CodeSimian – Code Similarity Analysis
Senior Project Report

Andrew Weng
California State University, Los Angeles
June 6, 2006
1 Abstract

Plagiarism is a serious problem in the educational community. With the advent of the widespread usage of computing on campuses in recent years, as well as easy access to the world’s largest public database of information in the world, plagiarism detection becomes a serious issue which must be addressed. Plagiarism or “code sharing” in Computer Science courses is also an increasingly prevalent issue. Successful plagiarism affects students not only by promoting cheating, but it negates the educational benefits of assignments and coursework.

Effective detection and the awareness of ease of detection is a powerful preventive tool which can ultimately benefit students in the long run. The application we present in this report intends to present a simple and accurate method in detecting plagiarism or code sharing in student source code submissions.

2 Introduction

Plagiarism is an issue concerning all fields of academia. In the past, plagiarism was a more difficult and involved task. Resources were limited, and the act of plagiarism involved manual copying. Published works were professionally published and its contents were commonly known among the learned community.

With the advent of the digital age, cheating through the unauthorized usage of publicly available works became significantly easier. Searching for resources can be as simple is an online search engine query, often resulting in a very large number of potential sources. The duplication of the relevant text involves only a simple copy and paste, resulting in the capability to patch a paper together from numerous sources. In addition, online sources are of varying quality, resulting in more difficult detection from simple inspection of content.

The internet promotes the sharing of information, which increases the public exposure of personal and privately owned intellectual property. Consequently, the inherently public nature of the internet can confuse the nature of the information available. The usage of such information can easily be misconstrued as “research” instead of theft of information technology, particularly with the ease with which information can be copied; the act of copying can seem inconsequential.
Source code plagiarism presents another dimension to the act of plagiarism from established and public code repositories online. The ease with which content can be copied from online sources is mirrored in the ease with which code can be shared among students. While collaborative work has its place in education, the individual effort involved in the implementation of concepts and ideas has tremendous benefit, all of which is lost when simply copying and pasting source code.

Although addressing academic integrity in classrooms is highly beneficial in discouraging code sharing and plagiarism among students, it is not sufficient to prevent students from doing so. By having a tool which can help identify similar code, professors can further discourage plagiarism and code sharing by publicly disclosing the plagiarism detection tools available.

The motivation behind this project was to provide assistance to professors who wish to enforce academic integrity policies without placing an undue additional burden on the professors. Although physical inspection of code can often easily identify suspect code, to do so on a large number of submissions can be daunting. Using a tool such as CodeSimian to identify highly suspect files is an important step in minimizing the time spent on enforcing academic integrity, while providing an effective deterrent against would-be code sharing students who fear being caught cheating.

After reviewing several solutions for plagiarism, we found many such solutions requiring registration to a website, and online submission of source code. These popular solutions, including JPlag (www.jplag.de), MOSS (http://www.cs.berkeley.edu/~aiken/moss.html), and SID (http://genome.math.uwaterloo.ca/SID/) all require registration and a submission process to have files analyzed.

Our solution is to present such a tool in a simple user interface which can easily assist in the identification of suspect source code files. CodeSimian (short for Code Similarity Analyzer\(^1\)) will run locally on the instructor’s computer, and allow easy access to files and directories. CodeSimian will list the applicable files in the user specified directory, and present an easy to understand matrix of percentage similarity between analyzed files.

---

\(^1\) CodeSimian is also a reference to the interpretation of “Simian” as monkey.
3 Technological Background

CodeSimian is built using Microsoft Visual Studio 2005 and the .NET 2.0 platform, coded in Visual C#. It is a Windows based application that aims to provide a simple and intuitive graphical user interface, providing as much information as possible with minimal configuration and tweaking possible.

