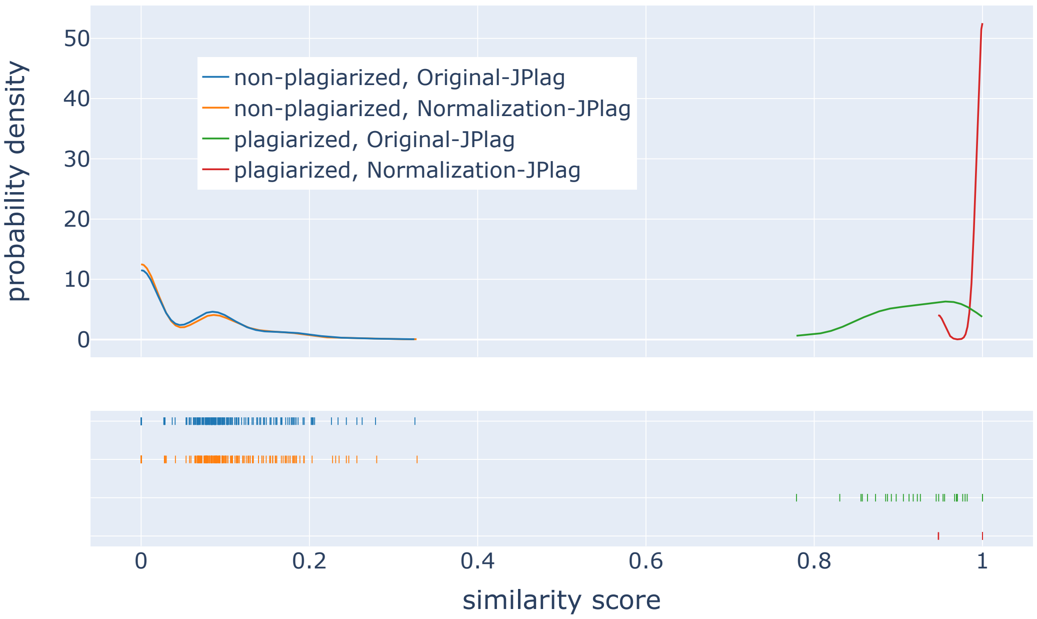

Figure 7.10.: The similarity score distribution of the non-plagiarized and the plagiarized pairs from task 56. The plagiarized pairs each consist of an original and a version plagiarized from it using reordering.

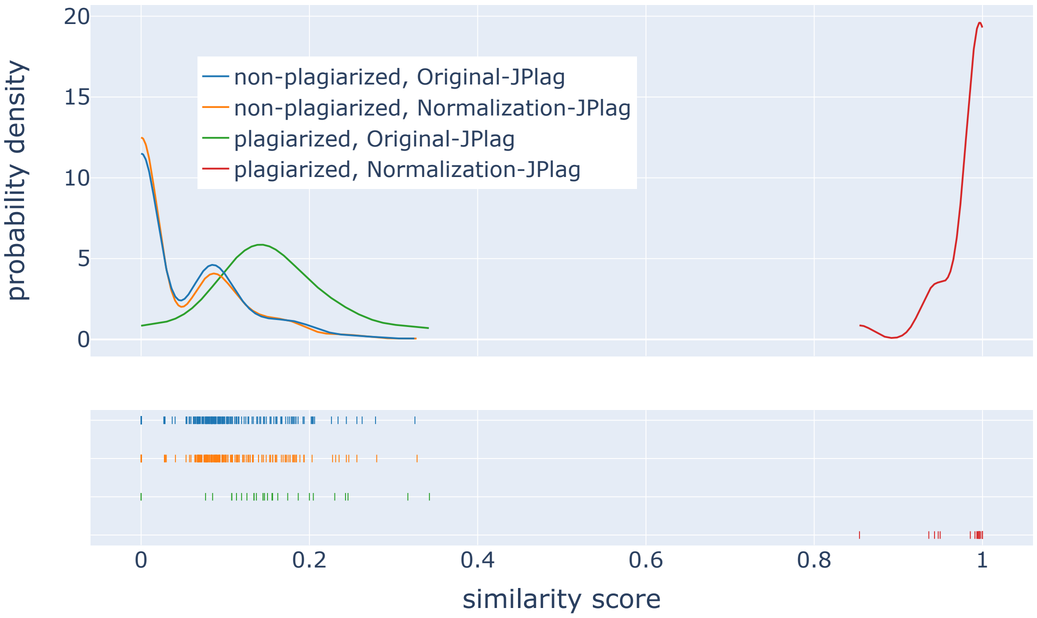

Figure 7.11.: The similarity score distribution of the non-plagiarized and the plagiarized pairs from task 56. The plagiarized pairs each consist of an original submission and one plagiarized from it using reordering followed by insertion.

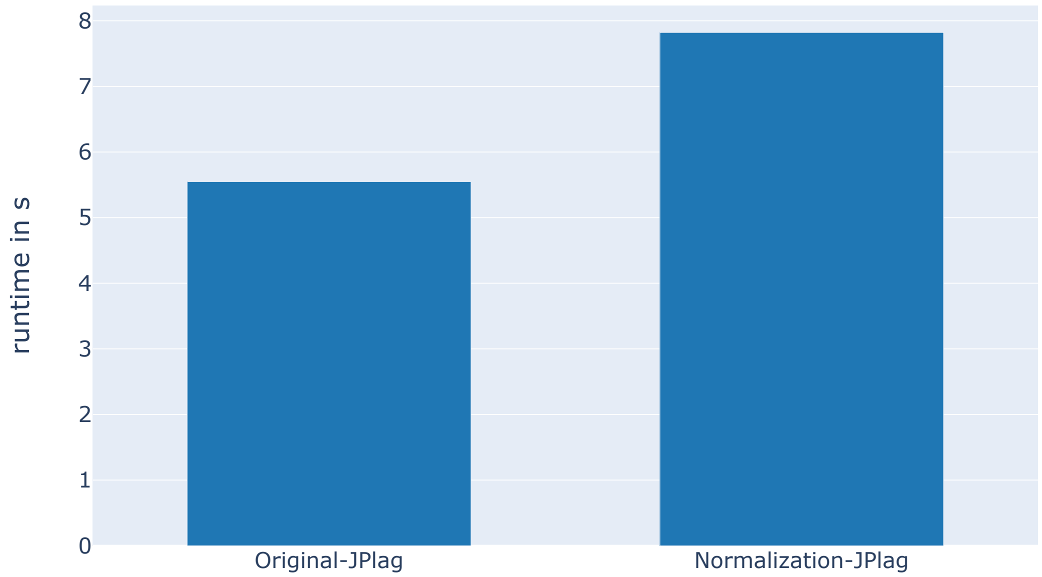

Figure 7.12.: The runtime for an upsized version of task 56 as measured over 100 runs on an Apple M1 Pro with clustering disabled.

