Internal Report 2013–21 August 2013

Universiteit Leiden
Opleiding Informatica

Plagiarism Detection

in

C++ Programs

Boyd Witte

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Plagiarism Detection in C++ Programs

Boyd Witte

August 29, 2013

Abstract

For certain courses in the bachelor program, students have to write C++ programs to solve problems. Unfortunately it sometimes happens that students copy code from their fellow classmates, which is not allowed. Currently Moss2 is used to detect this plagiarism, but it does not find all plagiarized files. Therefore we experiment with a few new methods to improve upon Moss2. These methods are based upon transforming a file into a numerical array, which can be compared to other array's of other files. Based upon these array's, some of the new methods can express a distance between two files that could indicate plagiarism. The other methods can indicate what parts of these files might be plagiarized. To nullify certain plagiarism techniques, such as variable renaming, a preprocessing option is included that is able to edit files before they are divided into the numerical array's.

1 Introduction

This project is about plagiarism between C++ code files. This research is done for a bachelor project at the Leiden Institute of Advanced Computer Science (LIACS) at the Leiden University, under supervision of Walter Kosters and Jeannette de Graaf.
The result of this project will be a program that can compare a set of up to 200 C++ files, and show which files might contain plagiarized content. The program will have a simple interface, usable from the command prompt. The project takes place at the LIACS where in the first year of the bachelor programs Mathematics, Computer Science, Astronomy, and Physics, every student follows a course to learn C++ programming. This course [6] includes several practical assignments, which students can do in pairs of two. Students are allowed to discuss the assignment, but they may not copy code from other groups. This, however, still tends to happen. That is why all the files are checked whether they contain code from other files. This is currently done by using Moss2 [4], which is fast in comparing a large amount of files and is capable of processing several different languages including C, C++, C# and Javascript, but it is not 100% accurate. With we focus only on C++, and by doing so reach the hoped for improvement upon Moss2
In Figure 1 we show the general flow of our program. This figure also shows a method called "Suspicious File Indication", which is not used by us, but proposed in Section 5.1.1. In Section 2 an explanation of plagiarism and the used plagiarism detection methods will be given. In Section 3 results will be discussed. In Section 4 related work and other plagiarism methods will be mentioned and discussed briefly. In Section 5 a conclusion will be made. In Section 5.1 future work will be suggested.

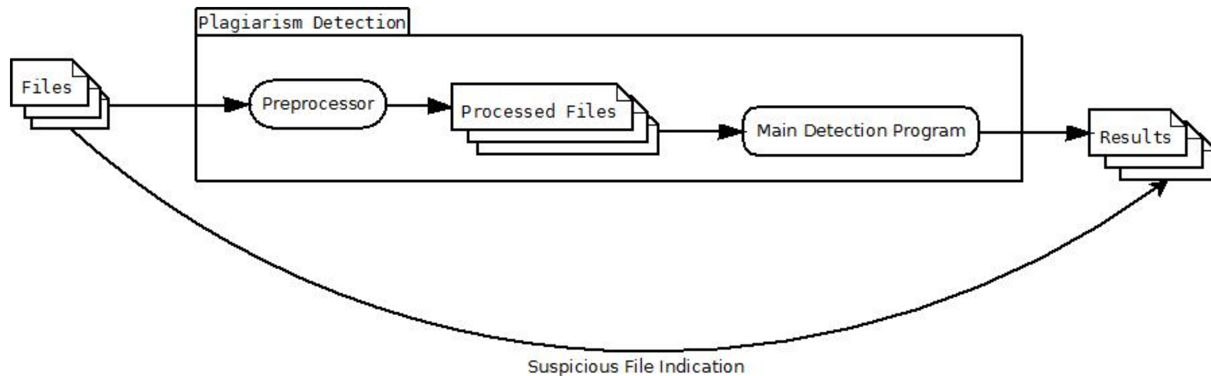

Figure 1: Plagiarism detection

2 Detection

First we will have to define plagiarism. This includes discussing the different degrees of plagiarism and the problems associated with defining plagiarism. Second we will introduce k-grams, which are involved in comparing files, followed by explaining the methods we used for determining the distance between two files.

2.1 When is it plagiarism?

According to the Van Dale dictionary [2], plagiarism is the act of copying intellectual work of others and making it public as your own. According to [5], plagiarism in source code can be divided into seven levels, each level indicating an increasing amount of modification. For simplicity, we narrow this down to three levels:

1. Copying: Directly copying a part of or the entire source code. Leaving everything as it was. This would be the easiest to detect, because you know exactly what you are looking for. However, there is a danger in calling code plagiarized when it is an exact match. Small pieces of code can often look the same. A fine example of this is a for-loop. In the course of Programming Methods [6], a for-loop header is shown looking like this:

   ```
   for ( i = 0; i <= n; i++ )
   ```

   It might very well be that the majority of the students writes the for-loop exactly like that. This should not be considered plagiarism, but common knowledge.

2. Editing and copying: Copying parts of the source code and editing it. Editing could be anything, e.g. removing comments or changing every variable and function name. This is harder to detect, since you do not know what you are looking for. The names of the variables could be anything, and the order of functions could be changed, leaving little resemblance to the original. You would have to use a comparison method that looks at more than exact matches.

3. Based on someone else's idea: Not copying, but using the exact same method as someone else. This is the hardest to detect, since the code is written by different people and can look very different. This case also brings us into a gray area of

plagiarism in source code. When detecting plagiarism in source code written for an assignment, you are bound to encounter source code with the same functions and goals. Every student is asked to solve the same problem, and they all get the same information. Chances that two or more students try the same technique would be large, especially if you take into account that new students might have access to source code or ideas from students who have already done the course.
It can also be argued that this form of plagiarism should not be detected. For if a student manages to program the code with ideas someone else gave him, did he not succeed in finishing the assignment?

2.2 k-Grams

As explained in [9], a k-gram is a substring of the text in a file, where k stands for the length of this string. Dividing a file into k-grams can be used to create a fingerprint for this file. When dividing a file into k-grams, we count how many times a k-gram appears inside a file. This includes whitespace and newlines. As an example: The string "ab␣cdef" (we use ␣ to denote a space) with $k = 3$, will divide into the following 3-grams:

```
ab␣
b␣c
␣cd
cde
def
efg
```

Dividing the entire file into these k-grams results in a large list of how many times each k-gram appears. This list can help detect copied and edited code. When two pieces of code contain the exact same function for example, this function will result in the same k-grams. Different files that result in partially the same or very similar k-gram lists could indicate plagiarism. When many instances of k-gram frequencies match (almost) exactly, it is likely these k-grams came from the exact same lines of code. When a line is changed, or moved, the k-grams also change, so this method should work well for pieces of code that are exactly the same.

