Plagiarism Detection in Computer Code

Matt G. Ellis, Claude W. Anderson
{ellism, anderson}@rose-hulman.edu

March 23, 2005

Abstract

The ease with which computer code can be copied by students presents opportunities for plagiarism on programming projects. Presently, instructors are provided with few tools (if any) that assist them in detecting possible plagiarism in an assignment. Therefore, the only solution is a tedious and error prone by hand check of student submissions, looking for similarity between any two. In this thesis, I discuss the history of software plagiarism detection as well as present new methods of detection and software that implements these methods.

Introduction

No matter the subject area, plagiarism is a very real concern for instructors. Furthermore, the ease with which one student can copy computer code from another increases the likelihood that some plagiarism will take place on an assignment. This, combined with the fact that many programs are graded on what results they produce sometimes with a cursory glace at the source code, if any provides a relatively low risk environment for a student to conduct plagiarism. Finally, even if instructors wish to check to make sure no two assignments are very similar, the numbers are once again in the students favor. For a typical entry level class, there may be over 100 submissions per assignment. Checking for plagiarism can easily become an extremely time consuming task.

Software which could identify pairs of assignments which contain a high degree of similarity would be extremely beneficial to instructors by allowing them to focus their time spent looking for plagiarism to likely plagiarized assignments. Furthermore, such software would provide peace of mind that each student submitted unique work. In this thesis, I present a model of what software plagiarism and similarity is, a history of software plagiarism detection, and then introduce a new method of plagiarism detection based on the structure of the *parse tree* for a program. I also present extensions to my method as well as provide and overview of software which implements these methods.

Model of Plagiarism

The accepted model of classifying plagiarism of computer code is due to Faidhi and Robinson.[2] In their paper, they presented a six level system for classifying program modification (Figure 1). Parker and Hamblen define a plagiarized program to be one "that has been produced from another program with a small number of text edit operations but [with] no detailed understanding of the program."[7]

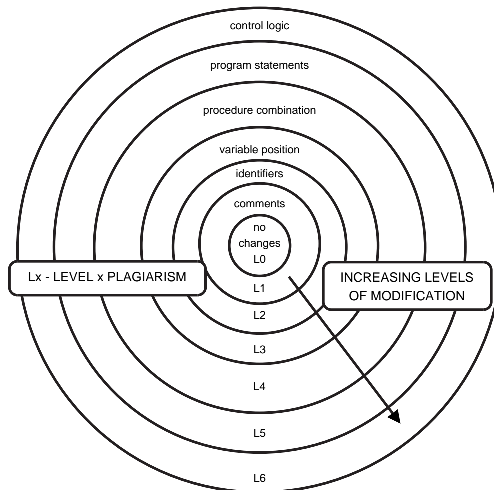

Figure 1: Program modification hierarchy

Taken together, this hierarchy system and definition provide the model with which we can describe plagiarism in student code. Note that as we increase in our plagiarism level, the amount of understanding required to make the necessary edits also increases. In fact, the first 3 levels (L0–L2) likely represent the majority of edits that take place when plagiarizing code. Some integrated development environments provided tools to aid with the most difficult of the three, changing identifiers, via refactoring tools.

Due to the above observations, focusing effort on detecting plagiarism which could be categorized in L0-L2, should help in identifying most common attempts at plagiarism.

History of Plagiarism Detection

The simplest algorithms for plagiarism detection focus around simple general purpose utilities for string searching. For example, using the UNIX utility `diff` provides a way to compare two files to one another for similarity. Indeed, simply running `diff` on all pairs of submitted programs would detect L0 plagiarism,

however more sophisticated techniques are needed to combat L1 and L2 methods. Heckel's algorithm [4], provides another tool to measure the similarity of two files. This algorithm differentiates between moved lines and changed lines, which would be helpful in combating the reordering of independent statements. Like the `diff` method, Heckel's algorithm, developed to find similarities between plain text files, is not resilient to changes in comments or identifiers.

Using the program `wc` to count the number of lines of output produced by `diff` (here, fewer lines indicates more similarity) would help to combat L1, and to an extent L2 modifications. However, renaming a frequently referenced identifier (or set of identifiers) could produce differences in almost every line of the copied file, thwarting this naïve method of detection.

Parker and Hamblen[7] discuss a more robust method of detection, which works by creating a metric used to measure each program. The metrics work by gathering statistics from a program (for example, the number of lines of a program, the number of `for` statements, etc.) and comparing the statistics of one program to another. An overview of some of the common metrics for Pascal and FORTRAN programs is provided in their paper.