7.4.2. Runtime

Our tasks are moderate in size, each containing under 30 submissions, with those submissions containing about 100 statements on average. Runtime only becomes a concern with relatively large inputs. For this reason, we measure the runtime with upsized versions of our tasks. The tasks themselves are upsized by a factor of ten, as every submission in the original task is contained ten times. The submissions are also upsized by a factor of ten, as every source code file in the original submission is contained ten times. In total, the upsized tasks are 100 times bigger than the original tasks, containing hundreds of thousands of statements. See Figure 7.12 for a plot of the runtime measurements. Original-JPlag's runtime is 5.5 seconds, while Normalization-JPlag's runtime is 7.8 seconds; this presents a relative increase of about 40%. Nevertheless, Normalization-JPlag is very fast in absolute terms. Moreover, the runtime of token string normalization scales linearly with both submission count and submission size, unlike the rest of JPlag, which has worse scaling. For this reason, the difference in runtime between Original-JPlag and Normalization-JPlag is bound to shrink as the input gets larger, making the measured increase in runtime even less of a concern. We can conclude that runtime is not an issue with Normalization-JPlag.

7.5. Discussion

In section 7.3, we observed that most insertion and reordering modifications do not affect Normalization-JPlag's SSO. There are some notable exceptions, however. We will look at them here. Most of the code examples are authentic in that JPlag-Gen generated them during the evaluation.

One example leverages an existing weakness of JPlag. If all branches of an **if**/**else** result in a **return**, we can remove the last **else** without affecting the submission's functionality. We illustrate this with the example method below. The original is on the left, and the manually plagiarized version is on the right.

```
public int compareTo(Node n) {
    if (dist < n.dist)
        return -1;
    else if (dist > n.dist)
        return 1;
    else
        return 0;
}
```

```
public int compareTo(Node n) {
    if (dist < n.dist)
        return -1;
    else if (dist > n.dist)
        return 1;
    // else
    return 0;
}
```

These two methods result in different token strings, as the right one is missing an **else**. JPlag-Gen can automatically recreate this using two insertions:

```
public int compareTo(Node n) {
    int index = 0;  // inserted
    if (dist < n.dist)
        return -1;
    else if (dist > n.dist)
        return 1;
    else
        index++;  // inserted
    return 0;
}
```

Both insertions are necessary, as declaring variables in unbraced control blocks is not allowed in Java. Normalization-JPlag removes both inserted statements during normalization. Regardless, the token string is different; `return 0` was inside the **else** block before, but now it is outside.

The next example relies on an empty constructor. JPlag-Gen can reorder it with attribute declarations, changing the token string in the process:

```
class Node {
    Node() {}  // reordered
    public int index;  // reordered
    public Node pai;  // reordered
    ...
}
```

The final two examples are related. In both, JPlag-Gen can insert redundant control statements, again changing the token string:

```
for (i = 0; i < n - 1; i++) {        void printResult() {
    ...                                  ...
    continue;  // inserted               return;  // inserted
}                                    }
```

7.6. Threats to Validity

We only showed that Normalization-JPlag prevents automatic plagiarism generation using our tool, JPlag-Gen. Other, more advanced tools may defeat Normalization-JPlag, possibly even using insertion and reordering. There are two tangible ways in which JPlag-Gen may be considered particularly weak. For one, insertion is somewhat limited, as ECJ, our compiler, only removes unused variables. However, Mossad uses GCC, which features much more powerful optimization, yet Mossad still mainly relies on unused variables [19]. Secondly, reordering uses the same information as token string normalization. As a result, for the most part, JPlag-Gen determining whether it swaps two lines corresponds to Normalization-JPlag determining whether it swaps them back. Therefore, it is unsurprising that Normalization-JPlag defends well against JPlag-Gen's reordering mode. Still, we successfully attacked Original-JPlag in two distinct ways and showed that Normalization-JPlag prevents these attacks.

Despite the language-independent nature of token string normalization, we only evaluated it for Java. It may work worse for other languages, albeit many widely used languages are similar to Java in structure ("C-like").

8. Future Work

There is much room for evaluating token string normalization more thoroughly. Testing it for languages other than Java is an obvious example. Besides only considering Java, we also constrained ourselves to plagiarized submissions created by automatic plagiarism generators. Checking the effect token string normalization has on identifying submissions plagiarized by hand could be an avenue for further research. One approach might involve comparing Normalization-JPlag's score with Original-JPlag's score. If the former is significantly higher, this could indicate that a student tried to hide their plagiarism through insertion or reordering. Unfortunately, there are no standardized datasets for plagiarism detection in the literature. Creating such a dataset would greatly aid future research in the field.

Due to its high-level view of code, token string normalization is vulnerable to some unusual cases of insertion and reordering. Mostly, this comes from how we handle control statements. Unfortunately, we have no conception of the program's control flow. In order to get around this limitation, future work could explore the possibility of using graphs that consider the program's control flow, such as the system dependence graph. A related issue is that we fix control statements in place to retain program functionality. As a result, we detect neither the insertion nor reordering of control blocks. A more advanced version of token string normalization may seek to prevent such attacks. It would need to track blocks generally, similar to how we currently track bidirectional blocks. The Normalization Graph would need to take this information into account.