However, because k-grams change so quickly when code is changed, it might be less suited for detecting copied code that was edited. To solve this, we use inverse-document-frequency. This method indicates the importance of a certain k-gram within a given set of files. With this we would tell that certain matches between files are more important than other matches. More about this is explained in Section 2.3.6.

2.3 Distance Methods

In this section we will discuss all the methods of comparison we have used to detect plagiarism. In an attempt to overcome certain problems or improve on our comparison methods, we will use a form of preprocessing to alter the files as the user desires. We have chosen to do this preprocessing in Perl. This Perl-preprocessor transforms all files into new files that can be read by our plagiarism detection program.

2.3.1 Preprocessing

As mentioned in Section 2.3.2, detection can improve if the length of the k-grams is larger than the length of common idioms. This can be achieved by decreasing the length of idioms instead of increasing the length of k-grams. The Perl-preprocessor can process all the files before having the main-program look at them. Preprocessing includes replacing words with symbols, removing spaces and tabs and changing comments. The Perl-preprocessor has the following switches:

- `-C`: Comments will be removed. The user can choose to use `-c` instead, and replace all comments with a special unique character indicating a comment.
- `-W`: Basic axioms will be changed to a special unique capital character indicating what axiom it is. This also removes all capitals from a file.
- `-IO`: `cin`, `cout` and strings will be changed to special capital characters indicating either `cin`, `cout` or a string. This also removes all capitals from a file, and `endl` will be removed.
- `-V`: Each word will be changed to a special unique capital character indicating it was a word. If common idioms and comments are not replaced yet, this function will replace them. The user can choose to use `-v` instead, and replace each word with a special character followed by a value. When a word is encountered that has already been replaced by, for example, `X2`, it will again be replaced by `X2`.
- `-S`: Remove all spaces.
- `-N`: Remove newlines.
- `-Sym`: Remove the symbols: `: ; , # < >`.
- `-Pre`: Enables all of the above mentioned switches.
- `-susp x`: Set the threshold for suspicious k-grams to `x` (see Section 2.3.2).

Besides these switches, the script will always replace a tab with tree spaces. Figure 2 shows an example of a file preprocessed with the `-Pre` switch. Note that newline characters are actually removed during preprocessing, but are shown in this example for simplicity. Arguments can be combined to suit the user's needs. A special case when combining these arguments is with `-Pre`. Using `-Pre -N`, for example, will enable all preprocessing switch, except for `-N`, but using `-N -Pre` will do the same as `-Pre`. Preprocessing will be executed (or skipped when the switch is not given) in the following order:

1. Remove all tabs.
2. Replace capitals with lowercase characters.
3. Remove comments indicated with `//`.
4. Replace common idioms with unique symbols.
5. Replace `cin`, `cout`, strings, and `endl`.

6. Replace numbers with a unique special capital character and replace words with a special unique character.
7. Remove spaces and line feeds.
8. Remove newlines.
9. Remove symbols.
10. Remove comments indicated with `/*...*/`.

This order makes sure that when the user wants to replace both words and common idioms, the common idioms are not replaced by the special unique character for a word before they can be replaced by a special unique character for that idiom. Since all tabs are always replaced by spaces, and this is done before all spaces are removed, using the `-S` switch will remove all tabs. Not removing comments but using an switch like `-V`, for example, will replace all words in the comments with the character that `-V` assigns to it. After the arguments have been handled and preprocessing has been done, the script will call the main program for plagiarism detection. When the program is done, the script will continue and remove all the preprocessed files.

Example code

```
1  while (!found) {
2          i++;
3          if (a[i] == b[i]) {
4                  found = true;
5          }
6  }
7  cout << "Found is true" << endl;
8  i = 0;
9  while (!found) {
10         i++;
11         if (a[i] == b[i]) {
12                 found = true;
13         }
14 }
```

Preprocessed example code

```
1  W(!X){
2  X++
3  I(X[X]==X[X]){
4  X=X
5  }
6  }
7  AQE
8  X=G
9  W(!X){
10 X++
11 I(X[X]==X[X]){
12 X=X
13 }
14 }
```

Figure 2: Preprocessing example

2.3.2 Storing the data

We have chosen to work with 3-grams. We want to store the entire list of k-grams for each file in the program, so the length of each k-gram has to be small. Each k-gram will be represented in a 3D array. For simplicity we haven chosen to limit the possible characters to an alphabet of 97 characters, instead of the entire ASCII table. The following 97 characters have been used:

```
a, b, c,..., z
A, B, C,..., Z
0, 1, 2,..., 9
\n  \r  ␣  !  "  #  $  %  &  '  (  )  *  +  ,  -  .
/  :  ;  <  =  >  ?  @  [  \  ]  ^  _  `  {  |  }  ~
```

To simplify matters more, and to ensure we don't run into memory problems on a computer, we limit the amount of documents to a maximum of 200. This results in an $200 \times 97 \times 97 \times 97$ integer sized array (roughly 700 MB) to store the k-grams for each file. This is a simple space-consuming method of counting k-grams, and expanding the length of k by 1 generates a new dimension of size 97 for each of the files. It is however desirable to increase k to 4 or even 5. In [9] the authors mention that to reduce noise (uninteresting matches) the length of k has to be greater than common idioms of the language. With $k = 3$, we met this criteria for the word 'if' for example, but 'for', 'int', 'while', 'char', 'bool' and 'cout' are still recognized as one or more k-grams, resulting in a lot of noise because these words have a high chance to be used by many students, as explained in Section 2.1.

When all files are known and each file has been transformed into an array of k-grams as mentioned above, we can begin comparing the files. For our examples we assume we have two files that are divided into 2-grams. For simplicity we assume only the characters `A`, `B` and `C` are relevant. The files are stored in an array as shown in Figure 3. It should be noted that in every comparison method, all files are compared twice. A comparison is made between file f_1 and file f_2, and between file f_2 and file f_1. This is done because file f_1 might be far larger than file f_2, and therefore the percentage that is plagiarized in file f_1 might be smaller than the percentage of plagiarized content in file f_2.

In the sequel we let $freq(t, f)$ be the frequency of k-gram t in document f, and K be the set of all k-grams. Note that $|K| = 97^k$, when using an alphabet of 97 characters.

2.3.3 Finding exact k-gram frequency matches

When two files have an exact match in the frequency of the same k-gram, this frequency is added to the amount of matches these files have.