3.1 The .NET Framework

The .NET Framework is a Windows component designed to build and run both managed and unmanaged applications in a consistent object oriented environment. The Common Language Runtime (CLR) and the .NET Framework class library are the two primary components in the .NET Framework. The Common Language Runtime is the core of the .NET Framework, managing code during runtime and providing services such as memory management, thread management, remoting and security. It can be compared to the Java Virtual Machine (JVM), arguably with some high-level and low-level differences. Stephen Gilmore deftly summarizes a key high-level difference in his talk on Resource-Bounded Functional Programming on the JVM and .NET:

- The Java Virtual Machine is an object-oriented execution environment for any language so long as it’s Java
- The .NET Platform is an object-oriented execution environment for any language so long as it isn’t Java

On the low-level end, the CLR always uses the Just-In-Time (JIT) compiler to compile into native code. The CLR also supports unmanaged code, making direct memory management and pointer arithmetic possible.

3.2 C#

The language we have used to develop CodeSimian is C#, and was developed by Anders Hejlsberg. Its focus was to provide similar procedural and object oriented design as C++, but addresses many of the issues which the lead developer felt plagued many of the existing object-oriented languages.
Many consider C# as the programming language which most directly represents the Common Language Infrastructure (CLI). It was designed from the ground up to take advantage of and integrate with the features that the CLI provides. The intrinsic data types within C# correspond to value-types implemented by the CLI framework, and thus seems to be the most logical language to utilize when developing under the CLI framework.

C#, pronounced as see sharp, is currently in its 3rd major revision, and is considered to be a Microsoft language, although the language itself has received standardization and approval from the ECMA in June of 2005.

4 Theoretical Background

Code plagiarism differs from literary plagiarism in that a minimal level of understanding is required for a programmer to read and understand code, and in turn, to implement the algorithms in their own program. As such, the program assumes that the extent of modifications to the code to avoid plagiarism detection are limited to adding single or minimal lines of code, removing and adding comments, and changing variable names or by rearranging declaration of variables. The student may also attempt to circumvent plagiarism detection by reformatting the code.

According to a study by Lancaster and Culwin, popular code detection engines such as JPlag and MOSS use paired metrics, structural metrics, and tokenization to detect plagiarism. Paired metrics refers to the practice of taking pairs of files in order to generate a similarity score. A less common method of detection is the usage of singular metrics, which involve generating a “fingerprint” of a document independently without the consideration of other documents.

Structural metrics refers to the act of utilizing some semantic knowledge in order to identify similarity, as opposed to superficial metrics, which utilize purely superficial analysis without any understanding of semantics. One benefit of this technique is the lack of source code dependence, allowing the detection routine to be utilized regardless of what language the code is presented in.

Finally, tokenization refers to the practice of reducing source code text to a representative symbolic form, such as replacing all variable names with a generic “variable name” byte representation.
The theoretical basis of the detection algorithm in CodeSimian shares many of the same attributes of these popular solutions. However, the application is based on the theory of Kolmogorov Complexity and Shared Information Distance and in order to identify similarity in code.

4.1 Kolmogorov Complexity

Kolmogorov complexity is the theoretical minimal descriptive length for any string. Given a string \( x \), \( K(x) \) is defined as the Kolmogorov complexity of \( x \). \( K(x) \) can be described as the theoretical shortest length program to generate the output \( x \) in a theoretical universal Turing machine language. For instance, the string containing the absolute value of \( \pi \) would be an infinitely long string: “3.1415926535897932384626433832795…” Equally, a true random number between 0 and 1 would be an infinitely long string: “0.52349587239573298753209…” Although both numbers would require an infinitely large storage space, the Kolmogorov complexity of \( \pi \) is small, as it would be described in formulaic form, such as Leibniz’s formula for \( \pi \):

\[
\pi = 4 \times \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = 4 \times \left( \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} \ldots \right)
\]

Computationally, Leibniz’s formula is highly inefficient, but the total space required to store the program is finite. The random number, on the other hand, would require an infinite amount of space to store the entire number.