9. Conclusion

Automatic plagiarism generation using statement insertion and reordering is a significant threat to academic integrity. Unfortunately, state-of-the-art plagiarism detectors using tokenization are vulnerable to such attacks. We sought to improve their resistance through token string normalization, which makes the token string invariant to statement insertion and reordering. By working with the token string, we achieved a high language independence. We used a program graph to perform the normalization. We implemented token string normalization into the state-of-the-art token-based plagiarism detector JPlag. For our evaluation, we created the automatic plagiarism generator JPlag-Gen, which uses statement insertion and reordering. This tool easily defeated the original version of JPlag. In contrast, our version was almost entirely unaffected, showing a high resistance against automatic plagiarism generation using statement insertion and reordering. Furthermore, we sought to confirm that we retained JPlag's existing capabilities. To this end, we compared the runtime and the likelihood of false positives between the original and our version. There was a noticeable increase in runtime; however, our version is still extremely fast in absolute terms. Regarding false positives, we could not discern a difference between the two versions. Overall, we concluded that JPlag's useability remains intact. In summary, our token string normalization approach significantly improves token-based plagiarism detectors' resistance to automatic plagiarism generation using statement insertion and reordering while preserving their useability.

Bibliography

[1] França B. Allyson et al. “Sherlock N-overlap: Invasive Normalization and Overlap Coefficient for the Similarity Analysis Between Source Code”. In: *IEEE Transactions on Computers* 68.5 (2019), pp. 740–751. DOI: `10.1109/TC.2018.2881449`.

[2] Jess Bidgood and Jeremy B. Merrill. “As Computer Coding Classes Swell, So Does Cheating”. In: *New York Times* (May 29, 2017). URL: `https://www.nytimes.com/2017/05/29/us/computer-science-cheating.html` (visited on 04/07/2023).

[3] K.W. Bowyer and L.O. Hall. “Experience using "MOSS" to detect cheating on programming assignments”. In: *FIE’99 Frontiers in Education. 29th Annual Frontiers in Education Conference. Designing the Future of Science and Engineering Education. Conference Proceedings (IEEE Cat. No.99CH37011.* Vol. 3. 1999, 13B3/18–13B3/22 vol.3. DOI: `10.1109/FIE.1999.840376`.

[4] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. “Code Normalization for Self-Mutating Malware”. In: *IEEE Security & Privacy* 5.2 (2007), pp. 46–54. DOI: `10.1109/MSP.2007.31`.

[5] Tracy Camp et al. “Generation CS: The Growth of Computer Science”. In: *ACM Inroads* 8.2 (May 2017), pp. 44–50. ISSN: 2153-2184. DOI: `10.1145/3084362`. URL: `https://doi.org/10.1145/3084362`.

[6] Dong-Kyu Chae et al. “Software Plagiarism Detection: A Graph-Based Approach”. In: *Proceedings of the 22nd ACM International Conference on Information & Knowledge Management.* CIKM ’13. San Francisco, California, USA: Association for Computing Machinery, 2013, pp. 1577–1580. ISBN: 9781450322638. DOI: `10.1145/2505515.2507848`. URL: `https://doi.org/10.1145/2505515.2507848`.

[7] Hayden Cheers. *BPlag.* Oct. 2020. URL: `https://github.com/hjc851/BPlag` (visited on 04/05/2023).

[8] Hayden Cheers, Yuqing Lin, and Shamus P. Smith. “Academic Source Code Plagiarism Detection by Measuring Program Behavioral Similarity”. In: *IEEE Access* 9 (2021), pp. 50391–50412. DOI: `10.1109/ACCESS.2021.3069367`.

[9] Georgina Cosma and Mike Joy. “Towards a Definition of Source-Code Plagiarism”. In: *IEEE Transactions on Education* 51.2 (May 2008), pp. 195–200. ISSN: 1557-9638. DOI: `10.1109/TE.2007.906776`.

[10] Tom Crick. "An Analysis of Introductory Programming Courses at UK Universities". In: *The Art, Science, and Engineering of Programming* 1.2 (2017). Ed. by James H. Davenport. URL: `http://programming-journal.org/2017/1/18/`.

[11] Breanna Devore-McDonald and Emery D. Berger. "Mossad: Defeating Software Plagiarism Detection". In: *Proc. ACM Program. Lang.* 4.OOPSLA (Nov. 2020). DOI: `10.1145/3428206`. URL: `https://doi.org/10.1145/3428206`.

[12] John L. Donaldson, Ann-Marie Lancaster, and Paula H. Sposato. "A Plagiarism Detection System". In: *Proceedings of the Twelfth SIGCSE Technical Symposium on Computer Science Education.* SIGCSE '81. St. Louis, Missouri, USA: Association for Computing Machinery, 1981, pp. 21–25. ISBN: 0897910362. DOI: `10.1145/800037.800955`. URL: `https://doi.org/10.1145/800037.800955`.

[13] J.A.W. Faidhi and S.K. Robinson. "An empirical approach for detecting program similarity and plagiarism within a university programming environment". In: *Computers & Education* 11.1 (1987), pp. 11–19. ISSN: 0360-1315. DOI: `https://doi.org/10.1016/0360-1315(87)90042-X`. URL: `https://www.sciencedirect.com/science/article/pii/036013158790042X`.

[14] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. "The Program Dependence Graph and Its Use in Optimization". In: *ACM Trans. Program. Lang. Syst.* 9.3 (July 1987), pp. 319–349. ISSN: 0164-0925. DOI: `10.1145/24039.24041`. URL: `https://doi.org/10.1145/24039.24041`.

[15] M. Joy and M. Luck. "Plagiarism in programming assignments". In: *IEEE Transactions on Education* 42.2 (May 1999), pp. 129–133. ISSN: 1557-9638. DOI: `10.1109/13.762946`.

[16] Elmar Juergens et al. "Do Code Clones Matter?" In: *Proceedings of the 31st International Conference on Software Engineering.* ICSE '09. USA: IEEE Computer Society, 2009, pp. 485–495. ISBN: 9781424434534. DOI: `10.1109/ICSE.2009.5070547`. URL: `https://doi.org/10.1109/ICSE.2009.5070547`.