Example: If file f_1 lists 2-gram `AA` 2 times and `BB` 2 times, and file f_2 lists `AA` 2 times and `BB` 10 times, then the result will show that f_1 and f_2 have 2 matches in one location. Figure 3 shows all matches between f_1 and f_2. In this example, f_1 and f_2 have a total of

(a) 2-grams of f_1

	A	B	C
A	2	0	3
B	0	2	0
C	4	1	0

(b) 2-grams of f_2

	A	B	C
A	2	0	3
B	1	10	0
C	1	1	0

Figure 3: Exact matches between f_1 and f_2

$2 + 1 + 3 = 6$ matches. This is expressed by the following formula:

$$exact_matches(f_1, f_2) = \sum_{t \in K} \delta(freq(t, f_1), freq(t, f_2)) \cdot freq(t, f_1)$$

This example has three different matches. We express the total amount of different matches as follows:

$$total_exact(f_1, f_2) = |\{t \in K \mid freq(t, f_1) = freq(t, f_2)\}|$$

Here $\delta(x, y)$ is 1 when $x = y$, and 0 otherwise.

2.3.4 Finding almost exact k-gram frequency matches

When two files have an almost identical frequency for the same k-gram, the smallest of the two frequencies is counted as the amount of matches these files have. To be "almost identical" the frequencies may not have a difference bigger then a certain threshold.
Example: If file f_1 lists 2-gram `CA` 4 times and `BB` 2 times, and file f_2 lists `CA` 1 time and `BB` 10 times, then the result will show that f_1 and f_2 have 1 ($\min(1, 4)$) match in one location, plus the matches found by the Exact method. The difference between 10 and 2 is too large, thus not counted as a match. Figure 4 shows all matches found between file f_1 and f_2. In this example, f_1 and f_2 have a total of $2 + 1 + 3 + \min(1, 4) = 7$ matches. The number of matches is expressed by the following formula:

(a) 2-grams of f_1

	A	B	C
A	2	0	3
B	0	2	0
C	4	1	0

(b) 2-grams of f_2

	A	B	C
A	2	0	3
B	1	10	0
C	1	1	0

Figure 4: Almost matches between f_1 and f_2

$$almost_exact_matches(f_1, f_2) = \sum_{t \in K} almost(freq(t, f_1), freq(t, f_2))$$

Here $almost(x, y)$ is defined as follows:

$$almost(x, y) = \begin{cases} \min(x, y) & \text{if } |x - y| \leq \dfrac{x + y}{20} \\ 0 & \text{otherwise} \end{cases}$$

The threshold is set to 10% of the average of x and y. Note that the threshold in our example is not the same threshold as defined in $almost(x, y)$. For the example we used an arbitrary threshold for simplicity.
We express the total amount of different matches as follows:

$$total_almost(f_1, f_2) = |\{t \in K \mid almost(freq(t, f_1), freq(t, f_2)) \neq 0\}|$$

2.3.5 Finding each k-gram match

When two files have at least 1 occurrence (*frequency* > 0) of the same k-gram, the smallest of the two frequencies is counted as the amount of matches these two files have.
Example: If file f_1 lists 2-gram `BB` 2 times and file f_2 lists `BB` 10 times, then the result will show that f_1 and f_2 have 2 ($\min(2, 10)$) matches in one location, plus the matches found by both the Exact method and the Almost method. Figure 5 shows all matches between f_1 and f_2. In this example, f_1 and f_2 have a total of $2+1+3+\min(1,4)+\min(2,10) = 9$ matches. This is expressed by the following formula:

(a) 2-grams of f_1

	A	B	C
A	2	0	3
B	0	2	0
C	4	1	0

(b) 2-grams of f_2

	A	B	C
A	2	0	3
B	1	10	0
C	1	1	0

Figure 5: All matches between f_1 and f_2

$$\mathit{all_matches}(f_1, f_2) = \sum_{t \in K} \min(\mathit{freq}(t, f_1), \mathit{freq}(t, f_2))$$

We express the total amount of different matches as follows:

$$\mathit{total_all}(f_1, f_2) = |\{t \in K \mid \min(\mathit{freq}(t, f_1), \mathit{freq}(t, f_2)) \neq 0\}|$$

2.3.6 tf-idf

To reduce noise, and search for important matches, a method must be used that calculates the importance of a k-gram in a document. This method is tf-idf [3]. Tf-idf stands for "term frequency-inverse document frequency", and can determine how important a string is within a set of documents.
As the name suggest, the tf-idf contains two parts.

1. The Term Frequency indicates how important a string is within a document. It is calculated based on the string frequency in that document and the frequency of the string with the highest frequency in that document. This is done with the following formula:

$$\text{Term Frequency } \mathit{tf}(t, f) = 0.5 + \frac{0.5 \cdot \mathit{freq}(t, f)}{\max_{w \in K}(\mathit{freq}(w, f))}$$

   Here f is a document, t is a string, $\mathit{freq}(t, f)$ is the frequency of string t in document f, and $\max_{w \in K}(\mathit{freq}(w, f))$ is the frequency of the string, which has the highest frequency in document f.

2. The Inverse Document Frequency is an indicator of how common a string is within the set of documents. The Inverse Document Frequency is calculated based on the size of the set of your documents, and the size of the set of documents that contain the string. This is done with the following formula:

$$\text{Inverse Document Frequency } idf(t, F) = \log \frac{|F|}{dfreq(t)}$$

Here F is the set of all documents, $|F|$ is the number of documents in this set, and $dfreq(t)$ is the number of documents containing string t at least once.

Together they form the tf-idf:

$$tf\text{-}idf(t, f, F) = tf(t, f) \cdot idf(t, F)$$

This value gives an indication of the importance of string t in document f within the total set of documents F. When we use this value to calculate the distance between two files, we get the following formulas:

$$weighted_exact_matches(f_1, f_2) = \sum_{t \in K} tf\text{-}idf(t, f_1) \cdot \delta(freq(t, f_1), freq(t, f_2)) \cdot freq(t, f_1)$$

$$weighted_almost_exact_matches(f_1, f_2) = \sum_{t \in K} tf\text{-}idf(t, f_1) \cdot almost(freq(t, f_1), freq(t, f_2))$$

$$weighted_all_matches(f_1, f_2) = \sum_{t \in K} tf\text{-}idf(t, f_1) \cdot \min(freq(t, f_1), freq(t, f_2))$$

2.3.7 Marking suspicious k-gram matches

A form of comparing files that is somewhat similar to the tf-idf method from Section 2.3.6 is the following. The method looks only at the document frequency $dfreq(t)$ of a k-gram: The number of documents containing that k-gram at least once. If the document frequency is lower than a given threshold θ, this is marked for both documents. When the threshold is set to 3, a suspicious match between two files means that only these files contain a certain k-gram. The amount of suspicious matches is counted as follows:

$$suspicious_matches(f_1, f_2) = \sum_{t \in K} suspicious(freq(t, d1), freq(t, f_2), dfreq(t))$$

Here $suspicious(i, j, k)$ is defined as

$$suspicious(i, j, k) = \begin{cases} 1 & \text{if } i > 0 \text{ and } j > 0 \text{ and } k < \theta \\ 0 & \text{otherwise} \end{cases}$$

The threshold has a default value of 3, but the user can set the value by using the `-susp x` argument in the Perl-preprocessor, where `x` is the desired threshold.