A further definition of Kolmogorov complexity is required to describe its function in determining similarity between two strings. \( K(x|y) \) is defined as the Kolmogorov complexity of \( x \) given \( y \) as an input. The relevance for plagiarism detection and similarity analysis becomes clear with the following example: Given \( x = \pi \), \( y = \pi / 4 \), the Kolmogorov complexity of \( K(x|y) \) would be significantly smaller than \( K(x) \), as the information contained in \( y \) greatly assists in defining a program to generate the value of \( \pi \).
4.2 Shared Information Distance

This leads into the discussion of how Kolmogorov complexity can be used to ascertain the similarity, or shared information between two strings. As Kolmogorov complexity can be considered the amount of unique information contained with the output string, one can utilize that information to determine the relative amount of “uniqueness” or information distance contained in the string.

The following formula is the basis for the concept of distance between information, where \( K(x) \) and \( K(x|y) \) is the Kolmogorov complexity of \( x \) and \( x \) given \( y \) respectively:

\[
d(x, y) = 1 - \frac{K(x) - K(x | y)}{K(xy)}
\]

Another interpretation is to compute the similarity rather than the distance between the two strings \( x \) and \( y \). By subtracting the distance from 1, we obtain the following formula:

\[
s(x, y) = \frac{K(x) - K(x | y)}{K(xy)}
\]

This formula calculates the shared information between \( x \) and \( y \) by taking the information in \( x \) (\( K(x) \)), removing the information shared between \( x \) and \( y \) (\( K(x|y) \)), and then dividing the result by \( K(xy) \). \( K(xy) \) can be easily understood as the normalizing factor by considering two distinct possibilities.

The first possibility is that \( x \) and \( y \) contain the exact same information, or are the same value. In that case, the function could be rewritten as \( s(x,x) \). \( K(x|x) \) would become essentially 0, as it require no modification to the input to produce the output; essentially \{ return x; \} would be the only code. In turn \( K(xx) \) would essentially be the same as \( K(x) \), requiring only a simple loop within the theoretical program to repeat the original output string once more. The resulting statement would simplify to:

\[
s(x,x) = \frac{K(x) - K(x | x)}{K(xx)} = \frac{K(x) - 0}{K(x)} = 1
\]
This satisfies the conclusion that the two strings are 100% identical, normalized by concatenation of the two strings.

The second possibility to consider is where \( x \) and \( y \) are unique and mutually exclusive. We define the two strings to contain absolutely no information about each other. As a result, \( K(x|y) \) would become equivalent to \( K(x) \), as no information in \( y \) can be utilized to describe \( x \) in any way. Similarly, \( K(xy) \) would be approximately equivalent to \( K(x) + K(y) \), as there is no way to incorporate the reproduction of \( y \) into the program which would produce the value of \( x \). The resulting statement can be shown as follows:

\[
s(x, y) = \frac{K(x) - K(x|y)}{K(xy)} = \frac{K(x) - K(x)}{K(x) + K(y)} = 0
\]

This shows how the comparison of two completely independent strings would result in a resulting similarity factor of 0.

### 4.3 Plagiarism Detection

Thus, Kolmogorov Complexity and Information Distance can be seen as potent tools when used to determine similarity factors and commonality between strings. Utilizing this formula, one can see how English texts, images, raw data, and even binary byte code can be analyzed for similarity indices. Plagiarism detection is a wholly viable application of Kolmogorov Complexity and Information Distance.

### 5 System Overview and Algorithm Description

The most compelling question, following discussion of the theoretical benefits of utilizing Kolmogorov complexity, is how to design an algorithm which can effectively produce the requisite complexity values, or rather as close an equivalent as possible. One interpretation of finding the absolute minimum information required to program or describe information is by performing lossless compression on the string, minimizing the length of the code, or the amount of description required to represent the string. For example, one can see the relation when compressing a string composed of twenty 0’s. As such, a simple RLE (run-length encoding)
compression would produce something along the lines of “0,20”. This would be potentially similar to what the Kolmogorov complexity can be theorized as for such a string.

