Second Origin

Capstone Report

Achraf Mamdouh

Supervised By

Dr.Violetta Cavalli Sforza
SECOND ORIGIN

Capstone Report

Student Statement:
I hereby affirm that I have applied ethics while designing Second origin.

[Signature]

Achraf Mandouh

[Signature]

Dr. Violetta Cavalli-Sforza
## Contents

1 Introduction .................................................. 1

2 System Requirements ........................................ 2

3 Background and Related Work .............................. 3

3.1 Levels of Plagiarism ........................................ 3

3.2 Techniques for Plagiarism Detection in Source Code .... 4

3.3 The Winnowing Algorithm .................................. 5

3.3.1 Detecting Partial Copies in a Document ............... 6

3.3.2 Winnowing Algorithm ................................... 6

3.3.3 Expected Density of Winnowing ......................... 7

3.4 Abstract Syntax Tree ....................................... 8

3.4.1 LibClang .................................................. 9

3.5 Suffix Arrays ................................................. 10

3.5.1 Finding Repeated Substrings in a String .............. 10
3.5.2 Finding Common Substrings between two Strings

4 Second Origin’s Algorithm
   4.1 First Pass of the Algorithm
   4.2 Comparing Two Programs
   4.3 The Blamer
       4.3.1 Increasing the Precision of the Blamer

5 Experimental Results

6 STEEPLE Analysis

7 Conclusions
List of Figures

3.1 Levels of plagiarism ............................................. 4
3.2 How to Fingerprint a Document ................................. 6
   (a) Some Text ...................................................... 6
   (b) The Sequence of 4 grams from Text (a) ...................... 6
   (c) Hypothetical Hashes for the 4 grams in (b) ................. 6
   (d) Selected 4 gram Hashes using 0 mod 3 ...................... 6
3.3 How to Fingerprint a Document using Winnowing .......... 8
   (a) Some Text ...................................................... 8
   (b) The Sequence of 3 grams from Text (a) ...................... 8
   (c) Hypothetical Hashes for the 3 grams in (b) ................. 8
   (d) Selected 3 gram Hashes using Winnowing with \( t = 5 \) .. 8
3.4 Example of Abstract Syntax Tree ............................... 9
   (a) Code Snippet ................................................ 9
   (b) Abstract Syntax Tree of (a) ................................ 9
3.5 Suffix Array for \( s = \text{abracadabra} \) ......................... 10

3.6 Finding Common Substrings in Two Strings .................. 12
   (a) Two Strings ................................................. 12
   (b) Suffix Array of \( S = s_1 + * + s_2 \) ......................... 12
   (c) Repeated Substrings in \( s_1 \) and \( s_2 \) ......................... 12

4.1 Example of Tokenization .................................. 13
   (a) Code Snippet .................................................. 13
   (b) Token Types of (a) ............................................. 13

4.2 Example of Code ............................................ 15

4.3 AST of code in figure 4.2 .................................. 15

4.4 Flattened Sequence of the AST in figure 4.3 ............... 15

4.5 Example of Stringer ........................................ 18
   (a) AST Example ................................................... 18
   (b) Stringer of AST in (a) ........................................ 18

4.6 Matching Functions between two Programs ............... 20
   (a) Hypothetical Graph for two Programs ..................... 20
   (b) Optimal Assignment for Graph in (a) ..................... 20
Abstract

Plagiarism detection in programming classrooms is an important step when reviewing students’ code in order to enforce integrity. This, when it is done, usually requires a significant investment of time and effort and deviates faculty and graders from their main task.

We present in this report Second Origin, a plagiarism detection tool for C and C++ languages. Second Origin tries first to detect potentially plagiarized files using the Winnowing algorithm. This approach fits perfectly in a changing environment where project submissions accumulate over time (e.g., at a University). Then, if a file is suspected of being plagiarized, Second Origin runs further analysis based on the Abstract Syntax Tree of the input program, and the potentially matching programs in the reference database. We then propose a new algorithm for blaming plagiarized sections that are then showcased to the user through a UI.
1 | Introduction

Digital content can be easily plagiarized. While it is easy to detect exact copies in a set of documents – by using their check-sums for example, it is more challenging to detect partial copies. Spotting plagiarism instances manually in large classrooms is hard and impractical. The use of an automatic tool would help instructors and graders. In this report we give a general algorithm for detecting plagiarism in computer programs that is suitable for a classroom setup.