2.3.8 Expanding suspicious matches

The method makes "suspicious matches" more suspicious. Starting from a k-gram that appears in both file f_1 and f_2 and that is marked as suspicious, we will try to expand that string of k letters to a longer string. We expand 3-gram `abc` with the letter `d` if the 3-gram `bcd` appears on the same line as `abc`. We then continue to search for a 3-gram starting with `cd`, on that same line. If, for example, the 3-gram `cde` would also appear on the same line as `abc` and `bcd`, we would construct the string `abcde`. We can continue to do this until all k-grams on that line are used, or no more k-grams matches it's first $k-1$ characters with the last $k-1$ characters of the currently constructed string. This method

will chain together as many k-grams as possible. Because we know nothing about the position of a k-gram within the line itself, we do not actually know if the created string appears on that line. Therefore we write the string to a file, and use a post-processor to filter out strings that are not found in the file. This post-processor is the same program as the Perl-preprocessor.

2.3.9 Finding multiple lines of code that match

Much like the method we use to expand suspicious matches, we start from a k-gram that appears in both the file f_1 and f_2 and that is marked as suspicious. We check on what line this k-gram appeared in f_1 and f_2, and than take a look at the previous line in f_1. If this line contains a k-gram that is also present in the previous line in f_2, the line before the previous line in f_2 or the line after the previous line in f_2, we note that these lines have a match. When a match is found, we continue again to the previous line in f_1 and f_2. And repeat the comparison of this line in f_1 and the 3 lines in f_2. To clarify we give the following example:
Consider the two pieces of code in Figure 6. Both pieces of code are very similar in

Code from f_1

```
1 int teller = 0;
2 char kar = invoer_file.get();
3 if (kar == 'a') {
4   teller++;
5   for (int i=0; i<teller; i++){
6     cout << "a";
7   }
8   cout << endl;
9 }
```

Code from f_2

```
1  int count;
2  char car;
3  bool found;
4  found = false;
5  car = input.get();
6  count = 0;
7  if (car == 'a') {
8    count++;
9    for (int j=0; j<count; j++) {
10     cout << "a" << flush;
11     found = true;
12   }
13
14   if (!found) {
15     cout << "We found nothing" << endl;
16   }
17 }
```

Figure 6: Two similar pieces of code

functionality and structure. The only difference is the names of the variables and the line numbers. Lets assume that the 3-gram `"a"` is marked as suspicious. This 3-gram appears on line 6 in file f_1 and on line 10 in file f_2. We begin by marking both theses lines as similar for these two files. We then continue as follows:

1. Compare line 5 in f_1 with line 8, 9 and 10 in f_2.
   Line 5 in f_1 contains at least one 3-gram equal to line 9 in f_2.

2. Compare line 4 in f_1 with line 7, 8 and 9 in f_2.
   Line 4 in f_1 contains at least one 3-gram equal to line 8 in f_2.

3. Compare line 3 in f_1 with line 6, 7 and 8 in f_2.
   Line 3 in f_1 contains at least one 3-gram equal to line 7 in f_2.

4. Compare line 2 in f_1 with line 5, 6 and 7 in f_2.
   Line 2 in f_1 contains at least one 3-gram equal to line 5 in f_2.

5. Compare line 1 in f_1 with line 4, 5 and 6 in f_2.
   Line 1 in f_1 contains at least one 3-gram equal to line 6 in f_2.

6. No more previous lines in f_1.
   Return to the original line 6 in f_1 and line 10 in f_2 and start looking at following lines.

7. Compare line 7 in f_1 with line 10, 11 and 12 in f_2.
   Line 7 in f_1 contains at least one 3-gram equal to line 12 in f_2.

8. Compare line 8 in f_1 with line 11, 12 and 13 in f_2.
   Line 8 in f_1 contains no 3-gram equal to any 3-gram in line 11, 12 or 13 in f_2.

9. Conclusion File f_1, line 1–7, and file f_2, line 5–12, are similar.

2.3.10 Calculating distance

Because each pair of files is compared twice, we will express the distance between two files as follows:

$$total_grams(f_1, f_2) = total(f_1) + total(f_2)$$

$$match_count(f_1, f_2) = matches(f_1, f_2) + matches(f_2, f_1)$$

$$distance(f_1, f_2) = \frac{match_count(f_1, f_2)}{total_grams(f_1, f_2)} \cdot 100$$

Here $matches(x, y)$ can be either $exact_matches(x, y)$, $almost_matches(x, y)$ or $all_matches(x, y)$ and $total(x)$ refers to the number of k-grams in file x.

3 Experiments

We use the proposed methods to experiment with. For the experiments we use a test set of 108 files. These files are gathered over several years, and are all made for the same assignment. The assignment had the students write a program to do run-length encoding. We divide our test set into four cases, so we can later indicate how well each method has done in finding plagiarism. These cases are:

- Case A: 2 files that are exactly identical or nearly identical, except for a few minor changes. For example: When a student submits source code that does not compile, and is asked to fix this. The fixed source code is nearly identical, except for a few lines that contained errors and that have been changed. Our test set contains one example of this case.

- Case B: 2 files which are extremely likely plagiarized. Many function structures and variable uses are the same. Many functions have the exact same purpose. These files are not completely identical, but many functions are highly similar with as only difference the names of the variables. It is highly unlikely that different people came up with the exact same assignments and structure for a function, and thus plagiarism is very likely.
  In this case we also include a change in the order of operations. The functionality and variable names are highly similar, but the order of operations has changed slightly. This can vary from switching two lines, or switching the order of entire while–loops, for example. Of the known plagiarized files in out test set, this is the largest group. Presumably because this form of plagiarizing takes the least effort, besides blatantly copying, and is harder to detect than an (partially) identical copy.