Jankowitz introduced a novel algorithm in [5] for detecting areas of a program to focus on when looking for plagiarism. His method resolves around building a *static execution tree* for each program then looking for similarities between the trees. If a similarity is found, the two procedures from the source programs reflected in the static execution trees are analyzed using the above metric methods for similarity. This approach is one of the first that exploited the "structure" of a program, requiring that programs be parsed and analyzed in an intermediate form. However, Jankowitz still relies on metric methods which are tied directly to the language the plagiarism detection software is being written for. Furthermore, languages with *dynamic binding* like Java and C# (which are common languages for today's student programs) are difficult to construct execution trees for. The information needed to construct said trees is not often known until runtime. It is unreasonable to require each program be run in order to create an *execution trace* which could be analyzed later.

The next major improvement in plagiarism detection comes from Wise [8] which presents what has become the defacto standard algorithm used for modern plagiarism detection. Wise's algorithm measures similarity between strings by constructing tiles, which represent a 1-to-1 correspondence between substrings of a pattern string and a text string. By using a greedy algorithm (greedy in that it finds long running matches first), Wise presents an algorithm with a asymptotic worse case of $O(n^3)$ (here n is the length of the text) but with an observed runtime between $O(n)$ and $O(n^2)$.

Wise's method for detecting plagiarism involves tokenizing each file into a string of *lexical tokens* and then comparing these strings. Note that while the greedy nature of the algorithm may not find the best correspondence between two strings, for plagiarism detection we are mostly concerned with large running similarities, which the greedy algorithm discovers first. Furthermore looking for similarities in the lexical token strings of programs provides a level of robustness, especially against L1 and L2 plagiarism attempts.

Detection via Lexical Similarities

The process of *lexical analysis* takes the human readable source code for a program and converts it into a stream of lexical tokens which a parser or compiler may use to extract meaning from the source. During the lexical analysis phase, the source code undergoes a series of transformation. Some of these transformations, such as the identification of reserved words, identifiers, and numbers are beneficial for plagiarism detection.

Consider the following two snippets of Java Code:

```
int[] A = {1, 2, 3, 4};
for(int i = 0; i < A.length; i++) {
    A[i] = A[i] + 1;
}

int[] B = {1, 2, 3, 4};
for(int j = 0; j < B.length; j++) {
    B[j] = B[j] + 1;
}
```

Figure 2: Two similar Java code snippets

Clearly, these there is no sematic difference between the two code snippets. However, by changing the array `A` to `B` and the looping variable from `i` to `j` we have introduced differences on all but the last lines.

However, each snippet of code has the same lexical stream, which is the following:

```
LITERAL_int LBRACK RBRACK IDENT ASSIGN LCURLY NUM_INT COMMA NUM_INT
COMMA NUM_INT COMMA NUM_INT RCURLY SEMI

LITERAL_for LPAREN LITERAL_int IDENT ASSIGN NUM_INT SEMI IDENT LT
IDENT DOT IDENT SEMI IDENT INC RPAREN LCURLY

IDENT LBRACK IDENT RBRACK ASSIGN IDENT LBRACK IDENT RBRACK PLUS
NUM_INT SEMI

RCURLY
```

Figure 3: Lexical stream for first snippet of code from Figure 2

Note that in the above, all references to the Array `A` or the loop counter `i` are represented as `IDENT` tokens in the lexical stream. Because of this, both snippets have the exact same lexical stream. It should be noted that the lexical tokens also contain information that relates back to the source file such as what line the token came from and in the case of an `IDENT` token what the

underlying identifier (in this example `A`, `B`, `i` or `j`) is. However, much of the original program is contained just in the stream of token types. It should also be noted that changes to whitespace and comments in a source file have no effect on the lexical stream, they are not reported by the lexical analysis process.

Using the realization that changes to identifiers, whitespace and comments do not effect the lexical stream we can develop a simple algorithm for detecting L0–L2 plagiarism. First, construct the lexical stream for each program. Then, for each pair of programs, use Greedy String Tiling to measure the degree of similarity between two lexical streams. If this similarity is high, report as possible plagiarism. This method of plagiarism detection is presented by Wise in his paper introducing Greedy String Tiling as well by Gitchell and Tran[3] (who use a different string similarity algorithm). Wise also uses his algorithm in `YAP3` [9] a plagiarism detection system.

Detection via Parse Tree Similarities

One of the benefits of the working on the lexical level is that the lexical stream better reflects the "structure" of a program. The *parse tree* or derivation tree built from the lexical for a program also exhibits structure for the underlying program. Is it viable then to use the parse tree of a program to aid in plagiarism detection?

Introduction to Parse Trees

In part, programming languages are defined by their grammars, which describe the set of all possible strings that represent programs (called a *language*). During the compilation process, a compiler builds a parse tree which represents the program and uses this tree to guide compilation.