Chapter 3 gives a brief overview of previous research in plagiarism detection in computer programs. We then, give all the necessary background before introducing Second Origin’s Algorithm in Chapter 4. In Chapter 5, we discuss our experimental results. Chapter 6 has a STEEPLE analysis of this project, and finally we conclude in chapter 7.
The main objective of our system is to detect plagiarism from a set programs. This is suitable in a classroom level where students submit assignments, and integrity should be enforced.

Second Origin’s work started as a mean of investigating plagiarism in source code based on existing work (e.g., MOSS [1], J-PLAG [2] ... etc). The goal was to create an automatic tool that would help instructors to catch cheating by highlighting potentially copied sections, then the instructor would assess their validity. We wanted the system to process the submitted programs in a reasonable amount of time – a couple of minutes, avoid false positives \(^1\), and have a user friendly interface.

---

\(^1\) A test result that says that a given condition is met while it is not.
3 | Background and Related Work

Before looking for plagiarism across files, we first had to understand how plagiarism works in a classroom level. [3] provides a good survey of plagiarism detection algorithms and ethical dilemmas around this subject.

3.1 Levels of Plagiarism

Parker and Hamblen in [3] have distinguished six levels of copying in source code. Figure 3.1 depicts those 6 levels.

Level 0 is the original file. In levels 1 and 2, the changes introduced are superficial (e.g., changing variable names, altering the indentation, and changing the comments). The person plagiarizing usually does not understand the original source code. Near the other end of the spectrum, at level 5, programming constructs are substituted with equivalent ones. For example, changing a for loop to a while loop or vice versa.

Levels 0 and 1 are easy to detect using string based comparison algorithms. While the other levels require more sophisticated techniques.
3.2 Techniques for Plagiarism Detection in Source Code

Castro in [4] calculate the longest common subsequence (LCS) between two programs. While there are no false positives \(^1\), this approach give poor results in case the order of statements are shuffled. Computing the LCS of two sequences of sizes \(n\) and \(m\) takes \(O(n \times m)\), this makes this technique impractical when we want to check for plagiarism in a large database of submissions.

\(^1\)A test result that says that a given condition is met while it is not.
Others use local fingerprinting algorithms to search for similarities. A famous example is Moss [1]. Moss operates on tokens rather than the original source code. It uses the Winnowing algorithm to detect plagiarism.

[5, 6, 7] make use of the Abstract syntax tree. [5], for example, hashes the subtrees of the AST. When two programs share a hash then it is likely that the two programs have similar sections of code.

Before we present Second Origin’s main algorithm, we will lay out the ground by giving a description of Winnowing, Abstract Syntax Trees, and Suffix Arrays.

### 3.3 The Winnowing Algorithm

Detecting partial copies across documents is fundamental in plagiarism detection. In the section below we explain the Winnowing algorithm for producing document fingerprints. The most appealing feature of Winnowing is that parts of a (long) match between two documents are guaranteed to be detected.

We first start by defining \textit{k-grams} as they are essential to the core algorithm of Winnowing.

**Definition 3.3.1.** \textit{k-gram} A \textit{k-gram} is a collection of \textit{k} contiguous items. The elements can be words if we are working on linguistic texts or tokens in programming source files for example [8]. The \textit{k-grams} of a sequence \(S\) are the collection of every \textit{k} consecutive elements from \(S\).

For example, given the sequence ‘\texttt{a b c d e f g h}’ its \textit{k-grams} for \(k = 5\) are:

\[
(a \ b \ c \ d \ e)(b \ c \ d \ e \ f)(c \ d \ e \ f \ g)(d \ e \ f \ g \ h)
\]

\textit{k-grams} can make up a fingerprint for a digital document, and hence can be used for plagiarism checking. \textit{k-grams} are usually stored in a database.
after being hashed for storage efficiency. The Rabin Karp algorithm [9] can be used to build the sequence of hashes in linear time.