- Case C: 2 files which are less likely plagiarized, but still show suspicious similarities. These files have a lot of differences in variable names, function names, and order of declaring and placing functions. However, the order of operations, and the control logic of both codes is very similar. Apart from a different order of variable declaration and two if–statements switched around here and there, the order of operations in many functions is identical and many functions have the same purpose. However it could be that these similarities came from the writers discussing ideas in an attempt to solve a problem together, and should thus not count as plagiarism.

- Case D: 2 files which are in no way plagiarized. These files are submissions for the same assignment so similarities might be found, but based on the reputation of the people who wrote the code, the level of programming skill expressed in the files, or the difference in the chosen method to solve the given assignment, we can conclude that the 2 files are not plagiarizing from each other.

The results will be categorized in these cases based on comparing the file pairs in the results ourself manually.
All results will be shown in a table that compares them to the results of Moss2. The first column shows the top 15 ranking of Moss2. The following columns show the ranking for those same files, but according to the respective method of comparison and the preprocessing switches used for those results. All methods will be tested without any preprocessing switches, and with all preprocessing switches. Each method will also be tested with three other preprocessing varieties. Moss2 is not 100% guaranteed to find all plagiarism, and is also not noise free. Therefor it is not guaranteed that the top 15 of Moss2 contains only plagiarized content, or that these results contains all plagiarized files in the test set. We have limited the Moss2 results to the top 15, instead of the total 250 produced results because this appeared to be the border between results containing mostly correct results against results containing mostly noise, based on manually comparing the files ourself.
The results should be interpreted as follows: The first row indicates that Moss2 put a certain pair of files on rank 1. If the first cell of "No Preprocessing" contains the number 8, this means that the pair that Moss2 ranked first, is ranked 8th by "No Preprocessing". A $*$ means that the pair of files found by Moss2 is not found within the top 30 of the respective method.
All results are gathered from tests done upon the same set of files, except for the results discussed in Section 3.9, which are gathered from a new test set.
Due to privacy reasons, the files from the test set will not be made public.

3.1 Exact matches

We have chosen to do the following extra tests with our exact matching method:

1. `-C -S`. Since we only look at exactly the same k-gram frequencies, we suspect that comments are to easily reworded into something with a similar meaning, but with entirely different words (and thus k-grams). Spaces are also removed. Since spaces are used a lot, and one can easily enter an extra space or two, even by accidentally hitting the space-bar, we think it's better to remove them. With this we are only left with the actual code.

2. `-Pre -Sym`. We tried to improve upon the `-Pre` switch which seemed to have the best results for this method. We thought that by leaving in the symbols, we could do more precise comparing because this would mean that the code `A < B` and `A > B` would no longer be seen as the same.

3. `-Pre -Sym -v`. Used as an improvement upon `-Pre -Sym`. Manual observation of code led us to believe that variables are often declared in the same order. In theory using `-v` should thus result in many the same k-grams for each operation that is also applied in a similar function in another file.

The results can be seen in Figure 7. Every method was able to find case A, the pair of

MOSS2	No preprocessing	`-C -S`	`-Pre -Sym`	`-Pre -Sym -v`	`-Pre`
1	15	6	1	4	2
2	1	1	7	3	5
3	*	15	3	5	3
4	*	*	*	*	*
5	*	2	5	1	4
6	*	*	2	*	1
7	*	22	*	*	25
8	*	19	*	*	*
9	*	*	*	*	*
10	*	*	*	*	*
11	*	*	*	*	*
12	*	*	*	*	*
13	*	*	*	*	*
14	*	*	*	*	*
15	*	*	*	*	*

Figure 7: Exact matches result

files that are nearly identical. `-Pre` did a decent job finding the rest in the top 7. `-Pre -Sym -v` had a false positive on rank 2. This is odd, considering that rank 1, 3, 4, and 5 were all correctly classified as plagiarized. The pair of files on this rank had in common that they both used the same variable names, and were both relatively simply written.
We have two things to note. The first is the lack of pair 4. None of the methods is able to find pair number 4, found by MOSS2. This pair, which would be classified as a case B, did significant renaming of variables. Where file f_1 would contain a variable name called

"test3", f_2 would contain "testtesttest", for example. This easily prevents exact k-gram matches to be made.
The second interesting pair is a result of `-C -S` on rank 7 (or ranked 10 by "No preprocessing"). This pair is not recognized by MOSS2, but could be classified as a case B. Most smaller functions and the main function are almost identical.
All other results given in the top 15 by this method are false positives.

3.2 Almost matches

This method is almost identical to the exact matches, so we have used the same preprocessing switches to test this method. The results can be seen in Figure 8. As one might

MOSS2	No preprocessing	`-C -S`	`-Pre -Sym`	`-Pre -Sym -v`	`-Pre`
1	*	10	1	5	1
2	1	1	7	4	4
3	*	3	2	2	2
4	*	*	*	*	*
5	*	24	14	1	5
6	*	22	3	8	3
7	*	*	*	*	*
8	*	*	*	*	*
9	*	*	*	*	*
10	*	*	*	*	*
11	*	*	*	*	*
12	*	*	*	*	*
13	*	*	*	*	*
14	*	*	*	*	*
15	*	*	*	*	*

Figure 8: Almost matches result

expect because of the results in Section 3.1, pair 4 from MOSS2 is again not recognized by any of the preprocessing varieties and pair 1 (case A) is found by all of them. We did notice one interesting case, while using `-Pre -Sym`, pair number 15, that was ranked 17 by MOSS2, is likely to be plagiarized. It is a case C, where it is unlikely that the similar code is coincidence.
`-Pre -Sym -v` has a false positive on rank 3. This is the same false positive that was ranked 2nd by our exact comparison method while using `-Pre -Sym -v`.

3.3 All matches

This method should not be influenced by small differences in names or comments. It can however be influenced by larger frequencies such as spaces. Since every tab is replaced by 3 spaces, we suspect that all files contain large quantities of the 3-gram ␣ ␣ ␣. This could lead to a wrong distance between two files, because the distance is based too much upon these 3-grams of spaces. Therefore we test with the `-S` preprocessing switch, and `-C -S` as an improvement upon it. For the same reason as in 3.1 we expect that the `-C`

Moss2	No preprocessing	`-C -S`	`-S`	`-Pre -Sym -v`	`-Pre`
1	7	10	3	6	1
2	1	1	1	2	5
3	*	2	9	1	2
4	11	9	14	5	6
5	6	4	2	4	*
6	*	*	*	*	3
7	*	*	*	*	*
8	*	*	*	*	*
9	*	*	*	*	*
10	*	*	*	*	*
11	*	*	*	*	*
12	*	*	*	*	*
13	*	*	*	*	*
14	*	*	*	*	*
15	*	*	*	*	*

Figure 9: All matches result

switches improves the result. The results can be seen in Figure 9 Almost all preprocessing varieties do a decent job at finding the top 5 of Moss2. Remarkable is that `-Pre -Sym -v` and `-Pre` score so well. By using all preprocessing switches, all files are reduced to a certain set of special characters. This would mean that all files contain many of the same k-grams, and thus we would suspect that we have a lot of noise (for the same reason we removed all spaces).
`-Pre` had a remarkable pair of files on rank 4. This pair of files is the same pair of files found by our exact method on rank 7 while using `-C -S` or on rank 10 with no preprocessing. As stated before, we suspect that this pair of files contains plagiarized content from each other.

