Investigating Cheating in Programming Assignments with Distance-based Similarity Analysis of Strings and Abstract Syntax Trees

Duke University Electrical & Computer Engineering
Graduation With Departmental Distinction Thesis

Natalia Androsz
advised by Dr. Andrew Hilton
November 2020

1 Introduction

Plagiarism and cheating are serious concerns for academic institutions. Duke University prides itself on its Student Honor Code, which condemns unauthorized collaboration, plagiarism, and cheating on all assignments. Although students agree to these standards upon enrolling in the university, students continue to self-report acting in academically dishonest ways in national surveys [1]. The primary deterrents for academic dishonesty are effective plagiarism detection tools. While plagiarism detection tools are quite robust for written assignments, there is a relative dearth of plagiarism tools intended for identifying cheating in programming assignments. In fact, most professors depend on a single tool, Moss [2]. Despite the existence of Moss, analyzing programming assignments for cheating is a time-consuming process [3]. The burden is on professors to both understand Moss output and compile it into meaningful evidence of cheating. The compiled evidence must then be presented to students, deans, and university administrators. University administrators are typically individuals without a technical background in programming. For this reason, it can be difficult to explain how a pair of code submissions is the product of plagiarism. Beyond simple modifications such as changing comments and variable names, there is a large set of structural manipulations that students can apply to copied code to make programs *look* different and *execute* the same. Consider the following code snippets which print the message "Hello world!" 10 times.

```
for(int i = 0; i < 10; i++){
  cout << "Hello world!";
}
```

```
int i = 0;
while(i < 10) {
  printf("Hello world!");
  i++;
}
```

Figure 1: Functionally equivalent pieces of code.

Individuals with a programming background would identify these two pieces of code as functionally equivalent. However, those without a programming background will most likely find these code snippets quite different.

This paper describes the framework for a new plagiarism detection tool for programming assignments. Lichen (Let's Investigate Cheating with Human-understandable Explanations and iNsights) aims to improve on the shortcomings of Moss in order to make it easier for professors to identify and report academic dishonesty in programming assignments.

2 Moss

Moss (Measure of Software Similarity) is a cheating detection tool developed in 1994 by Alex Aiken, associate professor of Computer Science at UC Berkeley [2]. Moss works with programs written in C, C++, Java, Pascal, Ada, and other languages. For a pool of student submissions, Moss compares all pairs of submissions for similarity. For each pair of programs, Moss reports the number of matching lines and the overall degree of similarity.

2.1 Winnowing Algorithm

Moss uses a *winnowing* algorithm to find matching sequences between submission files [2]. A high-level overview of this algorithm is provided below.

1. Determine the *fingerprint* of each submission.
   (a) Tokenize each student submission.
   (b) Divided the tokenized file into k-grams (contiguous substrings of length k).
   (c) Compute the hash values for each k-gram.
   (d) Select a subset of the file's k-grams. This is the *fingerprint* of the file.
2. *Winnow* the fingerprints.
   (a) Construct a window of consecutively defined hashes with size w by w, where w is the number of submissions.
   (b) Select the minimum hash value of each window, this is the new *winnowed fingerprint* of the document.
3. Sort and compare the winnowed fingerprints.

2.2 Shortcomings

Moss has several noteworthy shortcomings. These shortcomings provide insight into Lichen's goals.

1. **Moss does not provide deep insight into why a submission pair was flagged as similar.** Moss highlights portions of student code that were identified as similar but does not provide insights into why any particular segment was highlighted. For example, Moss does not specify whether a particular similarity is unique to a pair of submissions or if the similarity appears in multiple submissions. The opaqueness of Moss results means professors must dedicate additional time to deciding if Moss output is indicative of cheating or not.
2. **Moss cannot identify simple structural manipulations as plagiarism.** Moss has difficulty identifying similarities between code samples that have minor structural differences that do not impact the execution of a program [4]. This is the primary way that students have learned to beat Moss. A trove of online resources exist that describe, in detail, how to evade the Moss by making small structural manipulations to copied code [4, 5].
3. **Moss does not consider student-generated identifiers nor comments.** When code is tokenized, student-created identifiers such as variable names and function names are obscured. Comments are also discarded. While tokenization allows Moss to target the structure of student code, it also discards a large data set that can be used for additional similarity analysis. This loss is particularly tragic in the case of comments, which are written in plain English and should therefore be the most unique component of student code submissions.

3 Goals

From the previous discussion of Moss's shortcomings, we can define a set of desired characteristics for Lichen.

3.1 Analyze a variety of program fragments for similarity.

Lichen's first objective is to analyze more than just the structure of student code. Every student code submission can be broken down into sets of *fragments*. Fragments are sets of data that can be the targets of separate similarity analyses. Examples of fragments are comments, string literals, variable names, and abstract syntax trees.

3.2 Identify similar and rare fragment pairs.

Lichen aims to determine how *similar* fragments are to each other and how *dissimilar* they are from others fragments. The problems of *similarity* and *rarity* are quite different. As a motivating example, consider two students who have chosen the unusual variable name `zebra`[1]. This string pair has high *similarity*, however, the pair's *rareness* is determined by the frequency of variables named `zebra` in the greater submission pool. Consider the following scenarios.

1. If *only* these two students have variables named `zebra`, then this fragment pair is extremely interesting. This provides strong evidence that the submission pair may be the result of cheating.
2. These are the only two students with variables named `zebra`, however, there are other submissions with variables such as `zebra1`, `Zebra`, `Zebra2`. This may indicate a cluster of submission pairs that are the product of cheating, or it may indicate that this fragment is not particularly interesting. In other words, the number of submissions and how similar/dissimilar variable names in those submissions are determines whether the `zebra` pair is interesting or not.
3. If *all* students have a variable named `zebra`, then this fragment pair is not interesting and provides no evidence that the submission pair is the result of cheating.

3.3 Quantify the similarity and rarity of a submission pair.

Lichen intends to quantify the similarity and rarity of submissions pairs. This quantification will describe the likelihood that the similarity and rarity of all fragment pairs within a submission pair occurred by chance. It is important to note here that this probability is *not* the probability that a pair of students cheated on an assignment. Rather it is the probability that the magnitude of the similarity and rarity of this pair can be prescribed to chance alone. Low probability submission pairs are not guaranteed to be the result of cheating. There can be other reasonable explanations for the observed similarity and rarity in a submission pair. Faculty using Lichen will have to examine the results and decide for themselves whether the similarities and rarities are the results of cheating.

This probability can then be used to calculate the number of submission pairs that are expected to display this degree of similarity and rarity in a population of a particular size. An expected value of 1 is not at all compelling; however, an expected value of one in one million is quite compelling. By using the expected number of submission pairs, we account for similarities occurring by chance

[1] This example comes from a real case of cheating. The first time my advisor caught a pair of students cheating was when both students named variables `zebra` in an assignment unrelated to zebras.

more often in a larger number of submissions. For example, 100 students form 4950 pairs of submissions. A pair with probability 1/4,950,000 would have an expected value of 1/1000. However, if there were 1000 students, resulting in 499,500 pairs, then the same probability would have an expected value of only 1/10.

3.4 Provide insight into evidence.

Although the submission pairs that Lichen finds statistically unlikely are not guaranteed to be the result of cheating, we hope to facilitate a professor's understanding of Lichen results by creating a human-readable report that explains why each submission pair was identified as interesting. These insights will be presented in plain-English and in a format that students, administrators, and educators alike can understand. This report aims to decrease the amount of time that a professor needs to spend deciphering the results of our cheating detector. We also intend for this report to be written in such a way that it could be handed directly to an accused student, the university administration, or other faculty should a professor decide the evidence is compelling enough to accuse a pair of students of cheating.