Thus we have decided to approximate Kolmogorov complexity using a specific application of the Lempel-Ziv encoding algorithm. Compression alone was not the issue surrounding the adaptation of the Lempel-Ziv algorithm, but specifically the behavior of the compression algorithm when compressing concatenated code. As mentioned earlier in this article, the Kolmogorov complexity of a string concatenated with itself should be as close as possible to the Kolmogorov complexity of the string alone. Standardized compression algorithms fail to accommodate for long repeating blocks, and so the usage of a theoretically unbounded buffer Lempel-Ziv compression was used.

In addition to unbounded buffers, a further method of optimizing Kolmogorov complexity is in the usage of approximate string matching. It would require the ability for the algorithm to detect that by utilizing an approximate string match and the appropriate modification calls, such as addition, deletion or swapping, that the resulting compression would be more beneficial.

## 6 Design and Implementation

CodeSimian is designed to be an easy to use and simple to understand application, which generates a table of similarity values between files located in one directory. Primarily intended to assist professors in the detection of plagiarism, the tool is a straightforward and simple, with minimal options and configuration. However, in order to be a successful and accurate plagiarism detection tool, the algorithm behind the interface needs to be sufficiently robust to handle common plagiarism detection avoidance.

Code plagiarism differs from literary plagiarism in that a minimal level of understanding is required for a programmer to read and understand code, and in turn, to implement the algorithms in their own program. As such, the program assumes that the extent of alteration for students attempting to plagiarize or share code is minimal, involving simple changes such as renaming of variables, reformatting of code, elimination or addition of comments, and inserting non-critical code such as debugging statements or command line output status statements. Another issue is
reorganization of independent methods within the program itself without affecting the application.

The application reads a directory, processes individual pairs for their similarity index and displays the directory contents both as a list and in a 2x2 matrix, with the similarity results of the file comparison pairs displayed within the matrix.

Similar to JPlag and MOSS, the technology utilized in CodeSimian is a paired and symbolic metric analysis using a parsed and tokenized source. The three major components in the analysis performed by CodeSimian can be broken down into three steps. The code pair is parsed and tokenized into a string pair. The string pairs are compressed into their respective compressed string pairs. Finally the approximated Kolmogorov complexity of the strings as represented by the byte size of the post-compression string pair is used to calculate the similarity factor of the string pair.

6.1 Parsing Code

Code parsing was performed by utilizing a resource known as ANTLR. The author of ANTLR describes tool as follows:

ANTLR, ANother Tool for Language Recognition, (formerly PCCTS) is a language tool that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions containing Java, C#, C++, or Python actions. ANTLR provides excellent support for tree construction, tree walking, and translation. There are currently about 5,000 ANTLR source downloads a month. ([www.antlr.org](http://www.antlr.org))

We chose this as the parser for CodeSimian as it is a well supported and consistently updated code parser, which handles various languages. The source code used in the project was generated by using a publicly available Java language descriptor file. When processed with ANTLR, it generated the source code used in CodeSimian. We compiled the generated sources as provided in the examples directory and integrated the source into the CodeSimian project. Implementation of the ANTLR libraries was a simple matter of reading the source code file being analyzed into the JavaLexer class. We then use the IToken class to process the parsed and tokenized files. The tokenized output is a collection of integer values which correspond to a
matching type as depicted in the JavaTokenType class. This integer list is then matched to the corresponding ASCII code, and becomes the resultant string.

6.2 Compressing Code

Compression was implemented using an adaptation of the Lempel-Ziv algorithm. Due to the specific requirements of the application in approximating Kolmogorov complexity, modification to the generalized formula were necessary. Specific testing for the compression routine was the compression of a string concatenated to itself compared to the compression of the string alone. By utilizing unbounded buffers, and pointers to the existing portions of the requisite string, we were able to generate compression values of the test case specified to a very tight accuracy, with the size of the self-concatenated string insignificantly larger than the size of the string itself.