3.4 Weighted matches

We applied tf-idf to the exact, almost, and all comparison methods. We will discuss the results per method below.

3.4.1 Weighted exact

We first did the default no preprocessing and `-Pre` preprocessing switch. The last gave a very good result, so we tried to improve upon that with `-Pre -Sym -v`. With the other two tests we hope to see a difference between removing comments or replacing it with a special character. The results can be seen in Figure 10. Almost every variety of preprocessing did a good job finding the top 3.
We found one interesting case when we used no preprocessing. The results did not appear to be good, but we found a one pair of files that might be of interest. The 3th pair could not be considered plagiarized code, but it did contain a rather large, nearly exactly copied, piece of code that outputted some information to the screen. The information the code showed is not that interesting, it contains the names of the writers and information about

Moss2	No preprocessing	`-Pre -C -v`	`-Pre -v -c`	`-Pre -Sym -v`	`-Pre`
1	8	2	2	2	2
2	1	1	1	1	1
3	*	3	3	3	3
4	*	*	26	14	21
5	*	6	8	9	22
6	*	*	12	13	4
7	*	*	*	*	20
8	*	4	30	*	12
9	*	10	6	5	*
10	*	*	*	*	*
11	*	*	*	*	*
12	*	*	*	*	*
13	*	*	*	*	*
14	*	*	*	*	*
15	*	*	*	*	*

Figure 10: Weighted exact matches results

the assignment, but the way it was written is very noticeable. Both files had exactly the same layout of information. This led us to believe that this piece of code was copied.
We tried finding anything positive between the high ranked pairs of our results that Moss2 did not rank. We could however not find anything, and thus we should conclude that these results are false positives. The most interesting false positive is found by `-Pre -C -v`, which found a pair of files that is unlikely plagiarized, but both did contain one small function that was completely identical apart from variable names. But even variable names had a lot in common.

3.4.2 Weighted almost

As the results from `-Pre` show promising, we tried to expand on that again. The results can be seen in Figure 11. The first thing we noticed was a the pair of files on rank 4 of `-Pre`. This is again the same pair of files we found before in Section 3.1 and Section 3.3. Rank 4 in `-Pre -Sym -v` is this same pair of files. `-Pre -Sym -v` rank 7 and `-Pre` rank 6 are also the same pair of files. These files both contain the same rather large function and have written it very identical. There are however a few distinct differences that led us to believe that it is not plagiarized, but likely based upon the same idea students discussed with each other. On rank 6 of `-Pre` is also a previously found pair of files. These files were however not plagiarized but did contain one small function that was identical. On rank 7 of No preprocessing, we find the same pair of files with identical output we found in Weighted Exact without preprocessing.

3.4.3 Weighted all

These results were very poor. The best results came from `-Pre -Sym -v`, which found the top 3 of Moss2 but most other results seemed random with no indication of plagiarism. Case A was the only pair of files that got a decent ranking, regardless of preprocessing

Moss2	No preprocessing	-Pre -v	-Pre -Sym -v -c	-Pre -Sym -v	-Pre
1	8	2	3	3	2
2	1	1	1	1	1
3	*	4	4	5	5
4	*	*	*	*	*
5	13	3	2	2	9
6	*	5	5	6	3
7	*	*	30	*	12
8	*	*	*	*	22
9	*	8	14	8	*
10	*	*	*	*	*
11	*	*	*	*	*
12	*	*	*	*	*
13	*	*	*	*	*
14	*	*	*	*	*
15	*	*	*	*	*

Figure 11: Weighted almost matches results

switches. Mostly it could be found in the top 10, but sometimes just outside the top 15.

3.5 Suspicious matches

Each of the previously mentioned comparison methods also lists the amount of suspicious matches found. We found that these numbers are not sufficient to indicate plagiarism itself, but they can help detect false positives. Files that rank high but contain no suspicious matches might be false positives. Files with very low rankings but a lot of suspicious matches might indicate false negatives. We should note that this method stops working when a certain amount of preprocessing is used. Because preprocessing limits the set of characters, the chances of having a k-gram that appears only in a small set of document decreases.

3.6 Expanding suspicious matches

These results list all suspicious matches between two files. It turned out however that these lists were too large to conclude anything from. All data in the lists was correct, and showed strings that two files had in common, but no conclusion could be made upon viewing this data. When specifically searching for known plagiarism cases, the results disappoint and could not prove any definite plagiarism. This method suffers from the same drawback as the “Suspicious matches” method: It does not work when using too much preprocessing.

3.7 Multiple lines matching

This method is not meant to work on it’s own and thus it does not provide any helpful way of indicating plagiarism. It does however give great support. When one of the other methods has indicated some plagiarism, the results of this method can help you find it

and judge it. Most results that belong to a case A or B pair of files have a very good indication of where the plagiarism is located. Files in case C are less likely to give a correct result, as it is not always clear if these files contain plagiarized code or it could be edited to such an extend that k-gram matches can no longer be found. The files is case A have a nearly 100% line match. This method also suffers from the same drawback as "Suspicious matches", but it improves when using `-S`, `-C` or `-IO`. `-S` prevents from spaces from being matched to each other. Almost each line contains a 3-gram with 3 spaces. This method would conclude that each line has a match this way. `-S` prevents that. `-C` prevents comments from intervening with lines. When two pieces of code are identical, but one contains comments in between the lines, this method might fail to match the lines of code, because it tries to match with the lines code in f_1 with the lines of comments in f_2. The switch `-IO` improves by eliminating natural language. Lines of output contain strings that are easily changed. Trying to match these strings often returns a negative result. Thus replacing them with special characters indicating output enables us to say that lines of code that output strings are similar, regardless of the content of those strings.

3.8 Moss2

To put our results in perspective, we will briefly discuss the results of MOSS2. We will divide all results into the cases we defined in Section 3, and elaborate on a few interesting cases.