[17] A. B. Kahn. "Topological Sorting of Large Networks". In: *Commun. ACM* 5.11 (Nov. 1962), pp. 558–562. ISSN: 0001-0782. DOI: `10.1145/368996.369025`. URL: `https://doi.org/10.1145/368996.369025`.

[18] Oscar Karnalim. "Detecting source code plagiarism on introductory programming course assignments using a bytecode approach". In: *2016 International Conference on Information & Communication Technology and Systems (ICTS).* 2016, pp. 63–68. DOI: `10.1109/ICTS.2016.7910274`. URL: `https://ieeexplore.ieee.org/document/7910274`.

[19] Pascal Krieg. *Preventing Code Insertion Attacks on Token-Based Software Plagiarism Detectors.* Bachelor's Thesis. Karlsruhe, Germany, Sept. 2022.

[20] Chao Liu et al. “GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis”. In: *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’06. Philadelphia, PA, USA: Association for Computing Machinery, 2006, pp. 872–881. ISBN: 1595933395. DOI: 10.1145/1150402.1150522. URL: https://doi.org/10.1145/1150402.1150522.

[21] Kevin Ly. “Normalizer: Augmenting Code Clone Detectors Using Source Code Normalization”. MA thesis. San Luis Obispo: California Polytechnic State University, Mar. 2017.

[22] Leonardo Mariani and Daniela Micucci. “AuDeNTES: Automatic Detection of TeNtative Plagiarism According to a REference Solution”. In: *ACM Trans. Comput. Educ.* 12.1 (Mar. 2012). DOI: 10.1145/2133797.2133799. URL: https://doi.org/10.1145/2133797.2133799.

[23] Raina Mason and Simon. “Introductory Programming Courses in Australasia in 2016”. In: *Proceedings of the Nineteenth Australasian Computing Education Conference*. ACE ’17. Geelong, VIC, Australia: Association for Computing Machinery, 2017, pp. 81–89. ISBN: 9781450348232. DOI: 10.1145/3013499.3013512. URL: https://doi.org/10.1145/3013499.3013512.

[24] Matija Novak, Mike Joy, and Dragutin Kermek. “Source-Code Similarity Detection and Detection Tools Used in Academia: A Systematic Review”. In: *ACM Trans. Comput. Educ.* 19.3 (May 2019). DOI: 10.1145/3313290. URL: https://doi.org/10.1145/3313290.

[25] José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. “PROGpedia: Collection of source-code submitted to introductory programming assignments”. In: *Data in Brief* 46 (2023), p. 108887. ISSN: 2352-3409. DOI: https://doi.org/10.1016/j.dib.2023.108887. URL: https://www.sciencedirect.com/science/article/pii/S2352340923000057.

[26] A. Parker and J.O. Hamblen. “Computer algorithms for plagiarism detection”. In: *IEEE Transactions on Education* 32.2 (May 1989), pp. 94–99. ISSN: 1557-9638. DOI: 10.1109/13.28038.

[27] Lutz Prechelt and Guido Malpohl. “Finding Plagiarisms among a Set of Programs with JPlag”. In: *Journal of Universal Computer Science* 8 (Mar. 2003). URL: https://www.researchgate.net/publication/2832828_Finding_Plagiarisms_among_a_Set_of_Programs_with_JPlag.

[28] Chanchal K. Roy and James R. Cordy. “NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization”. In: *2008 16th IEEE International Conference on Program Comprehension*. 2008, pp. 172–181. DOI: 10.1109/ICPC.2008.41.

[29] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. “Winnowing: Local Algorithms for Document Fingerprinting”. In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data.* SIGMOD ’03. San Diego, California: Association for Computing Machinery, 2003, pp. 76–85. ISBN: 158113634X. DOI: 10.1145/872757.872770. URL: https://doi.org/10.1145/872757.872770.

[30] Shahriar Shamsian et al. “Teaching Large Computer Science Classes”. In: *2016 ASEE Annual Conference & Exposition.* 10.18260/p.26034. https://peer.asee.org/26034. New Orleans, Louisiana: ASEE Conferences, June 2016.

[31] Pengcheng Wang et al. “CCAligner: A Token Based Large-Gap Clone Detector”. In: *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE).* 2018, pp. 1066–1077. DOI: 10.1145/3180155.3180179.

[32] Tiantian Wang, Xiaohong Su, and Peijun Ma. “Program Normalization for Removing Code Variations”. In: *2008 International Conference on Computer Science and Software Engineering.* Vol. 2. 2008, pp. 306–309. DOI: 10.1109/CSSE.2008.957.

[33] Debora Weber-Wulff, Katrin Köhler, and Christoph Möller. *Collusion Detection System Test Report 2012.* Tech. rep. Berlin, Germany: Hochschule für Technik und Wirtschaft Berlin, 2012. URL: https://plagiat.htw-berlin.de/collusion-test-2012/.

[34] Mark Weiser. “Program Slicing”. In: *IEEE Transactions on Software Engineering* SE-10.4 (1984), pp. 352–357. DOI: 10.1109/TSE.1984.5010248.

[35] Martin White et al. “Deep Learning Code Fragments for Code Clone Detection”. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering.* ASE 2016. Singapore, Singapore: Association for Computing Machinery, 2016, pp. 87–98. ISBN: 9781450338455. DOI: 10.1145/2970276.2970326. URL: https://doi.org/10.1145/2970276.2970326.

[36] Michael Wise. “String Similarity via Greedy String Tiling and Running Karp-Rabin Matching”. In: *Unpublished Basser Department of Computer Science Report* (Jan. 1993).

[37] Fangfang Zhang et al. “Program Logic Based Software Plagiarism Detection”. In: *2014 IEEE 25th International Symposium on Software Reliability Engineering.* Nov. 2014, pp. 66–77. DOI: 10.1109/ISSRE.2014.18.

A. Appendix

The evaluation results for task 19 can be found in the following. They are similar to those for task 56.