However, due to the highly theoretical nature of this exercise, we chose not to implement the compression algorithm in such a way that it would be storable and decompressed. The additional time it would take to do so was not necessary, as the goal of the exercise was not to develop a specialized compression routine for the actual lossless storage of text which contain repeating blocks.

In order to calculate the Kolmogorov complexity of \(x\) given \(y\), we need to utilize \(y\) as a free library for the algorithm to access during the compression of \(x\). In order to emulate this without further specializing the compression algorithm, we approximate this value by compressing \(y+x\) and then subtracting the compressed \(y\). This is essentially extracting the unique information of \(y\) from \(y+x\) while letting \(x\) get the benefit of the preprocessed \(y\) as the compression algorithm moves from left to right.

6.3 Comparison Formula

The final comparison formula for the compressed strings are straightforward. Utilizing the information distance formula defined above, we can easily calculate the similarity index through the compression of the following string sets. Given \(x\) as one tokenized code string, and \(y\) as the other tokenized code string, we compress the following sets of strings:
\[ K(x) = \text{TokenCompress}(x) \]
\[ K(y) = \text{TokenCompress}(y) \]
\[ K(xy) = \text{TokenCompress}(x+y) \]
\[ K(yx) = \text{TokenCompress}(y+x) \]
\[ K(x|y) = K(yx) - K(y) \]

With all of these components, we can then apply the information distance formula to arrive at our similarity index for each pair.

### 7 System Evaluation | Performance Evaluation

Testing for CodeSimian was performed on several levels. Simple test cases were developed, most of them trivial in nature. In particular, testing for removal of white spaces and comments, reorganization of code, renaming of the variable names and addition of "junk" code was emphasized. Also tested were programs that implemented some sorting algorithm, all of which performed the same task, but in a different way. Attached as Appendices are the test sources for the trivial code.

Codesimian was also tested using actual student submissions from the CS201 class. We were particularly interested in the results from a beginning level course, as the program requirements are small and tend to be similar. We felt that testing against false positives is important, as this issue was one of concern.

#### 7.1 Trivial Test Cases

The trivial test cases proved very effective; with the modified LinkedList.java files all being caught with very high similarity factors. The sorting files were the most interesting, because of the high similarity factors between the files. However, upon visual inspection of the code for bubblesort.java, insertsort.java and selectsort.java, the files are indeed nearly identical, save for the only difference being the actual sorting algorithm implementation. Those proves that very high similarity factors are worth looking into, as they may present unexpected results.
7.2 CS201 HW1

The results of the testing for plagiarism among the homework assignment submissions for the CS201 class were surprisingly good. Note the very low (30-50%) similarity despite the simplistic nature of the assignment, and the very high percentage match on the plagiarized code. Only minimal modifications were made to the source code to mask the plagiarizing when visually inspected.
7.3 CS201 HW4

Similar results were recorded for homework assignment 4. However due to the class files being declared separate of the main program, with very specific and simple functions for the class files, comparison results showed that some similar matches were found. When inspecting the students with very similar matches, suspicious activity was found, for instance the main programming containing almost exactly the same code except for slightly different bounds checking on the user selection process for the main menu.
8 Conclusion and Future Work

The testing of CodeSimian showed positive results, with promising results under real-world testing. Several weaknesses within the specific implementation of the compression algorithm can be improved upon, perhaps by using Levenshtein distance to better utilize partial string matching and recognition. This would better recognize code which has elements swapped around by recognizing that the differences between two strings can be represented as pointers to swap, add, or delete portions of the string to make them match.

In addition, the user interface can be improved to demonstrate where matching blocks of code were discovered, in order to better help the testers verify by visual inspection code which is suspect, but not obviously marked as plagiarized; for example similarity detection which is in the 75%-85% range.

We are pleased with the results of CodeSimian, and believe that it can play a positive role in the deterrence and detection of plagiarism in source code among student submissions. The ultimate goal of CodeSimian, is to improve the learning environment for students, and to assist the academic staff in their ever-demanding task of developing and educating students.
References