3.3.1 Detecting Partial Copies in a Document

One possible algorithm for catching partial copies uses the following idea. We compute the \( k \)-grams of a document, those \( k \)-grams are then converted to hashes. Among all the hashes generated, we keep only a small subset which represent a fingerprint of the document (see figure 3.2). If two documents share at least one hash, then it is likely that they share a \( k \)-gram as well. Note that two different \( k \)-grams may still generate the same hash.

\[
\begin{align*}
\text{the lazy fox jumped over the lazy dog} \\
\text{(a) Some Text} \\
\text{(the lazy fox jumped)} \\
\text{(lazy fox jumped over)} \\
\text{(fox jumped over the)} \\
\text{(jumped over the lazy)} \\
\text{(over the lazy dog)} \\
\end{align*}
\]

\[
\text{(b) The Sequence of 4 grams from Text (a)}
\]

\[
45 31 44 81 181
\]

\[
\text{Hypothetical Hashes for the 4 grams in (b)}
\]

\[
45 81
\]

\[
\text{Selected 4 gram Hashes using 0 mod 3}
\]

\[\text{Figure 3.2: How to Fingerprint a Document}\]

3.3.2 Winnowing Algorithm

From a set of \( k \)-gram hashes of a document \( d \), the Winnowing algorithm selects a small subset which represent a fingerprint of \( d \). The hashes are selected such that when we are looking for substring matches across documents, two properties hold:
1. Matching substrings of length smaller than a noise threshold $n$ are never detected.

2. Matching substrings of length bigger than or equal than $t$ should always be detected.

The constants $n$, and $t$ are set by the user — $t$ should be obviously as large as $n$. To avoid matching on substrings less than $n$, we consider only $k$-grams with $k \geq n$.

To detect matching substrings of length bigger than or equal to $t$ in a consecutive sequence of $k$-gram hashes $h_1, h_2, \ldots, h_m$, where $m > t - n$, at least one $h_i$ has to be selected. Notice that the total number of words that represent a sequence of $m$ $k$-grams is $k + m - 1$: $h_1$ contributes with $k$ items, and each subsequent hash $(h_2, \ldots, h_m)$ adds only one extra item, hence giving $k + m - 1$ consecutive words.

If we set $w = t - k + 1$, then for each window of $w$ hashes, we have to pick at least one hash to guarantee the second property. Notice that for a window of length $w$ we have exactly $t$ consecutive items.

This leads to the following algorithm: for each window of size $w$, we choose the smallest hash value in that window. If a hash is repeated over consecutive windows we count it only once. The chosen hashes represent a fingerprint for the document $d$. The idea behind selecting the minimum hash $h_{\text{min}}$ is that it is likely for $h_{\text{min}}$ to be again the minimum in the next adjacent windows. Figure 3.3 gives an example of how Winnowing works. Notice that in subfigure 3.3 (c) we have $w = t - k + 1 = 3$.

### 3.3.3 Expected Density of Winnowing

Let $X_i$ be a random variable that gives the probability of choosing a new hash from window $W_i$. Let $W_{i-1}$ be the previous window. Both $W_i$ and $W_{i-1}$ account for a total of $w + 1$ hashes, and have an intersection of $w - 2$ hashes.

For Winnowing to not choose a new element at step $i$, the minimum hash of both $W_{i-1}$ and $W_i$ has to be part of their intersection. This happens with
the lazy fox jumped over the lazy dog
(a) Some Text

(the lazy fox)
lazy fox jumped
fox jumped over
(jumped over the)
over the lazy (the lazy dog)
(b) The Sequence of 3 grams from Text (a)

105 31 44 79 1 19
(c) Hypothetical Hashes for the 3 grams in (b)

31 1
(d) Selected 3 gram Hashes using Winnowing with $t = 5$

Figure 3.3: How to Fingerprint a Document using Winnowing

probability of $\frac{w-2}{w+1}$, and thus $X_i = \frac{2}{w+1}$. Therefore the expected number of hashes that Winnowing will select as a fingerprint is:

$$E = \sum_{i} X_i = N \times \frac{2}{w+1},$$

where $N$ is the number of windows.

Since we have as many windows as words in the document, then the density is $2/(w + 1)$.

### 3.4 Abstract Syntax Tree

The Abstract syntax tree (AST) is a tree that represents the structure of a program. An example of an Abstract Syntax tree is given in figure 3.4.

Abstract syntax trees are used by compilers during semantic analysis. ASTs can also be used by users to create programs that operate on source code, such as refactoring tools, syntax highlighters and more.
int add(int a, int b) {
    return a + b;
}

(a) Code Snippet

(b) Abstract Syntax Tree of (a)

Figure 3.4: Example of Abstract Syntax Tree

3.4.1 LibClang

In our system we rely on LibClang [10] for building Abstract Syntax Trees of the programs that will be checked for plagiarism. LibClang provides a C API for scanning and parsing C/C++ code, and routines for traversing ASTs.
### 3.5 Suffix Arrays

**Definition 3.5.1.** Suffix Array A suffix array (SA) of a string $s$ is a lexicographically sorted array of all suffixes of $s$. It’s a simple yet powerful data structure for string processing [11].