The parse tree for the code presented in Figure 2 follows:

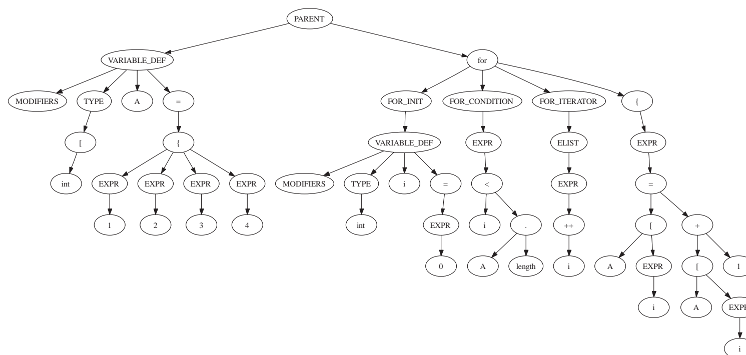

Figure 4: Parse tree for snippet of code

It should be clear that the parse tree for the second snippet of code will have the same structure (that is nodes and edges) as the first, since the lexical streams are identical.

Plagiarism and Binary Trees

Given the above, an algorithm for detecting plagiarism using parse tress could work as follows: First, parse each program. Next, for each pair of parse trees, attempt to find as many common subtrees as possible (perhaps setting a lower bound on the number of nodes two subtrees must share). Use this number as a measure of similarity between the two programs.

However, finding these common subtrees is not an easy task. Some algorithms exist which determine if one tree is isomorphic to one another under some mapping, but these algorithms are not useful for our task. Small changes to a parse tree would destroy parse tree isomorphism even if large subtrees were isomorphic. Furthermore, I was unable to construct or find a metric which measure the similarity of two binary tress operating on the binary trees themselves.

Similarity By Strings

Greedy String Tiling provides a way to measure the similarity between two strings. However, our parse trees are fundamentally different from strings. While a string can be thought of as a tree of words (each node except for the last has one child), the reverse is not true. However, classical tree traversal algorithms (for example, *pre-order traversal*) provide a coding of a tree as a string. Similarities in the parse tree should be reflected as similarities in the string generated by a traversal.

One such coding of a tree can be generated by visiting nodes in the tree using a pre-order traversal and at every node emit a token for the number of children of the node. Another such coding could produce a token representing the *depth* of the node. Each of these codings captures some of the structure of a parse tree.

Revised Algorithm

Using the observations above, an algorithm for detecting plagiarism on parse trees is as follows: First, parse each program. Next, for each pair of parse trees, convert each parse tree to a string using some coding method (in practice I have chosen to do a pre-order traversal of each tree and emit a token representing the number of children for each node). Using these strings, construct Greedy String Tiling to obtain a metric of similarity. Report this as the similarity between the two programs.

This is similar to the method proposed in [5], except that Jankowitz's method worked on static execution trees, not parse-trees and similarities in the trees were used to guide metric based measures of similarity. My method which works only

on the parse tree level. This distinction makes my method language agnostic, new metrics do not need to be developed for each language.

Extensions and Generalizations

There are a few interesting extensions and generalizations of the above algorithm. First, as hinted to above, any function which maps a parse tree to a string is a candidate for use in our algorithm. Adding more and more context to the token emitted for each node provides a better view of the structure of the parse tree and it follows that this has the possibility of making matches more meaningful.

Furthermore, It should be noted that the whole parse tree need not be converted to a string. An intermediate stage in our algorithm could transform the original parse tree into a "degenerate" parse-tree by removing nodes. For example, dropping the nodes dealing with the looping conditions is a `for` node, replacing the different types of looping constructs (`for`, `while`, `do`, etc) with a general loop construct node may provide better results. These techniques can be used to combat some common attacks. For example, in Java, local fields of can normally be accessed by simply `varname` instead of `this.varname`. These two statements have different parse trees. However a simple transformation can change one into the other which would thwart an attack where the `this` token was added or removed.

Finally, since it is possible to take a prase tree and emit a valid program from it or the lexical stream that generated the parse tree, previous methods for plagiarism detection can be viewed as a special case of this more general method.

Reporting Results

Each similarity detection method provides a measurement of the similarity of programs. Each method identifies one or more similar sections of two programs. Since all methods use Greedy String Tiling, for each program we have a count of tokens tiled (that is tokens that could be put in correspondence with one another) as well as the length of each tile. The ratio $\frac{|tile_i|}{|totalTokens|}$ represents ratio of tile length for a specific match to the number of tokens in the program. The weighted sum:

$$\sum_{i \in matches} \left(\frac{|tile_i|}{|totalTokens|} \right)^2$$

Gives a total measurement of how much of the program is copied. Note that this measurement is not linear. It has has the nice effect that longer runs of plagiarism have scores closer to 1, which represents a pair of programs that contains no differences.

For each detection method preformed a weighed score can be computed as show above, and then a linear combination of these scores (perhaps with some

tests given more weight than others) can be taken to give a final measurement of software similarity.