The results of MOSS2 can be seen in Figure 12. The top 9 is a very good result. All files

MOSS2	Case
1	B
2	A
3	B
4	B
5	B
6	B
7	B
8	B
9	B
10	C
11	C
12	C
13	C
14	C
15	C

Figure 12: MOSS2 results

are almost certainly plagiarized. Results 10 to 15, however are less interesting. These file pairs show some similarities, but this might very well be a coincidence. With this info we can conclude that all of the previous methods did a good job in not finding the last 6 false positives that MOSS2 falsely recognized as plagiarism. On the other hand many of the previous methods contain many false negatives: Files that MOSS2 did correctly

recognize as plagiarism but our methods did not place within the top 30. It is remarkable that number 16 and 17 (not shown in the table) might very well be pairs of files that contain plagiarized code, but result number 20 belongs to case D, and thus is in no way plagiarized. We know this because one of the files in this pair is a file not written by a student, but by the teacher of the course.

3.9 Second test set

Based upon the results from the first test set, we have conducted a few experiments upon a second set of C++ files. This second test set contained 102 files created by students for an assignment in programming course. For this assignment students had to program Conway's Game of Life [1] and they had to make use of C++ classes, which was not done for the run-length assignment. Because of this, the second test set could be considered more complex.
We conducted the experiments with the best results on the first test set, and the experiments without preprocessing and with all preprocessing switches enabled. We also ran Moss2 on the test set. The results are as follows:

1. Moss2 listed seven file pairs within the top 15, that we judged as plagiarized. Four of these file pairs are categorized within case B, one pair of these four might even be considered a case A. The remaining three file pairs belong to case C.

2. The Weighted Almost method with preprocessing switches `-Pre -Sym -v` found five pairs of files that Moss2 listed in the top 15. Four of these five file pairs can be seen as a case B file. The other file we categorized as a case C, since we are unsure if it is plagiarized. All other file pairs in the top 15 appear unlikely to be plagiarized upon manual inspection. The Weighted Almost method was able to find four file pairs that Moss2 listed in the top 15, with other preprocessing switches. These switches were: No preprocessing, `-Pre`, `-Pre -Sym`, or `-S -C -IO`.

3. Other methods, including variants of those methods with different preprocessing switches enabled, contained three or less file pairs the top 15 of Moss2 also listed. These pairs that were found were often listed high by Moss2, and we did judge them to be plagiarized after manual inspection.

4 Related work

There are already many papers on the subject of plagiarism, but relatively little has been released on the specific subject of plagiarism between a set of C++ files. Here we will discuss tree other methods of detecting plagiarism in program code.

- Winnowing
- Machine Semantic Analysis
- Parse Tree Analysis

For further reading and references, please see the references list [7].

4.1 Winnowing

Winnowing is an algorithm for selecting document fingerprints (as defined in [9]) from hashes of k-grams, used by the MOSS2 program. Much like the method we used, this method relies on k-grams, but also uses more information like line numbers of a k-gram. A file is transformed into a file without spaces, comma's and other features that are irrelevant according to the writers of [9]. Then k-grams are made of the file, and each k-gram is hashed. This hashes are then grouped in 'windows'. A hash can appear in more then one window. If a window has hashes (w, x, y), the next window will contain (x, y, z). The smallest hash of each window is then choses as fingerprint for the document, if that has was not already used as a fingerprint.
The authors state that when selecting the right window size, k-gram length and detection threshold (length of substring to count as match), this guarantees that the chosen hashes will have a match with other documents' fingerprints, if there is plagiarism.
Also see [9].

4.2 Machine Semantic Analysis

The authors of the article [8] developed a system using semantic analysis as a detection method. The system itself is used for natural languages, but the method behind it can be of value when detecting plagiarism in source code.
Their main method is called tokenization. It converts an entire file to token-strings. Strings that only contains tokens and operators. All variable and function names are removed, so that a program only has to check for which operations are being done.

4.3 Parse Tree Analysis

As the name suggests, this method uses the parse trees of source code to compare them. In [10] they used an existing compiler create a program that constructs parse-trees of source code. The higher structure of these trees are compared, and when two subtrees seem similar, they are compared on a more lower level. The lower level comparison does more checks in an attempt to detect changes in operators or order of the code.

5 Conclusions and future work

In this paper we presented our methods of plagiarism detection and forms of preprocessing. Our methods were based on k-grams as presented by MOSS2. We provided 3 methods of basic detection, that we called "Exact Matches", "Almost Matches' and "All Matches". We used tf-idf to expand upon these methods, and gave the user the option to get "Weighted Exact Matches", "Weighted Almost Matches' and "Weighted All Matches" as results. Besides these results, we give the user 3 other ways to judge the results we presented them. These ways are "Suspicious Matches", "Extended Suspicious Matches" and "Multiple Lines Matches".
As a result of our experiments, we can conclude that the program was most successful when using either our Weighted Exact Matches method with `-Pre -Sym -v`, Weighted Almost Matches method with `-Pre -Sym -v -c` or `-Pre`, or All Matches method with `-Pre -Sym -v -c` or `-Pre`.

- Weighted exact and Weighted Almost matches have the highest rate of plagiarized files found in the top 30 with 8 cases found by us.
- All matches finds less plagiarized files, with 6 cases within the top 30 that we also judged to be plagiarized, but had less false positives, since these 6 files were all sequentially ranked starting from rank 1.

Other methods, like the "Weighted All matches", did not provide any result that could be considered helpful for finding plagiarism except for files that are nearly identical. We do not recommend this method when higher degrees of plagiarism have to be found.
Our supporting method Multiple Lines Matches did a good job in pointing out plagiarism when other methods gave an indication which files were plagiarized so we can recommend using this method as much as possible. This might require rerunning the user's test with the `-S` or `-S -C` switches, and renaming or moving the user's current results.
We would like to end the conclusion with a few general notes on our methods:

- Do not rely on one single method to find all the plagiarized files. Our results have indicated no single method is capable of finding all cases of plagiarism.
- Use the results a guides only when determining plagiarism. Do not rely on any of the methods to guarantee plagiarism in a file when a a pair of files is ranked high in the results.
- We mainly used one test set of 108 files made for the same assignment. It is possible that because of these conditions the results may vary when using other sets of files. We expect that our files were all very similar due to the fact that they all came from a relatively simple assignment. Smaller experiments done upon a second test set from a more complex assignment have indicated that this could indeed be the case. It also appeared that the Weighted Almost Method was still preferred to detect plagiarism, although the results of our experiments were less positive than the results of this same method on the first test set.

5.1 Future work

Many ideas still remain untested. We will briefly discuss a few of them that we think might improve upon our current methods.