An example of suffix array is given in Figure 3.5.

Building the suffix array of string $s$ can be done in $O(n)$, where $n$ is the number of characters in $s$.

#### 3.5.1 Finding Repeated Substrings in a String

A useful operation in suffix arrays is calculating the longest common prefix (lcp) of two suffixes. For example, abra and abracadabra have an lcp of 4.

Since the suffixes are sorted in lexicographical order, then suffixes that share a common prefix are adjacent to each other. It suffices then to iterate over the suffixes in sorted order, and then check if the current suffix and the next one have an lcp bigger than 0. If it is the case then a substring of length
equal to their \( lcp \) occurs more than once.

In the string above, we have the \( lcp \) of suffixes 5 (bra), and 6 (bracadabra) is equal to 3. Hence, the substring bra is repeated.

### 3.5.2 Finding Common Substrings between two Strings

We can reduce the problem of finding common substrings between two strings \( s_1 \), and \( s_2 \) to finding repeated substrings in a single string. The reduction step consists of concatenating \( s_1 \) and \( s_2 \) which we call \( S \).

When we find a repeated substring in \( S \), we have to check if it occurs in both \( s_1 \), and \( s_2 \). This can be done using the lengths of the two suffixes where this common substring occurs.

We have to be careful though with substrings that span over two strings, in other words, those substrings that are composed of a suffix of \( s_1 \), and a prefix of \( s_2 \). A useful trick to avoid such cases is to concatenate the two strings using a symbol that does not occur in any of them (e.g., \( s_1 + \ast + s_2 \), where \( \ast \) is a symbol that does not exist in both \( s_1 \) and \( s_2 \)).

If we take index 3 and 4 in figure 3.6 (b), we have a \( lcp \) of 2, hence bc is repeated in both \( s_1 \) and \( s_2 \).
\[ s_1 = \text{abc} \]
\[ s_2 = \text{bcb} \]

(a) Two Strings

<table>
<thead>
<tr>
<th>index</th>
<th>suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>*bcb</td>
</tr>
<tr>
<td>1</td>
<td>abc*bcb</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>bc*bcb</td>
</tr>
<tr>
<td>4</td>
<td>bcb</td>
</tr>
<tr>
<td>5</td>
<td>c*bcb</td>
</tr>
<tr>
<td>6</td>
<td>cb</td>
</tr>
</tbody>
</table>

(b) Suffix Array of \( S = s_1 + * + s_2 \)

\[ b, c, bc \]

(c) Repeated Substrings in \( s_1 \) and \( s_2 \)

Figure 3.6: Finding Common Substrings in Two Strings
Second Origin’s Algorithm

4.1 First Pass of the Algorithm

The first step when checking plagiarism in a program $p$ consists of extracting the tokens from its files by means of lexical analysis. Those tokens are stripped from any details (e.g., for a constant, its value is ignored). Only their token types are kept (see figure 4.1). We use the LibClang API [10] for lexical analysis.

```c
int magic() {
    return 0;
}
```

(a) Code Snippet

<table>
<thead>
<tr>
<th>keyword</th>
<th>identifier</th>
<th>punctuation</th>
</tr>
</thead>
<tbody>
<tr>
<td>punctuation</td>
<td>punctuation</td>
<td>keyword</td>
</tr>
<tr>
<td>literal</td>
<td>punctuation</td>
<td>punctuation</td>
</tr>
</tbody>
</table>

(b) Token Types of (a)

Figure 4.1: Example of Tokenization

The stream of tokens is then converted to a set of $k$-gram hashes using the Winnowing algorithm [1]. Those hashes are looked up in a database containing $k$-grams of other projects. In case of a match, we mark the programs that contain this $k$-gram as candidates for further analysis. For each project only a small set of candidates is kept for which a more extensive analysis will be run.
4.2 Comparing Two Programs

From the small set of candidates that is left from the first pass, we compare the program $p$, that we started with, to each one of them.

When comparing two programs $p_1$ and $p_2$ for plagiarism. We first generate their Abstract Syntax Trees. LibClang [10] is used for this task. The AST gives more structure, and more information than tokens alone. In our system, some nodes from the AST are ignored because they are not relevant to plagiarism detection – usually LibClang is very verbose. The AST is then flattened by using a preorder traversal. Next, we generate all $k$-grams from the flattened AST. Figures 4.2, 4.3, and 4.4 demonstrate this process.