4 Lichen

Lichen's architecture is divided into 6 phases: Build, Compare, Measure, Score, Calculate, and Report. Each phase is described in more detail in the sections that follow. A general overview of each phase is provided here. The overall architecture is also shown Figure 2.

1. **Build.** Convert student code submissions into sets of fragments to be analyzed.
2. **Compare.** Compare every fragment to all other fragments in every submission. This phase computes the *distance* of every pair of fragments.
3. **Measure.** Measure the distance of fragment pairs from other fragments in the population. This phase computes the *near set* and *frontier sets* of each fragment pair. At a high level, *near* and *frontier sets* are sets of fragments located at increasing distances away from the pair itself. These terms are defined in more detail later in the paper.
4. **Score.** Score fragment pairs based on the results of the Compare and Measure phase. This score captures the similarity and rarity of a fragment pair in a single value.
5. **Calculate.** Convert the results of the Score phase into probabilities. Once the probabilities of all fragment pairs have been computed, filter independent fragment pairs for each submission pair and compute the expected number of submission pairs with this many independent, similar, and rare fragment pairs.
6. **Report.** Create human-readable reports for low-probability submissions pairs explaining the compiled evidence.

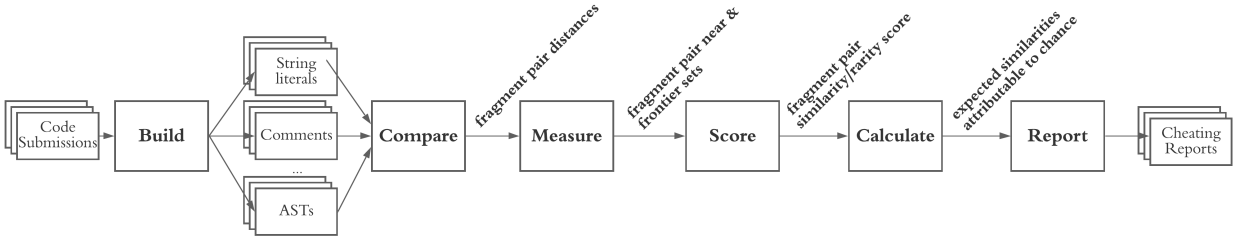

Figure 2: Overview of Lichen architecture.

4.1 Build

Lichen's first goal is to analyze a variety of program fragments for similarities. In order to do this, Lichen must break down student submissions into sets of fragments to compare for similarity. This is accomplished by partially compiling a student code submission using ANTLR, a parser generator for reading, processing, executing, and translating structured text or binary files [6]. The first step of compilation is lexing [7]. In this step, code is tokenized and white spaces and comments are removed. In order to analyze the comments in student code submissions, we needed to lexify student code in two different ways.

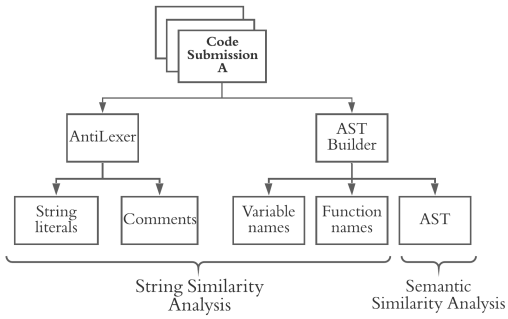

Figure 3: Build phase.

One lexer tokenizes student submissions in the traditional way—white spaces and comments are ignored. The product of this lexer is used in the second step of compilation to construct an Abstract Syntax Tree (AST). An AST is a tree representation of the source code [7]. An example of code compiled into a simplified AST is shown in Figure 4. Each node of the tree corresponds to a functional element in the code. This tree provides Lichen insight into the execution of the student program and will be used for semantic similarity analysis. The AST also provides us variable names and function names for string similarity analysis.

```
int main(void) {
  printf("Hello world");
  return 0;
}
```

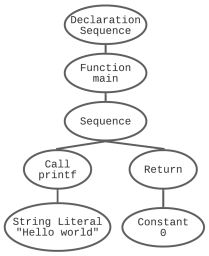

Figure 4: AST of a simple program.

The second lexer does the opposite of a standard lexer—student code is ignored and comments are not. This backward approach led us to name this second lexer the *antilexer*. The antilexer produces comments and string literals for analysis. String literals were included in the antilexer because they have relatively simple grammar rules—anything between double quotes is considered a string literal. On the other hand, defining grammar rules in the antilexer for variable names and function names would have greatly complicated the antilexer grammar. For this reason, variable and function names are derived from the AST while comments and string literals are derived from the antilexer.

4.2 Compare

The Compare phase is responsible for quantifying the distance between all fragment pairs. There are two distinct categories of distance metrics that Lichen currently supports: string distance metrics and tree distance metrics. Our initial approach to string and tree distance metrics are described in the sections below. At the end of this phase, Lichen has created a ranked list of the best matches for every fragment in every submission. This ranked list contains at least one closest match from all other submissions in the population. A submission will not be represented in this list only if the unrepresented submission had no instance of a particular fragment. For example, in

a submission population of 100 submissions, if one submission did not have any comments, then every fragment in every submission would have 98 best matches in their ranked list instead of 99.

4.2.1 String Distance Metrics

String distance metrics are used to compute the difference between pairs of string literals, comments, variable names, and function names. There are many well-known string distance metrics. In preliminary testing of Lichen, string distance was computed using the Levenshtein distance metric. Levenshtein distance is computed by applying one of three transformations to a pair of strings: insertion, adding a new character; deletion, deleting a character; and substitution, replacing one character with another [8]. Each of these transformations increases the distance of a pair of strings by 1. Figure 5 provides an example of calculating the Levenshtein edit distance of two strings. There are many advantages and disadvantages to using different types of string distance metrics. We explored other options such as Cosine distance, Jaro-Winkler distance, and others [9]. This set of distance metrics is normalized and return distances between 0 and 1, where the former is the least similar value and the latter is the most similar value. We decided to use the Levenshtein distance metric because the reasoning behind the distance given to a pair of strings is more straightforward to understand. After testing the Lichen system we have identified some undesirable properties of the Levenshtein edit distance metric for our particular use case. For this reason we look to develop our own distance metric for strings in the future.

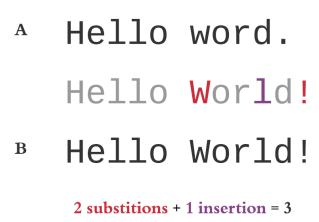

Figure 5: Levenshtein distance.

4.2.2 Tree Distance Metrics

Determining the distance of a pair of AST fragments is a more difficult problem than finding the distance between two strings. Unlike strings, there are no off-the-shelf options for tree distance metrics. As a result, we had to develop our own. We approached this problem knowing that we are trying to identify the intentional obfuscation of cheating by students. When students attempt to hide cheating, they apply a set of semantic-preserving transformations to their program so that it *executes* the same but *looks* different than the student they copied from. With this in mind, we developed a distance metric where Lichen applies a set of known semantic-preserving transformations to an AST pair until the two AST fragments become equal. In cases where trees are quite different, it is impossible to transform one submission into another with only semantic-preserving transformations. For this reason, Lichen may need to apply non-semantic-preserving transformations. Semantic preserving transformations contribute less to the overall distance between a pair of tree fragments than non-semantic-preserving transformations. There is also a set of transformations that may or may not preserve the semantics of the program. The three categories of tree transformations are summarized in Table 1.

Transformation Type	Cost	Examples
Semantic-preserving	low	For <->while loop, In-line function
Maybe semantic preserving	medium	Reorder statements, Loop transposition
Non-semantic-preserving	high	Adding a subtree, Removing a subtree

Table 1: Summary of three categories of tree transformations

Tree Distance Example

In order to better demonstrate Lichen's approach to finding the distance between two AST subtrees, consider the transformations that need to take place in order to make the code samples in Figure 6 equivalent. At first glance these programs may look quite different.