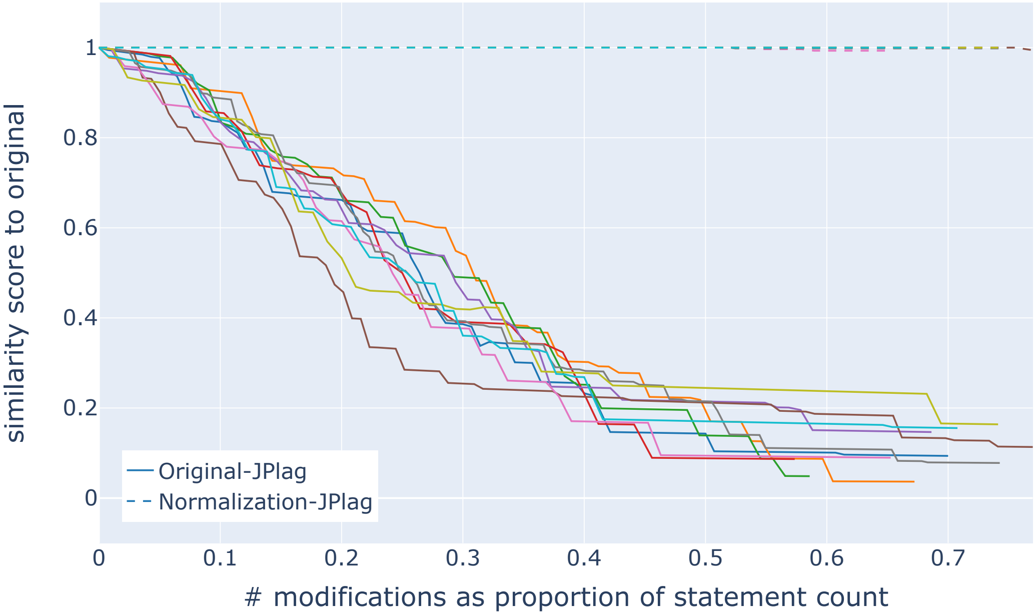

Figure A.1.: The SSO during the process of creating ten random plagiarized submissions from task 19 using insertion.

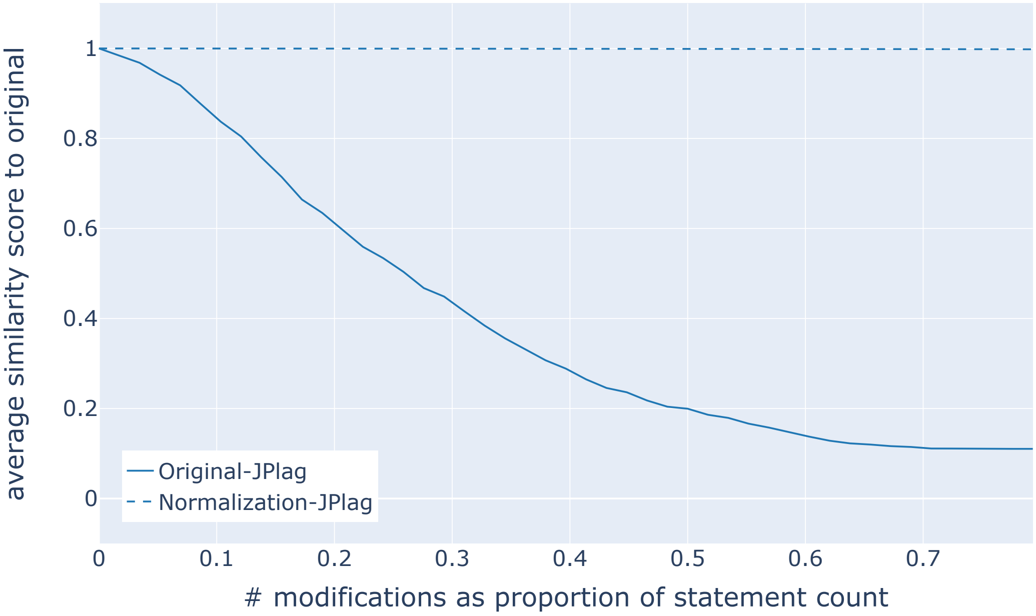

Figure A.2.: The average SSO during the process of creating all plagiarized submissions from task 19 using insertion.

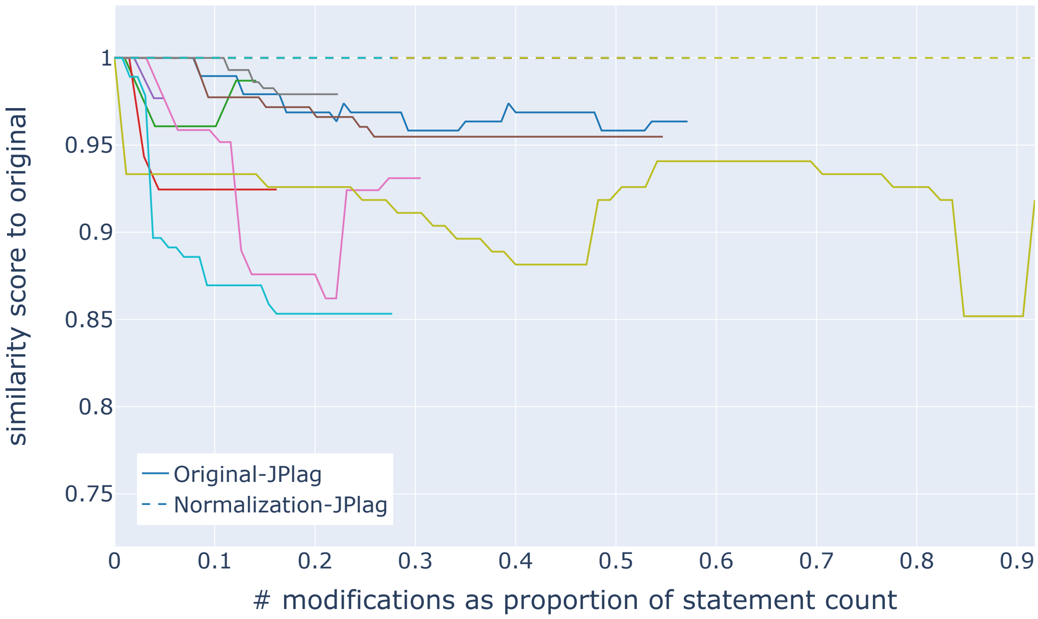

Figure A.3.: The SSO during the process of creating ten random plagiarized submissions from task 19 using reordering.