Then, we calculate a similarity metric based on 3 coefficients:

\[
\text{Jaccard Index}(p_1, p_2) = \frac{|k\text{-gram}(p_1) \cap k\text{-gram}(p_2)|}{|k\text{-gram}(p_1) \cup k\text{-gram}(p_2)|} \quad (i)
\]

\[
\text{Coverage Index}(p_1, p_2) = \frac{|k\text{-gram}(p_1) \cap k\text{-gram}(p_2)|}{\max(|k\text{-gram}(p_1)|, |k\text{-gram}(p_2)|)} \quad (ii)
\]

\[
\text{Inclusion Index}(p_1, p_2) = \frac{|k\text{-gram}(p_1) \cap k\text{-gram}(p_2)|}{\min(|k\text{-gram}(p_1)|, |k\text{-gram}(p_2)|)} \quad (iii)
\]

The similarity index is the average of the 3 indexes.

\[
\text{Similarity Index}(p_1, p_2) = \frac{(i) + (ii) + (iii)}{3}
\]

Intuitively, the Similarity Index gives an indication on how close the structure of the two programs are.

The use of $(ii)$ and $(iii)$ is to make the similarity index higher for programs that are small in size and included in larger ones. In case the smaller program
int magic(int a) {
    return a;
}

Figure 4.2: Example of Code

Function
 Parameter
 Body
 return
 variable

Figure 4.3: AST of code in figure 4.2

Function Parameter Body return variable

Figure 4.4: Flattened Sequence of the AST in figure 4.3

15
is almost included in the largest one – for example when \( k-gram(p1) \subseteq k-gram(p2) \), then the similarity index is approximated to:

\[
\text{Similarity Index}(p_1, p_2) \approx \frac{2 \times (ii) + 1}{3}
\]

Cases of plagiarism where only a small subset is kept were found during experimentation. In those cases, using the Jaccard coefficient alone gave a low similarity index.

We can get for each program the closest programs to it in terms of similarity. Here, a number of options are available for selecting which to keep. We explore two here. We can either set a threshold percentage \( p_{\text{plag}} \), and all projects that scores more than \( p_{\text{plag}} \) are kept as evidence, or select the \( h \) highest program. In the current system only the the project with the highest score is kept – we set \( h = 1 \).

4.3 The Blamer

For a tool like this to be useful, the matching sections between two projects should be visible to the user.

The plagiarized sections constitute substrings in the source file. However, merely looking for common substrings [11] between two programs may fail. In level 2 and above \(^1\) string based algorithm don’t work anymore because variables names can be changed, order of statements can be shuffled . . . etc.

We developed a new algorithm that detects occurrences of plagiarized sections. The algorithm converts the programs to a suitable representation which the general algorithm for finding common substrings between two strings works.

Intuitively, we will convert each program to a string, that we call a stringer. Common sections in two programs are also common substrings in their stringer representations.

\(^1\) Levels of plagiarism were introduced in section 3.1
To create a *stringer* from a program, we use algorithm 1. Note that in Algorithm 1 we can create the *stringer* while flattening the tree.

**Algorithm 1 Make Stringer**

1: procedure **MAKESTRINGER**\( (p) \) \( \triangleright \) Make stringer of program \( p \)
2: \( ast \leftarrow \text{parse}(p) \)
3: \( nodes \leftarrow \text{flatten}(ast) \) \( \triangleright \) using a preorder traversal of the ast
4: 
5: \( s \leftarrow \epsilon \) \( \triangleright \) empty array
6: for node in nodes do
7: \( s \leftarrow s + \text{typeof}(node) \) \( \triangleright \) append type of this node
8: end for
9: 
10: return \( s \) \( \triangleright \) The stringer of \( p \)
11: end procedure

The result is an array of tags that represents an abstract description of the program (see figure 4.5).

Using two *stringers* \( s_1, s_2 \) it is easy to find similar copied sections, it suffices to find common substrings between \( s_1 \) and \( s_2 \). In our system only common substrings that exceed a threshold \( t_{\text{plag}} \) are kept. The common substrings are then highlighted. In case two common substrings overlap we keep the one which has the largest number of tokens.