Code A

```
1  int square (int x) {
2    return x * x;
3  }
4
5  int main(void) {
6    for(int i = 0; i < 42; i++){
7        printf("%d: %d", i, square(i));
8    }
9  }
```

Code B

```
1  int main(void) {
2    uint32_t y = 0;
3    while (y < 42) {
4        int temp = y * y;
5        printf("%d=> %d", y, temp);
6        y++;
7    }
8  }
```

Figure 6: This pair of programs can be transformed into one another by applying six semantic-preserving AST transformations.

Figure 7 demonstrates how six transformations can be applied to Code A to turn it into Code B. Each transformation in Figure 7 is labeled with a letter A–G. Transformations A–G are described in detail below. For each transformation applied to Code A, a code snippet is provided to demonstrate how the previous version of Code A is modified. Changes at each step are highlighted in yellow. Note that these transformation do not actually occur on the file itself. Rather, these transformations are applied to the AST representation of the programs as shown in Figure 7.

(A) Extract `square` call to local variable.

```
1  int square (int x) {
2    return x * x;
3  }
4
5  int main(void) {
6    for(int i = 0; i < 42; i++){
7        int temp = square(i);
8        printf("%d: %d", i, temp);
9    }
10 }
```

On line 7 of Code A, a call is made to `square(i)`. This call is the second parameter of the function `printf`. Observe that on line 6 of Code B, there is a variable `temp` passed in as the second parameter to the same `printf` function. The first transformation applied to Code A is to extract the call to `square` to a variable declaration before the call to `printf`.

(B) In-line `square` function.

```
1  int square (int x) {
2    return x * x;
3  }
4
5  int main(void) {
6    for(int i = 0; i < 42; i++){
7        int temp = i * i;
8        printf("%d: %d", i, temp);
9    }
10 }
```

The `square` method in Code A does the same thing as the initializing statement of the variable `temp` in Code B on line 4. `square` is in-lined by replacing the call to it with the function's body.

(C) Remove the unused `square` function declaration.

```
1   int main(void) {
2     for(int i = 0; i < 42; i++){
3         int temp = i * i;
4         printf("%d: %d", i, temp);
5     }
6   }
```

Once the call to `square` has been in-lined, there are no other references to `square` in Code A. Therefore, we can remove the declaration of the `square` function entirely.

(D) Transform `for` to `while` loop.

```
1   int main(void) {
2     int i = 0;
3     while (i < 42){
4         int temp = i * i;
5         printf("%d: %d", i, temp);
6         i++;
7     }
8   }
```

A `for` loop can be transformed into a `while` loop without changing the execution of the program by rearranging statements in the for loop declaration: first, move the initialization of the variable `i`, `int i = 0`, outside the loop; second, move the statement that increments `i` with every iteration, `i++`, to the end of the loop's body.

(E) Rename `i` to `y`.

```
1   int main(void) {
2     int y = 0;
3     while(y < 42){
4         int temp = y * y;
5         printf("%d:%d", y, temp);
6         y++;
7     }
8   }
```

Next, the name of variable `i` is changed to `y`. This involves changing the declaration of the variable as well as all references to the variable later in the program.

(F) Swap similar C++ type `int` with `uint32_t`

```
1   int main(void) {
2     uint32_t y = 0;
3     while(y < 42){
4         int temp = y * y;
5         printf("%d:%d", y, temp);
6         y++;
7     }
8   }
```

The renamed variable `y` is declared with type `int`. On line 2 of Code B, the variable `y` is declared with type `uint32_t`. These C++ types are functionally equivalent when the number being stored in them are positive [10]. They can be interchanged without changing the behavior of the program.

(G) Make the string literals in the `printf` function calls equivalent.

```
1   int main(void) {
2     uint32_t y = 0;
3     while(y < 42){
4         int temp = y * y;
5         printf("%d=> %d", y, temp);
6         y++;
7     }
8   }
```

The only remaining difference between Code A and Code B is the string literal printed by the call to `printf`. Code A has "$\%d : \%d$" while Code B has "$\%d => \%d$". Once this transformation is applied, the submissions are equivalent. Note that this transformation is non-semantic-preserving.

Each transformation contributes some amount to the overall distance of Code A and Code B. A hypothetical distance for Code A and Code B is computed in Table 2. Transformations A–F have low costs because they are semantic-preserving. Transformation G has a higher cost because it is non-semantic-preserving.

Transformation	(A)	(B)	(C)	(D)	(E)	(F)	(G)	Total distance
Cost	1	1	1	1	1	1	3	9

Table 2: Each transformation contributes to the distance between AST fragments.

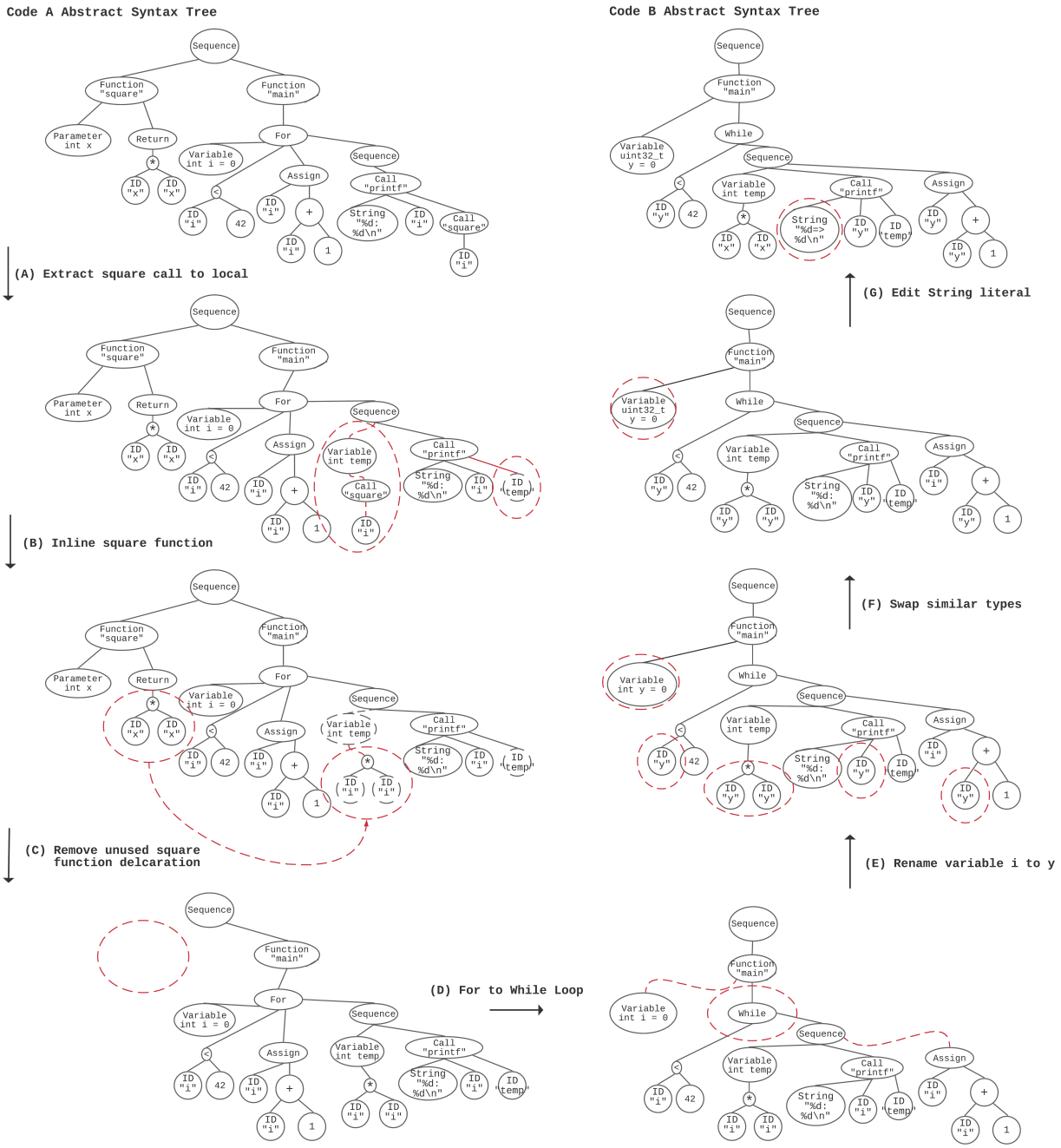

Figure 7: The transformations described on pages 7–8 are applied on the AST representations of Code A and Code B.