5.1.1 Detecting suspicious files

It can happen that a file is likely to contain plagiarism even before it has been actually compared to other files. These files we call suspicious. We have illustrated these method in Figure 1 with the "Suspicious File Indication" line. We suggest the following methods to detect suspicious files.

- Writing-irregularities.
  When text is blindly copied from another source, the markup is often different from the rest of the document. There can be small differences such as the size of a tab, the use of (or lack of) spaces between variable names and operators, or the choice

of operators such as `+=1` or `++`.
We suggest that a list of "rregularity checks" is made that keeps track of standard tab-size, spaces or no spaces on certain locations, and other such properties of a document. When a piece of code does not comply with the standard styles used, this should be reflected in the documents fingerprint since there will be a good chance another document contains the same or highly similar code.
One should keep in mind that since this plagiarism detection program is mostly used for first year students, their code can be very irregular and can contain multiple different styles regardless of what is taught in class or is used most in a document. Therefore the definition of "irregularity" should not be based on just the most used style but on all styles used. Only when one style is used very little compared to the other styles, it should be marked as irregular.

- Incorrect or suspicious comment placement.
A very often used technique to hide plagiarism is the changing of variable and function names, or the switching of lines and operations. Since this is a very error-sensitive task, small mistakes are easily made and there exists a decent chance that comments contain the old name of a variable or function.
We suggest that certain locations or comments are checked for names that do not match the actual names used. A fine example of this would be the comment at the end of each function. Many first year students end a function with a comment stating the function name. If this comment is present and does not match the function name, this comment can indicate plagiarism and is a reason to inspect this file further.

- Shady programmers.
A very simple method for marking files as suspicious is keeping track of the writers, and their plagiarism history. When a file is written by at least one person who has plagiarized before, this file should be marked as suspicious and can be inspected more thoroughly.

- Late hand-ins.
According to teacher of the programming course [6], plagiarism mostly happens in files that are handed in late. The early submission rarely to never contain plagiarism. Limiting you file set to late submissions only might speed up the comparison methods, but we suggest marking all late submissions as suspicious.

5.1.2 Improving plagiarism detection

We suggest the programs plagiarism detection can be improved with the following ideas.

- Structure.
We suspect that some forms of plagiarism can be found by comparing the structure of the code. The structure of a code could be defined in numbers. For example, one could count the number of operations or lines between two succeeding while-loops. When we look at figure 2 we could say that the distance between the two while-loops is 33, which is the number of characters between the first `W` and the second `W`. If we were to do this for every pair of succeeding while-loops, we get a list of numbers that represent the while-loop-structure of a file. Comparing these sort of lists from two files might reveal plagiarism. The same method is applied for if-statements.

- Improving detection method relations.
  Currently, several different comparing methods are applied, and each of them outputs a number describing the distance between two files. With this distance a conclusion about plagiarism can be made. We suggest that a better conclusion could be made if only one value of distance was outputted, based on the results of all the comparing methods. The way each method can helps at determining this value i with the help of a neural network. Each method would determine individually if and how much a document is plagiarized. The neural network could take all these inputs and determine if the document is actually plagiarized.

- Line numbers.
  In the current state, the program does not check for any line numbers. While these numbers are easily changed by simply adding or removing just one line, when a certain part of the code matches code from another file, having the same line number(s) makes it all the more suspicious. But not only exact matches between line numbers can be of use. When multiple matches are found near each other, this could be more indication that that certain part of the code is indeed plagiarized. In this case one could also use a more precise (but more time consuming) method of comparing on this smaller piece of code. This way one could detect plagiarism even when the order of operations is changed.
  Note: The current program does have comparison methods that includes the location of a k-gram, and matches k-grams near each other, as seen in 2.3. It displays the lines which might be plagiarized, but it does not do extra comparing methods on these parts of the code, as we suggest here.

- More preprocessing.
  The existing preprocessing has the ability to generalize many aspects of a file, such as variable names or function names. This could be further improved by generalizing operators too. There are many ways of writing the same expression. If all similar expressions would be converted to the same way of writing, this would negate any changes made to expressions.
  One could also scan for the use of functions. It could be that a plagiarizer has taken a function call, and replaced it with the code of the function itself. Or the other way around, replaced a part of the code with a function. When a function call is found, it can be replaced by the function code. Alternatively, one could only replace function calls if it is the only call for this particular function. This would prevent that files grow to an enormous size or that the plagiarism detection gives distorted results in case only the function was plagiarized, but called multiple times (this would result in the function being marked as plagiarized each function call, heavily influencing the plagiarism results).
  The last preprocessing idea we propose is an improvement upon the existing variable name generalization. The current switches allow for replacing each variable with the letter V, or rename them to V_X, where X is the number of the order in which this variable appeared in a block of code. This works fine when variables are only renamed, but it fails when the order of variables is changed. For better results, the renaming should not only be based on the order of appearance, but also on the type of variable. The first integer would become $V1$, and the first character would become $C1$ for example.

Note: It is desirable to rename to one letter and one number only, so that the length of a variable name stays below the length of the k-grams. See Section 2.3.

References

[1] Conway's game of life
http://conwaylife.com/. [Accessed: 26–08–2013].

[2] Website Van Dale woordenboek
http://www.vandale.nl. [Accessed: 10–07–2013].

[3] Wikipedia
http://en.wikipedia.org/wiki/Tf%E2%80%93idf. [Accessed: 10–07–2013].

[4] A. Aiken. Moss: A system for detecting software plagiarism
http://theory.stanford.edu/~aiken/moss/. [Accessed: 10–07–2013].

[5] M.G. Ellis and C.W. Anderson. Plagiarism detection in computer code. 2005.

[6] W.A. Kosters. Programming methods
http://www.liacs.nl/~kosters. [Accessed: 10–07–2013].

[7] T. Kowaltowski. Plagiarism detection in computer programs: A bibliography (preliminary version)
http://www.ic.unicamp.br/~tomasz/projects/plag_bibl.pdf, 2010. [Accessed: 26–08–2013].

[8] M. Mozgovoy, V. Tusov, and V. Klyuev. The use of machine semantic analysis in plagiarism detection. In *Proceedings of the 9th International Conference on Humans and Computers*, pages 72–77. 2006.

[9] S. Schleimer, D.S. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In *SIGMOD '03 Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 76–85. 2003.

[10] J.W. Son, S.B. Park, and S.Y. Park. Program plagiarism detection using parse tree kernels. In *Proceeding PRICAI'06 Proceedings of the 9th Pacific Rim international conference on Artificial intelligence*, pages 1000–1004. 2006.