*Stringers* that were created using tokens did not give good results during experimentation compared to the *stringers* created using AST nodes. Actually, many long sections that are not similar were found to have the same token types.

### 4.3.1 Increasing the Precision of the Blamer

To increase the confidence when matching against two different sections of code. We developed a mapping of functions. A function is mapped to another function from a different program if and only if they are considered similar.

To construct the mapping, we first start by creating a bipartite graph
G = (L, R, E) where the set L represents functions from the first program, and R functions from the second one. Each function u ∈ L is connected with another function v ∈ R in G with a cost w representing an edge (u, v, w). The weights on edges represent the number of common k-grams between two functions u and v.

The k-grams of a function are extracted using the following algorithm.

**Definition 4.3.1.** The call graph of a function f is the set of functions that are called directly or indirectly from f. f is also part of its call graph. We can compute the call graph using a depth first traversal on the AST (see Algorithm 2).
Algorithm 2 Computing the Call Graph

1: `procedure visit(node, F)` \footnote{For AST traversal}
2: mark $f$ as seen
3: Add $f$ to $F$
4:
5: if node is a function call to $f$ then
6: if $f$ was not seen before then
7: \hspace{1em} `visit($f$)`
8: end if
9: else
10: for each child of node do
11: \hspace{1em} `visit($child$)`
12: end for
13: end if
14: end procedure

16: `procedure ComputeCallGraph($f$)`
17: $F \leftarrow \emptyset$ \footnote{Empty set}
18: `visit($f$, $F$)`
19: return $F$
20: end procedure
On the call graph of function $f$, we compute the $k$-grams of each function that appears in the set. The union of the $k$-grams constitute the $k$-grams of function $f$.

Once, our graph $G$ is constructed, we then look for an assignment that maximizes the sum of edge weights. This can be done using the Hungarian Algorithm [12] (see figure 4.6).

![Hypothetical Graph for two Programs](a)

![Optimal Assignment for Graph in (a)](b)

Figure 4.6: Matching Functions between two Programs

When a function is mapped to another one then, we have more confidence blaming section in codes that belong to both of them. Actually, two different thresholds $t_1$ and $t_2$ can be set, where $t_1 \leq t_2$. If a plagiarized section belongs to functions that are mapped to each other we use $t_1$, otherwise we use $t_2$. 

20
We have done two types of experiments. The first experiment's purpose was to try to find suitable values for constants used in our algorithm (e.g., noise value of $k$-gram). The second experiment consisted of testing the system against real data.

The data set of our first experiment was manually crafted. After trying our algorithm with different constants, we found that for Winnowing ($n = 4, t = 10$), and a threshold of $t_{plag} = 10$ for the blamer give good results.

Our second experiment was on 4 previous semesters of first year college student programs. The programs were of middle length ($\sim 300 - 400$ lines of code). The constants found on our previous experiments gave good practical results. We managed to catch all instances that were reported by the instructor plus some others. The programs that were not caught by the instructor were smartly plagiarized, and are hard to find by human eye — change of variable names, shuffling of statements, and using equivalent structures (e.g., \texttt{switch} instead of \texttt{if}). The blamer did not perform well on those instances, but the similarity metric gave a better hint.
The proposed project’s primary impact is in the societal, technical, and ethical spheres. Since the aim of this project is to develop a tool that would help instructors detect plagiarism in programming assignments, there is a clear ethical component. The focus is on introductory courses where the student has to submit a final project (e.g., CSC1401 in Al Akhawayn University). The use of the targeted tool should promote integrity among students from early on in their career as programmers, and should push students to avoid copying in assignments, and to rely more on their own abilities.

From a technical perspective, the project takes an existing algorithm and proposes some improvements and extensions that will potentially improve code plagiarism detection, therefore hopefully contributing to the current body of knowledge in this domain.

Finally, we aim to make Second Origin an open source project. This will not only contribute to the growth of the system itself, but would also help people interested in this subject use it both as a learning tool as well as a basis for making further advances in the field.
7 | Conclusions

We have presented in this report a general algorithm for checking against plagiarism in source codes. The system implemented works only with C/C++ programs, but the same algorithm can be extended to other programming languages, only the lexer and parser should be adapted. Moreover, we presented a new and efficient algorithm for blaming copied sections of code using suffix arrays, and a way to increase the confidence when blaming by mapping functions to their potential clones. We think that the system is useful to instructors who grade medium sized projects.
Bibliography


on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2014.