4.3 Measure: Near & Frontier Sets

At this point in the analysis, Lichen has quantified the distance between every pair of fragments. The next step is to determine how rare the similarity between a pair of fragments is. Rarity of a fragment pair is determined by building the near set and frontier sets for all fragment pairs. As shown in Figure 8, the **near set** of (A, B), an arbitrary fragment pair with distance 4, is comprised of all fragments that are at most a distance of 4 from either A or B. The near set is therefore {C, D} because fragment C is a distance of 3 from B, and fragment D is a distance of 4 from A. The **frontier sets** of (A, B) are constructed by finding all fragments that are distance x away from the pair, where x is greater than the distance between (A, B). A frontier set at distance x will be denoted as F_x. In Figure 8, the frontier set at distance 5, F_5, is empty. The frontier set at distance 6, F_6, is {E, F}. Both E and F are a distance of 6 away from either A or B. To build a complete set of frontier sets for (A, B), the distance would continue to be incremented until a fragment from all submissions has been included in a frontier set.

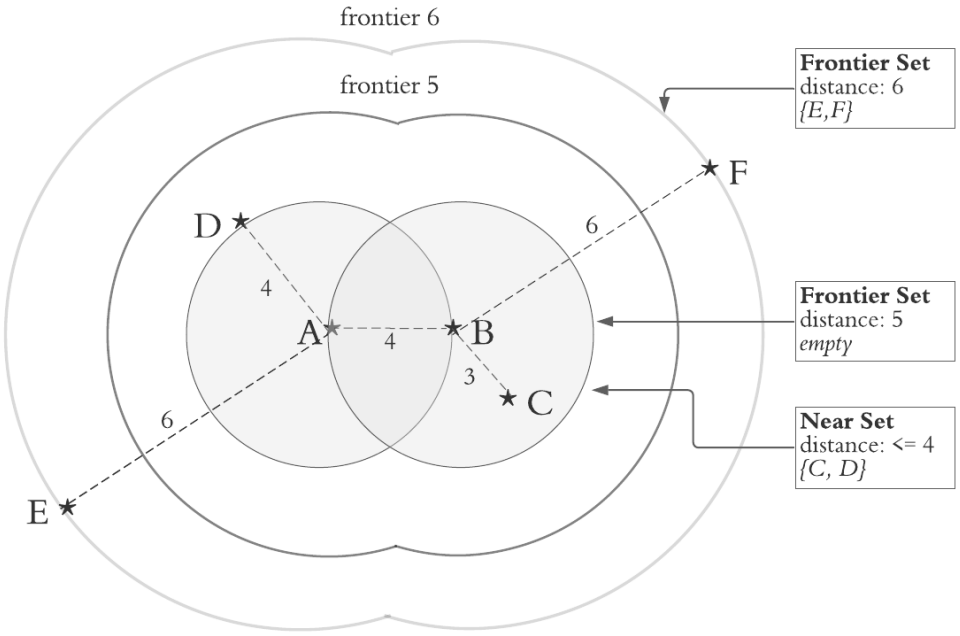

Figure 8: Illustration of *near* and *frontier* sets for an arbitrary fragment pair (A,B)

When the distance of a pair of fragments is considered in conjunction with its near and frontier sets, the rarity of the pair becomes more clear. This relationship is shown in Figure 9. Figure 9 is best explained by using the `zebra` example provided in the Goals section of this paper. The top left quadrant of this chart describes scenario 1. Two students used an extremely similar fragment pair such as `zebra` and `Zebra`, and there were no other strings similar to them in the population. This fragment pair is extremely interesting. The top right quadrant describes scenario 2. This diagram shows a similar string pair such as `zebra` and `Zebra`, but there are many other `zebra`-like identifiers nearby: another two variable names with a distance of 0 or 1 from A or B are found in the near set and all remaining best matched strings in the population are found within two frontier sets of the (A, B). This fragment pair is less interesting than the previous example. The bottom left quadrant describes a string pair that has a larger distance—for example, `zebra` and `ZebraParty`. This pair has no other strings in its near set, and all other strings in the population are far away. This fragment pair may or may not be interesting depending on how large the distance between

the fragments is and how close the nearby strings are. The bottom right quadrant shows scenario 3 for a fragment pair such as `zebra` and `ZebraParty` where all other strings in the population are within the near set of this pair. This pair is not interesting. Most fragments will fall into this last category.

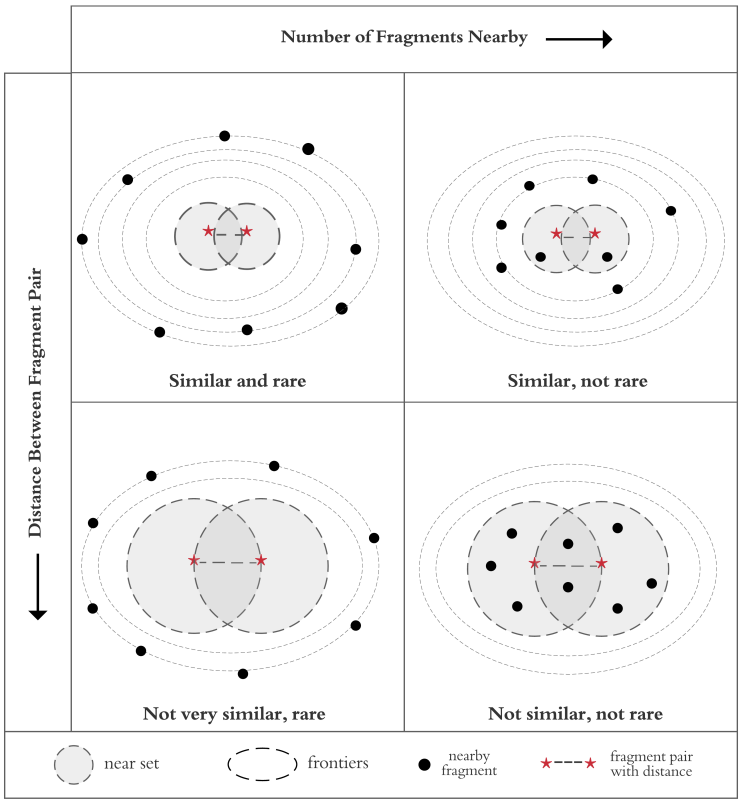

Figure 9: The relationship between the distance of a fragment pair and the number of nearby fragments allows Lichen to identify the similarity and rarity of a fragment pair.