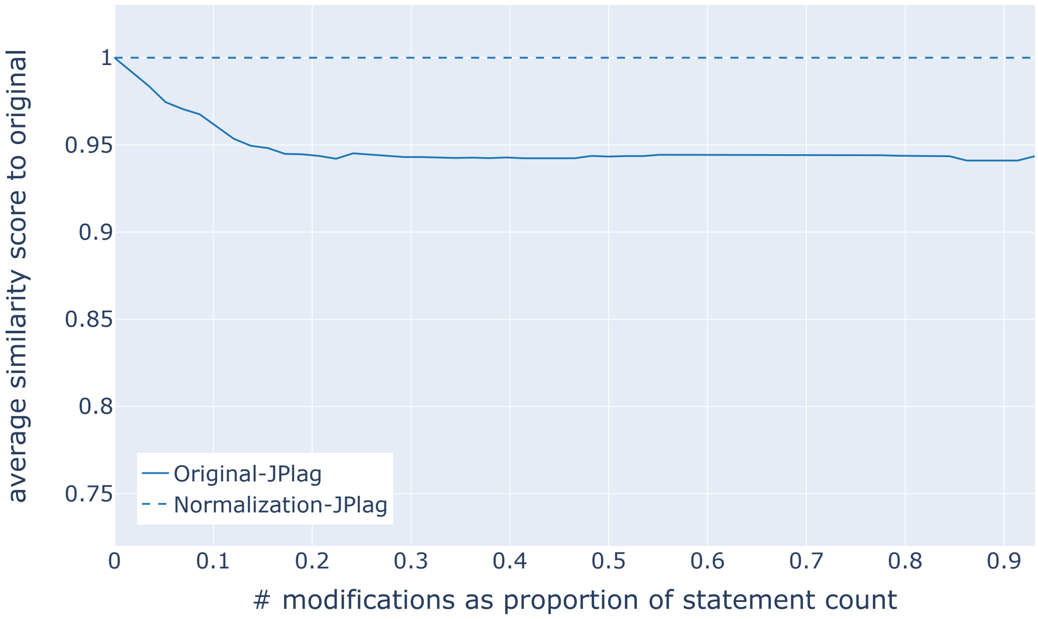

Figure A.4.: The average SSO during the process of creating all plagiarized submissions from task 19 using reordering.

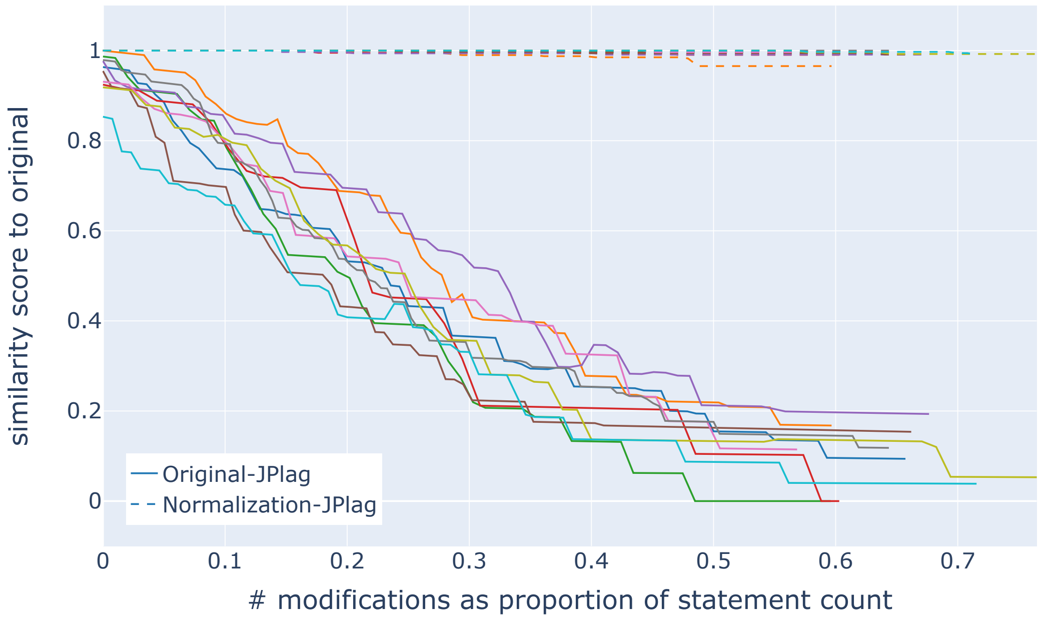

Figure A.5.: The SSO during the process of creating ten random plagiarized submissions from task 19 using reordering followed by insertion. Only the insertion stage is shown, for the reordering stage, see Figure A.3.