4.4 Score

The Score phase aims to combine the results of the Compare and Measure phases into a single number per fragment pair. We created a scoring function based on the principles described in Figure 9. In particular, we want the result of this function to be higher for fragment pairs with near/frontiers sets in the upper left of this figure. We want this function to output lower scores for fragment pairs with near/frontier sets found bottom or right-hand portions of the figure. Accordingly, we built a function that for any fragment pair, starts from the pair's distance d and near set N, and iterates outwards to further and further frontier distances. At each frontier, the score is increased. The amount the score increases depends on the distance of the frontier and the number of fragments that can be found up to that frontier. Both of these variables impact the score added at a particular frontier based some **score gradient function** $f(x)$, a decreasing function. The scoring gradient f

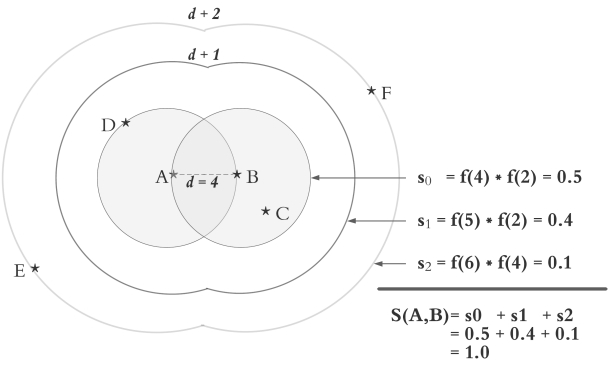

Figure 10: Scoring near and frontier sets.

will be discussed in more detail later in this section.

Figure 10 demonstrates how the near and frontier sets of an arbitrary fragment pair (A,B) are scored. The overall score of the fragment pair, *S(A,B)*, is sum of scores contributed by the near set and all frontier sets. First, the near set of the pair contributes a score s_0. This initial score is the product of *f(4)* and *f(2)* since the pair has a distance of 4 and its near set contains 2 fragments. Next, the frontier set at distance *5* is visited and contributes a score s_1. The value of s_1 is the product of *f(5)* and *f(2)* since the frontier's distance is 5 and there are no fragments at this frontier. The next frontier contributes score s_2. The value of s_2 is the product of *f(6)* and *f(4)* since the frontier's distance is 6 and the frontier set F_6 contains two fragments. This continues until we reach a fixed distance that is considered the maximum frontier from any fragment pair.

The score of a fragment pair (A,B) with distance d, near set N, and frontier sets $\{F_{d+1}, F_{d+2}, F_{d+3}, ...F_n\}$ can be summarized as:

$$S_{A,B} = \sum_{i=d}^{n} f(i) * f(|N \cup \bigcup_{k=d}^{i} F_k|) \tag{1}$$

4.4.1 Scoring Gradient Function

The scoring gradient function f can be any decreasing function. In other words, f must give higher weight to fragment pairs with low distance and few accumulated fragment pairs. When developing our own f we also realized that there is a point for both distance and number of nearby fragments where no additional information is gleaned. In the case of distance, this point can be thought of as the frontier distance at which a fragment pair would be more different than similar from fragments found at further frontiers. In the case of accumulated nearby fragments, this point can be thought of as the point at which a critical mass of submissions have been represented in the accumulated fragments. Nearby fragments found beyond this point merely reinforce the fact that the fragment pair is not rare. For this reason, we define the idea of a useless point. A **useless point** is defined as the point at which the score contributed by distance or accumulated nearby fragments goes to 0. We developed the following scoring gradient function f for initial tests of Lichen.

$$f(x) = \max(0, 1 - k * x) \tag{2}$$

In Equation 2, k is the slope between (0,1) and (useless point, 0); x is either the distance away from fragment pair or the number of accumulated nearby fragments. Observe that this function meets the expectations outlined above. The maximum value of the scoring function is 1. The function returns 1 if the distance or number of accumulated fragments is 0. Further, the value *f(x)* is inversely proportional to x.

Finally, it is important to note what we defined the useless points of distance and of accumulated nearby fragments to be. In the case of accumulated nearby fragments, the useless point was defined as one-fifth of the submissions in the population are represented within the accumulated nearby fragments. In the case of distance, we defined the useless point to be when the distance away from the fragment pairs has reached the average length of the fragment pair and all accumulated nearby fragments.

4.4.2 Scoring Case Study

In this case study we compare the behavior of the scoring function and the scoring gradient function on two pairs of comments from a real runs of Lichen. String similarity analysis was performed on the comments of an assignment with 146 unique student submissions, 3 of which

Comment 1	Comment 2	Levenshtein Distance	Score Gradient									Score
			Near Set	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	
`wrong command`	`wrong command`	0	1.0	.89	.82	.75	.40	.25	.15	.10	0	4.81
`wrong command`	`wrong commend`	1	.89	N/A	.82	.75	.40	.25	.15	.05	0	3.31

Table 3: Comparison of scoring of comment pair (`wrong command`, `wrong command`) and (`wrong command`, `wrong commend`).

had no comments. Note that the useless point for nearby fragments is therefore 29 comments. Each comment in each pair is identified with the submission it comes from in parentheses. Pair A, (`wrong command` (s324), `wrong command` (s121)), has distance 0. Pair B (`wrong command` (s324), `wrong commend` (s149)) has distance 1. Observe that `wrong command` (s121) will be in the near set of Pair B. For this reason we expect Pair A to be scored higher than Pair B.

The behavior of the score gradient function for Pair A and Pair B can be seen in Table 3 and Figure 11. Pair A has no comments in its near set and has a distance of 0. As a result, it receives the maximum score possible for its near set, 1. On the other hand, the distance of Pair B is larger and its near set contains one comment. Pair B's near set receives a lower score than Pair A's. For Pair A, F_1 contains one comment. F_2, F_3 are empty. Pair B does not have a F_1 since its near set exists at this distance. Pair B's F_2, F_3 are also empty. The scoring gradient function outputs steadily decreasing values for frontiers 1, 2, and 3. At frontier 4, 11 nearby comments are found for both Pair A and Pair B. Adding this quantity of nearby comments causes a sharp decrease in the output of the score gradient function between frontiers 3 and 4. After frontier 4, the score gradient function result continues decreasing at the same rate for both pairs until frontier distance 7. This pattern exists because the number of accumulated nearby comments and the distance are growing at the same rate for both pairs. At frontier distance 7, however, we see that Pair A's F_7

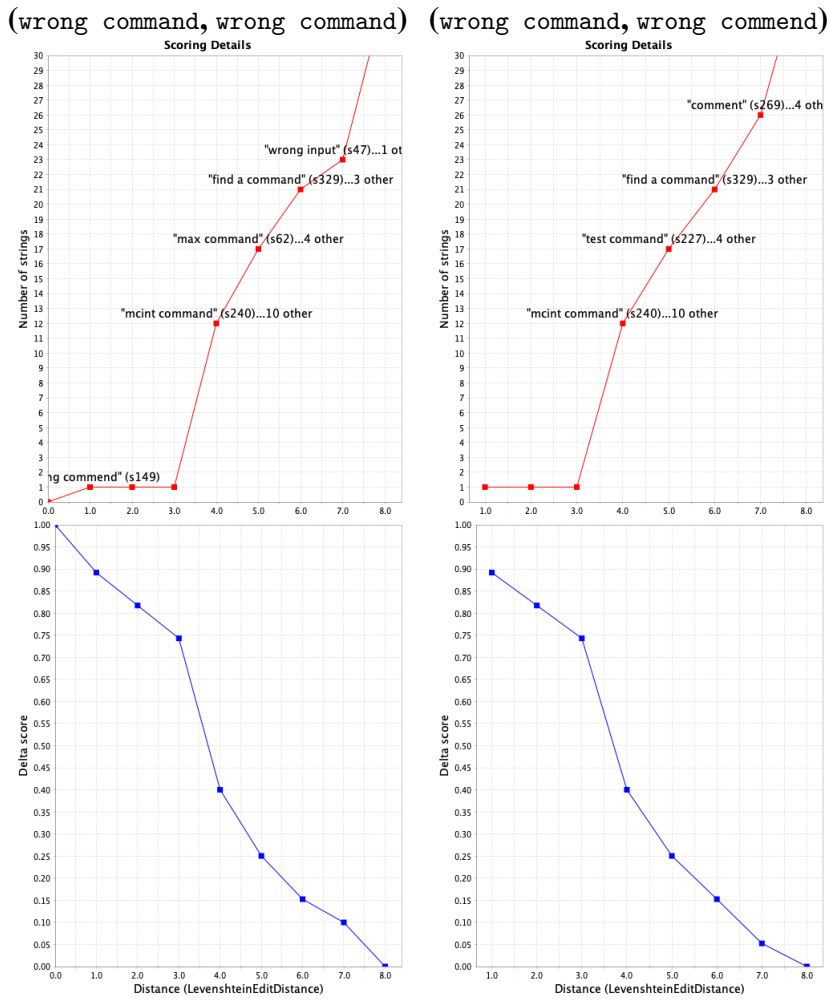

Figure 11: Comparison of the scoring of (`wrong command`, `wrong command`) (right) and (`wrong command`, `wrong commend`) (left). The top graphs show the accumulated nearby comments. The bottom graphs show how score gradient function scores each frontier.

contains 2 comments, while Pair B's F_7 contains 4 comments. For this reason the score gradient function outputs a higher score at frontier 7 for Pair A than Pair B. At frontier distance 8, Lichen stops iterating. Notice that the number of accumulated nearby comments exceeds the useless point of 29 at this frontier. As a result, the score gradient function returns 0. In the end, we can observe that the scores of Pair A and Pair B fulfill our expectations. Pair A is scored 4.81 and Pair B is scored 3.31.

4.5 Calculate

The Score phase converted the rarity and similarity of every fragment pair into a single number per pair. The Calculate phase now aims to convert each score into a probability describing the likeliness that the magnitude of a fragment pair's similarity and rarity can be attributed to chance. Low probability fragment pairs are kept as potential evidence of cheating for the submission pair the fragment pair originated from. An evidence pool is built up for every submission pair. This pool contains the most interesting fragment pairs for each submission pair. Once the evidence pool has been constructed, the likelihood that a particular submission pair's similarities can be attributed to chance is computed by taking the product of independent fragment pairs in its evidence pool. This processes is summarized in Figure 12.

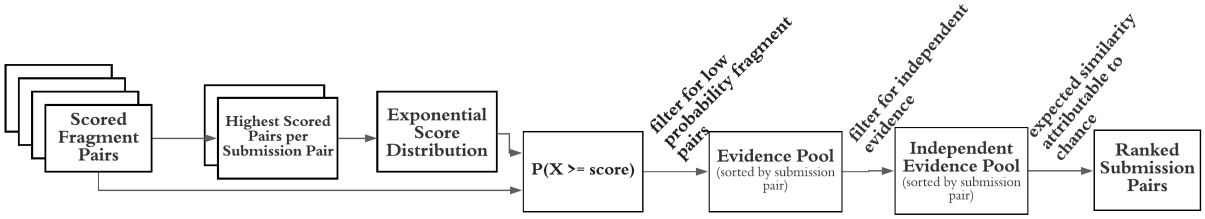

Figure 12: Calculate phase.

4.5.1 Score Distribution

Lichen creates an exponential distribution from the set of highest scored fragment pairs from every submission. The subset of highest scored fragment pairs is used to construct the distribution instead of the complete set of scored fragment pairs because we are ultimately looking for the probability that the similarity and rarity of all evidence for a *submission pair* can be attributed to chance. We are *not* looking for the probability of the similarity and rarity of the fragment pairs themselves. If we created the distribution from the larger population of scored fragment pairs, we would get unreasonable results. For this reason, only the highest scoring pairs are used to construct the distribution.

An exponential distribution is typically appropriate when used to model either the rate that events occur—for example, the time between earthquakes—or the intensity of an event—for example, the magnitude of earthquakes. Lichen's similarity and rareness scores measure the intensity of an event—the magnitude of the combined similarity and rarity of a fragment pair. The exponential distribution is therefore appropriate. The decision to use the exponential distribution is supported empirically by examining the observed scores and the exponential distribution fit to them. Consider the comparison of the observed and expected values of comment and string literal scores in a run of Lichen in Figure 13. The exponential distribution is then used to calculate $P(X >= \text{score})$ for each fragment pair. Fragment pairs that are outliers are added to the evidence pool of their submission pair.

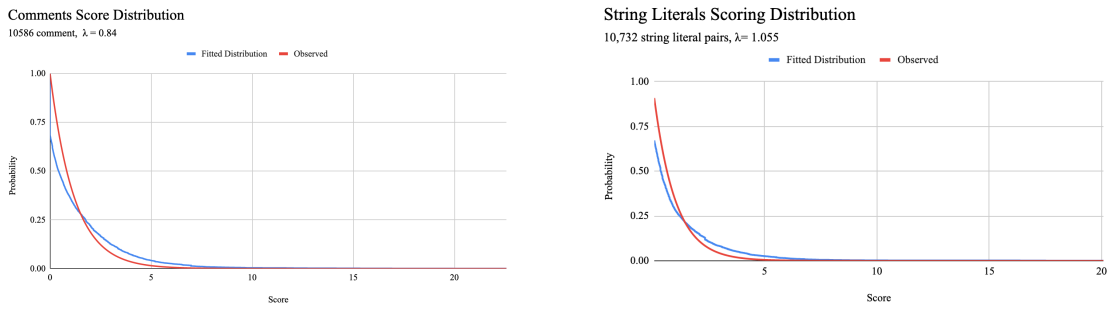

Figure 13: Score distribution approximately follows an exponential distribution.

Evidence Pool Submission A & B

	Submission A String	Submission B String	Probability of greater score
1	`zebra`	`Zebra`	0.03
2	`Zebra1`	`Zebra2`	0.05
3	`Hello!`	`hello`	0.006

Independent Evidence Submission A & B

	Submission A String	Submission B String	Probability of greater score
1	`zebra`	`Zebra`	0.03
3	`Hello!`	`hello`	0.006

Figure 14: Example of string independence filtering for Levenshtein edit distance metric.

4.5.2 Independent Evidence Filtering

At this point in the analysis, every submission pair has a pool of interesting evidence associated with it. The pool of evidence for every submission pair must now be filtered so that only independent evidence is used to compute the overall probability that the submission pair's similarity and rareness can be attributed to chance. Independence of fragment pairs is determined differently depending on the distance metric used and the type of fragment being considered. Consider the evidence pool for an arbitrary pair of submissions shown in Figure 14. The evidence pool contains three interesting string pairs. Observe that pairs 1 and 2 in the evidence pool are not independent. This is because if a student submission already contains one `zebra`-like variable name, then it is more likely to have other `zebra`-like variable names. Therefore, pairs 1 and 2 are not independent. We are not aware of any standard methods for determining the independence of two strings.

We define two strings to be independent of each other if their edit distance is more than half their average length. We define two string pairs to be independent when both strings in the pair are independent of each other. Put another way, taking $\perp$ to mean independence

$$(A_1, B_1) \perp (A_2, B_2) \iff \text{dist}(A_1, A_2) >= \text{length}(\frac{A_1 + A_2}{4}) \bigwedge \text{dist}(B_1, B_2) >= \text{length}(\frac{B_1 + B_2}{4}) \quad (3)$$

4.5.3 Ranking submission pairs

The likelihood that a submission pair's similarity and rarity can be attributed to chance is calculated by taking the product of the probabilities of its independent evidence. This probability is then multiplied by the total number of submissions pairs in the population. For a population of submissions of size N this would be $\frac{N*(N-1)}{2}$. This new value represents the number of submissions pairs in a population of size N that are expected to display this same degree of similarity and rarity. This value is computed for all submission pairs.