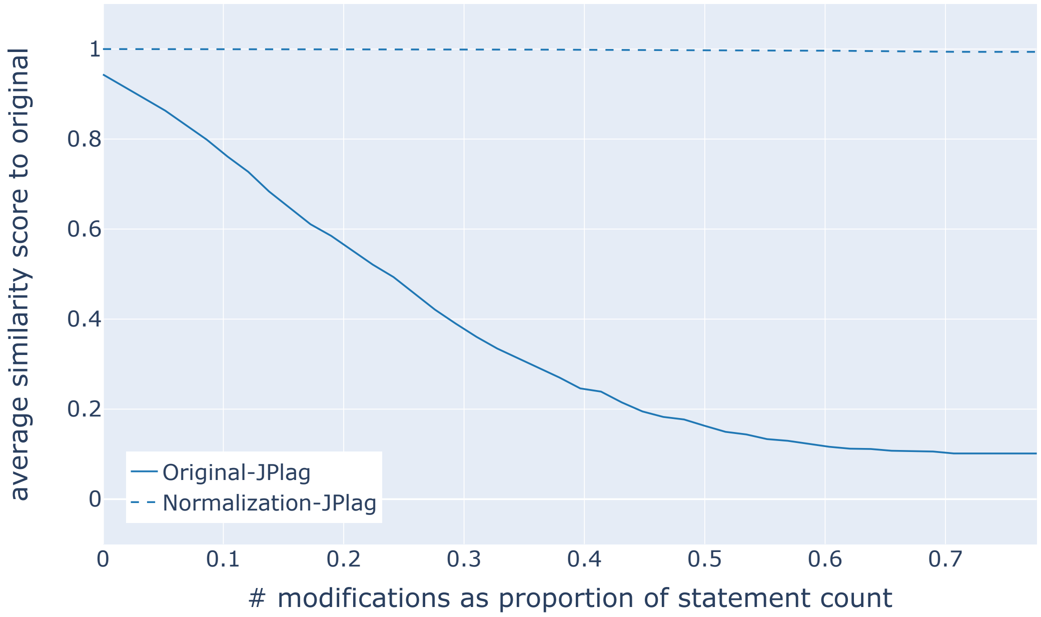

Figure A.6.: The average SSO during the process of creating all plagiarized submissions from task 19 using reordering followed by insertion. Only the insertion stage is shown, for the reordering stage, see Figure A.4.

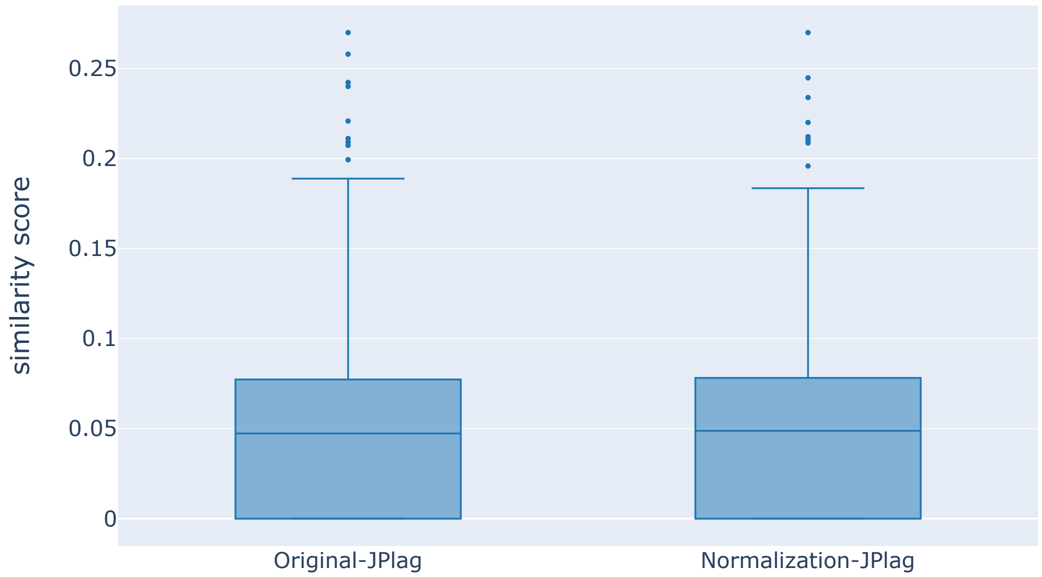

Figure A.7.: The similarity scores of the submission pairs in task 19, which are all non-plagiarized.

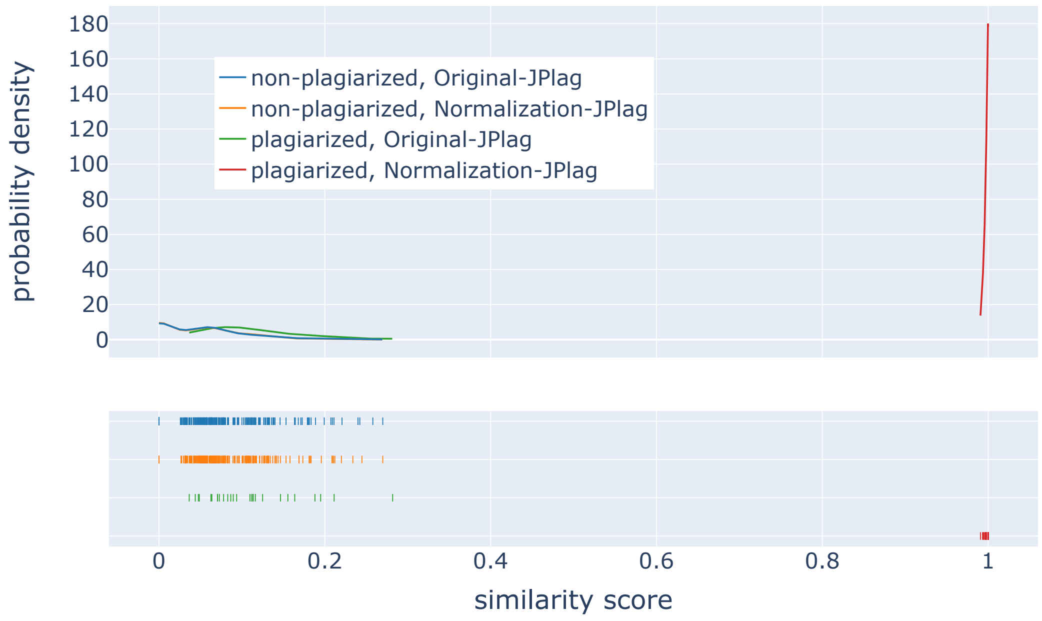

Figure A.8.: The similarity score distribution of the non-plagiarized and the plagiarized pairs from task 19. The plagiarized pairs each consist of an original and a plagiarized from it using insertion.

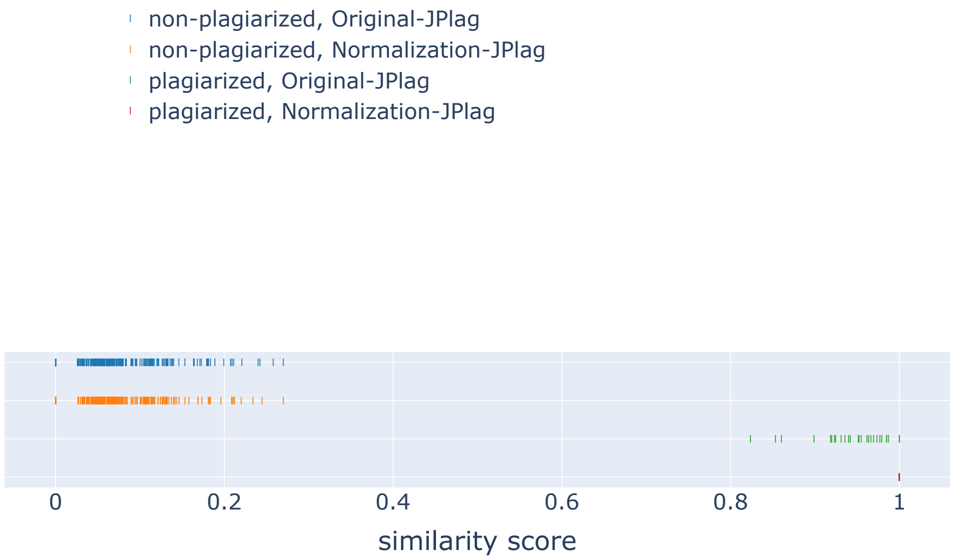

Figure A.9.: The similarity score distribution of the non-plagiarized and the plagiarized pairs from task 19. The plagiarized pairs each consist of an original and a version plagiarized from it using reordering. As all Normalization-JPlag scores for plagiarized pairs are 1 no probability density can be calculated.

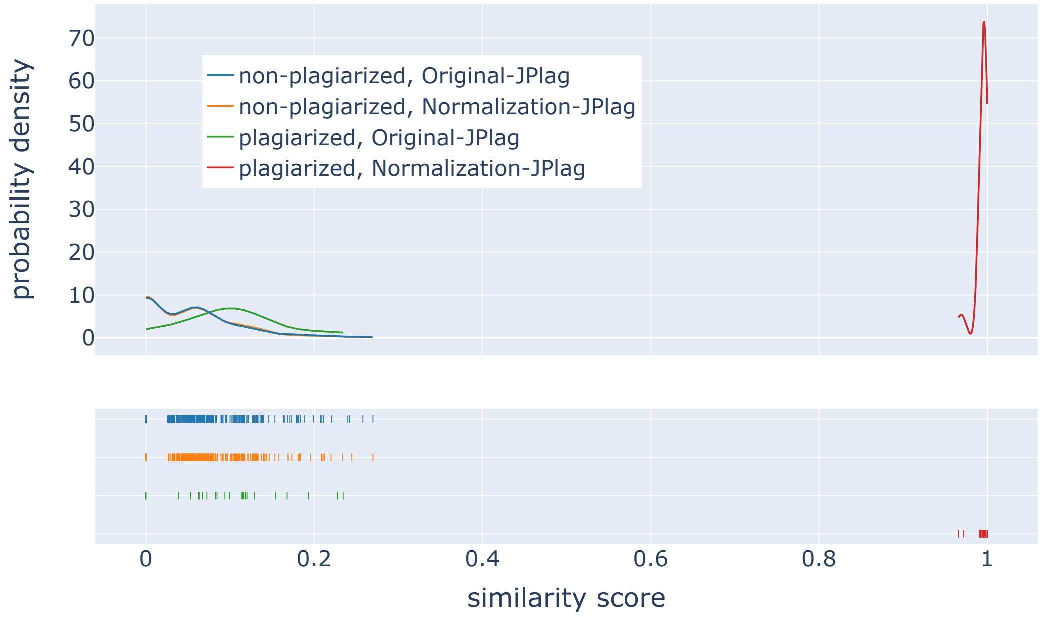

Figure A.10.: The similarity score distribution of the non-plagiarized and the plagiarized pairs from task 19. The plagiarized pairs each consist of an original submission and one plagiarized from it using reordering followed by insertion.

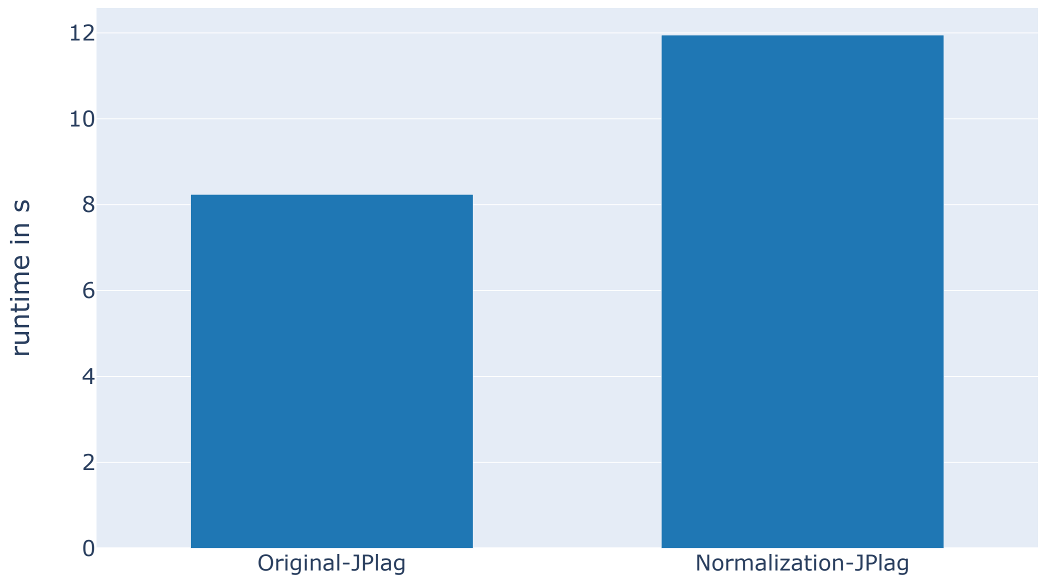

Figure A.11.: The runtime for an upsized version of task 19 as measured over 100 runs on an Apple M1 Pro with clustering disabled.