4.6 Report

In the final Lichen phase, human-readable reports are constructed for submission pairs whose similarity and rarity are unlikely to be attributed to chance. This report contains descriptions of

the analyses performed, interesting evidence gathered for each analysis, and supporting charts and figures. Figure 11 is an example of a chart that will appear in reports. Professors will be able to customize reports to include assignment descriptions and other course-specific information. This report will be written in language that can be easily understood by students, educators, and university administrators alike. Should a professor decide that the evidence described in the report is indeed indicative of cheating, the report can be submitted directly to university administrators.

5 Results

Preliminary testing of Lichen has yielded promising results. Lichen was tested on data from three real C/C++ assignments. The submission for each assignment were anonymized by my advisor. My advisor also provided an anonymized list of the submission pairs within each assignment that had been accused of cheating and successfully convicted of cheating on the assignment. Please note the expected cheating pairs list does not contain information about student pairs accused of cheating in semesters when my advisor was not teaching. The number of submissions for each assignment ranged from 146, 319, to 437. The expectations for running Lichen on these three assignments are summarized here.

1. **All expected cheating pairs are identified.** For a particular assignment, we expect Lichen to successfully identify all expected cheating student pairs as having similarities that are extremely unlikely to have occurred by chance.
2. **New cheating pairs are identified.** We also expect Lichen to identify pairs of submissions that likely cheated but were undetected by both Moss and graders of the assignment. In addition, we expect many high ranking student pairs that are not in out expected set will reflect students who cheated on the assignment during semesters when my advisor was not teaching.
3. **Low false positive count.** We expect few false positives. Ideally, false positives can be easily explained. We expect to gather insights about how to improve Lichen from these false positives.

One of the reasons we may be unable to achieve the above results is because Lichen currently only supports string similarity analysis of comments, string literals, function names, and variable names. We are unable to report the results of AST similarity analysis at this time. This is elaborated on in the Future Work section of this paper. Even though Lichen only performed string similarity analysis in these runs, the results are quite satisfactory. Consider as a case study the assignment with 146 submissions.

5.1 Case Study

This assignment comes from a graduate-level Electrical & Computer Engineering course at Duke University. This assignment has been given for multiple semesters of this course. Collaboration was explicitly prohibited for the assignment across all semesters. Five pairs of students have been accused and convicted of academic dishonesty on this assignment. These pairs are identified with the letters A–E. The following list provides some context for the cheating pairs.

1. Pairs A–D contain submissions from two students from the same semester who collaborated.
2. Pair E contains submissions from two students who both copied code from Github, an online code sharing, and version management platform.

5.1.1 Results for Expected Cheating Pairs

Of the five expected pairs, Lichen ranked three within the top 10 most unlikely submission pairings; one as the 20th most unlikely pairing; and one as the 32nd most unlikely pairing. Submission pair ranking along with the expected number of submissions with this level of similarity and rarity are summarized in Table 4. The results are satisfactory for all expected pairs. For A–C, Lichen expects one in one trillion submission pairs in a population of this size to exhibit the same magnitude similarity and rarity. For D, Lichen expects one in 150,000 submission pairs in a population of this size to exhibit the same magnitude similarity and rarity. Finally for E, Lichen expected one in 11,000 submission pairs in a population of this size to exhibit the same magnitude similarity and rarity.

Expected Pairing	Expected number of submission pairs with this degree of similarity	Ranking
A	$<10^{-12}$	4
B	$<10^{-12}$	7
C	$<10^{-12}$	9
D	1.5×10^{-5}	20
E	1.1×10^{-4}	32

Table 4: Results for expected cheating pairs A–E.

5.1.2 Lichen Identifies New Cheating Pairs

Lichen has also identified some cheating pairs that were not in my advisor's set of expected cheating pairs. We give detailed examination to one particular pair, Pair F. Upon showing Lichen results about Pair F to my advisor, he looked at them and the code and was convinced the students had cheated within a matter of minutes. The most convincing pieces of evidence came from a unique comment and unique string literal inside an identical region of code.

This pair had gone unnoticed by my professor previously because Moss had ranked it as the 233rd most similar pair in the population. It determined that only 4–14% of code was common between these two submissions. My advisor said he would have been unlikely to look at the 233rd Moss result to check for cheating. He also noted that even if he had looked at this match from Moss, the highlighted similarities were uninteresting.

Lichen, on the other hand, ranked this pair as the 8th most unusual pair in the population. Lichen expects 1 in 1 trillion submission pairs to display their magnitude of similarity and rarity. A collection of interesting fragment collected for Pair F is shown Table 5. Observe that in addition to being quite similar, the fragments are also rare. The closest matching fragments in other submissions are dissimilar to the fragment pairs themselves. We note here that one of the students in this pair had a comment referencing a variable `pnew` even though their code had no variable with that name. Moss was unable to identify the similarity in Pair F because it does not take into account the similarity between string literals and comments.

Fragment 1	Fragment 2	Probability of Similarity and Rarity	Closest matching Fragment In Another Submission
For the id's that aren't allowed to contain numbers use this function	For the id's that aren't allowed to contain numbers use this function	3×10^{-8}	N/A[2]
pnew = pcurr + gamma*gradient(fcurr)	pnew = pcurr + gamma*gradient(f,pcurr)	8×10^{-6}	pnew = pcurr + gamma * gradient f of pcurr
This string contains a number. Invalid Id.\\n	This string contains a number. Invalid Id.\\n	2×10^{-8}	This function name is invaild\\n

Table 5: Pair F's most unusual comments and string literals.

5.1.3 False Positives

There were some false positives in the results of this case study. These false positives were easy to explain and provide motivation for improving Lichen string analysis. We have categorized three types of false positives and address how we plan to avoid these false positives in the future.

[2]There were no other comments in the submission population with an edit distance of less than 25 from this comment pair.

1. **Comments provided in the template code.** Professors may give some amount of template code and comments with every assignment. Students can either keep these provided fragments, or they can delete them. Several false positives were observed when students kept descriptive comments provided by the professor. This can be prevented in the future by feeding Lichen the template code and comments provided to students and disregarding fragments that match these templates in the similarity analysis.
2. **Similarities between strings in different contexts.** The context of a string literal or comment is something that Lichen does not currently take into consideration. For this reason, there were several false positives attributed to commonalities between strings that were used in completely different parts of the coding assignment. In the future, we hope to take the location of strings in the source file into account when performing a similarity analysis.
3. **Commented out print methods.** It is common practice for students to use print statements as a debugging method. Students may not delete these print statements and instead opt to comment them out. We noticed a couple false positives where the unusual comment pairs were commented out print statements. Printing functions are standard library functions—meaning that if a student wishes to print something, they have to use the same syntax. The similarities in these false positives could be attributed to the universal syntax for printing.

6 Future Work

Lichen is a work in progress. Many features still need to be developed for this tool to reach its full potential.

6.1 AST Similarity Analysis

There are two primary reasons why we are unable to report results including AST similarity analysis. Lichen was first tested on C/C++ assignments. The ASTs for these submissions were constructed using an ANTLR-provided C/C++ lexer and parser [6]. We have noticed significant flaws in the C++ ANTLR grammar. Of particular concern is the grammar's inability to distinguish type names from value names. This error leads to the incorrect parsing of various type names. One example is template types.

A template type name is a type of the format `template_type<...>`. The ellipse is filled with another type name. For example, `vector<list<string>>` is a template type `vector` with a nested template type `list` of type `string`. Consider the example in Figure 15. For the variable declaration `vector<list<string>> foo;` we would expect to see an AST fragment rooted at a variable declaration node with name `foo` and a subtree containing a the nested template type `vector<list<string>>`. Instead, we can see that ANTLR parses the variable declaration as an expression. In plain English, the ANTLR parse can be described as: `vector` is less than `list` is less than `string`; right shift `foo` by the result of the previous relational expression. After further investigation, it appears there is no satisfactory ANTLR C++ grammar readily available. We have identified several ways to fix this problem, however, they are all quite hacky. We have idea for an elegant solution that involves pre-parsing C++ to differentiate type names from other identifiers by iterating to a fixed point.

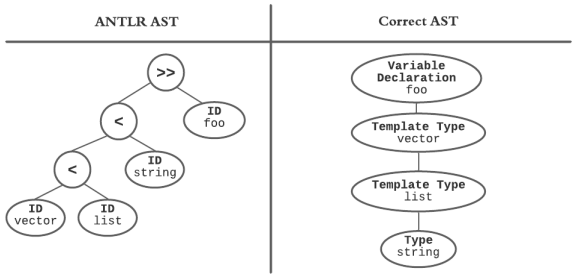

Figure 15: Incorrect and correct parses of `vector<list<string>>`.

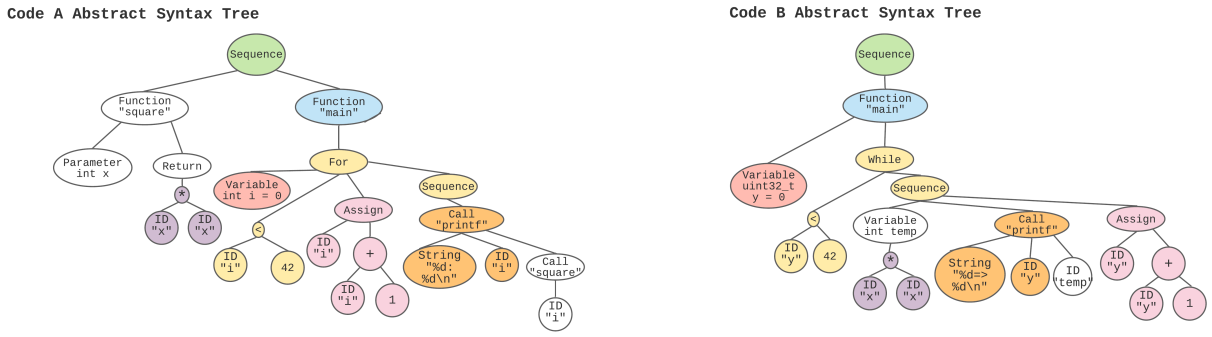

Figure 16: Reproduced ASTs for Code A and Code B from Figure 7 which color coded matching fragments. Algorithmically deciding this matching has proven to be a difficult task.

Another roadblock for AST similarity analysis is matching up two different ASTs in order to transform one into the other. This matching problem involves deciding when and where to apply a transformation on different trees. Consider again the ASTs for Code A and Code B from our Tree Distance Example in 7. These ASTs are reproduced in Figure 16 with their matching subtrees are color coded. Algorithmically determining this matching, which we call **correspondence mapping**, has proved to be quite challenging. For instance, nodes may correspond to things that are far away. Consider the case of the purple `*` subtree. In Code A, this subtree is in the the body of the `square` method; however in Code B, this subtree is in the initializing expression of a variable declaration in the body of a `while` loop. Another challenge is posed by the fact that subtrees that match may be completely different node types. For example, the rightmost child of the the orange subtree—which represents the call to `printf`—is a `Call` in Code A, but an `ID` in Code B. These problems are exacerbated further in a real program, where there are many instance of the same structures, such as loops. The reason that matching is a crucial step in tree similarity analysis is because in many cases of cheating, only parts of a program are copied and not others. We note that when the correspondence between trees is hand mapped, Lichen can perform transformations successfully.

Our most recent idea is inspired from decision tree classification algorithms [11]. At a high level, we plan to use the transformations Lichen knows of to determine which parts of a pair of ASTs can be transformed into one another. Lichen will then apply all possible transformation to the root node. Then repeat this process of identifying the transformations that can be applied and apply them all for all children. This continues recursively until the pair becomes equal. We would then select the minimum cost transformation sequence.

6.2 Semantic `grep`

`grep` is a command-line tool that allows a user to search for a text pattern in a file system. This tool can be used as a powerful tool to identify unusual strings in submissions. For example, if a student uses an unusual variable name such as `zebra`, a professor can search through all submissions for an assignment to see if any other students used a variable named `zebra`. This sort of search, however, is not as easy for structural peculiarities in code. Consider the pair of coding fragments taken from a real case of cheating in Figure 17. Given the context of the assignment, my advisor found the left-hand-side snippet to be an extremely peculiar approach to the assignment. He also noticed that another student had a similarly peculiar approach, captured in the right-hand-side code snippet. It appears as if small structural manipulations were made to conceal the fact that these two students approached this part of the assignment in the same way. It would be useful to have a

```
uint64_t lowerthanhalf =
    pop / 2 - pop / 200;
uint64_t higherthanhalf =
    pop / 2 + pop / 200;

if (pop % 2 == 1) {
    if (pop % 200 > 100) {
      higherthanhalf++;
    }
    else {
      lowerthanhalf++;
    }
}
```

```
halfpop = pop / 2;
pop_by_200 = pop / 200;

lower = halfpop - pop_by_200;
upper = halfpop + pop_by_200;
if (pop % 2 == 1) {
    if (pop % 200 < 100) {
          lower++;
    } else {
          upper++;
    }
}
```

Figure 17: Structurally similar code that is the target of semantic **grep**. Standard **grep** cannot search for similarities like this.

semantic `grep` tool that allowed professors to quickly search a body of code submission for similar AST fragments. Such a tool would have allowed my advisor to quickly identify that these were the only two students who approached this particular part of the assignment in this unusual way.

7 Conclusion

Lichen is a new cheating detection tool that aims to decrease the amount of time professors spend on identifying and reporting cheating on programming assignments. Lichen's approach to cheating detection is different from today's most-used cheating detection tool, Moss. Unlike Moss, Lichen's similarity analysis accounts for submission components such as comments, function names, variable names, and string literals. Lichen also provides greater insight into the flagged similarities between submissions. My advisor is excited about the preliminary results of Lichen. We look forward to continuing to work on Lichen in order to submit a paper to an educational conference in the future. In the long term, we aim to open source Lichen so that educators everywhere can use it.

References

[1] Duke University. *2019-2020 Duke Community Standard in Practice*. 2020.

[2] 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, page 76–85, New York, NY, USA, 2003. Association for Computing Machinery.

[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*, volume 3, pages 13B3/18–13B3/22 vol.3, 1999.

[4] Christian Collberg and Ginger Myles. Cheating cheating detectors. 04 2004.

[5] Gen Chang. How-to-cheat-in-computer-science-101, 2015.

[6] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

[7] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1997.

[8] V. I. Levenshtein. Binary codes capable of correcting spurious insertions and deletions of ones. 1965.

[9] William W. Cohen, Pradeep Ravikumar, and Stephen E. Fienberg. A comparison of string distance metrics for name-matching tasks. In *Proceedings of the 2003 International Conference on Information Integration on the Web*, IIWEB'03, page 73–78. AAAI Press, 2003.

[10] CPP Reference. Fundamental types, Oct 2020.

[11] Pang-Ning Tan, Michael Steinbach, Anuj Karpatne, and Vipin Kumar. *Classification: Basic Concepts and Techniques*, page 113–192. Pearson Education, Inc., 2 edition, 2019.