To present similarity, we use a method used by Lehenbauer and Young[6] which presents the similarities measurements of an assignment in upper triangular matrix form. Each cell represents the comparison of two assignments. Assignments with low similarity are colored green and ones with high similarity are colored red. Clicking on any cell reviles more information about the comparison between the two programs and allows an instructor to view the two source files in question.

An example of such a matrix follows. Note that darker regions indicate similar programs. This example has two pairs of programs the exhibit similarity.

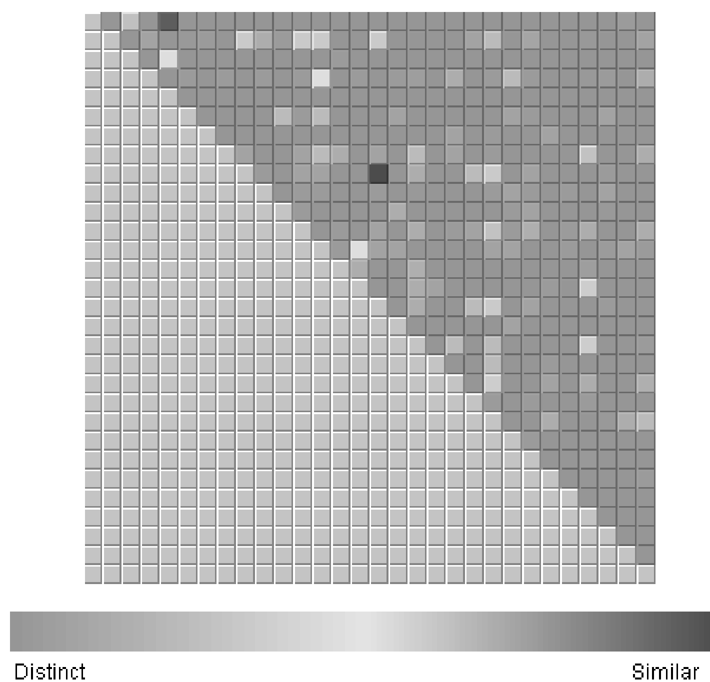

Figure 5: Example similarity matrix

Software

I have written software which uses the lexical and parse tree methods discussed above to detect plagiarism in student code. The software is simple to use, an instructor simply provides it with a directory containing a directory for each student. Each student directory contains the code that student submitted for the assignment in question.

The software is written in Java, using the ANTLR Parser generator to build a lexical analyzer and parser for the Java 5.0 Grammar. The implementation of the Greedy String Tiling algorithm is due to the PMD project.

The software is constructed to be language agnostic. Additional languages can be added to the program by simply including a new parser and lexer for each langauge an instructor wishes to add. Furthermore, I plan removing the coupling between the detection algorithm and the report generation algorithm so different methods of similarity reporting could be easily interchanged.

Finally, the software preforms quickly, analyzing the submissions of a class of roughly 50 students in a few seconds. While speed is not a necessity for this problem domain, it is a nice side effect. Finally, the software works well, detecting cases on known plagiarism.

Future Work

As discussed above, the implementation of my research could be improved by reducing coupling of unrelated parts of the program. We also wish to support the ability to provide additional submissions which are not compared to each other, but are compared to student submissions. For example, if an assignment is reused in subsequent offerings of a course, we don't wish to re-compare (and represent) programs from old offerings. It would also be nice to extend the software to include support for a few other languages out of the box (perhaps C and C++) which are common in other classes.

The extensions and generalizations sections discuss interesting manners in which this method of detection can be extended. Investigation into different functions that transform a parse tree into a string may help to improve robustness of the system. Furthermore, investigating the effect of "degenerate" parse trees on the performance of the system is an area for further study. Finally, examining how to combine scores from different detection methods may be beneficial.

One interesting transformation would be compiling the program either to bytecodes in the case of Java or machine instructions in the case of C and looking for similarities in the resulting output. Some preliminary work on detecting similarities via bytecodes is discussed in [1].

Additionally, an overall study of the robustness of the system including attempting to find straightforward transformations to the source which fool the detection algorithm would be useful in qualifying the quality of the tool as well and providing direction for studying ways to combat these attacks.

Finally, it may be useful to study the transformations done by an optimizing compiler to the parse tree for a program. It may be that many of the common changes made by a student wishing to disguise a program are "undone" by an optimizing compiler.

Conclusion

In this thesis, I have presented an overview of some common methods for detecting plagiarism in computer code. I have also introduced a new method based on program parse trees and shown how existing methods are a special case of my general method. I have also given the overview of software which implements these methods and discussed the ways in which it presents information about similarity to instructors. Finally, I have discussed some areas for further research related to my method of plagiarism detection.

References

[1] Brenda S. Baker and Udi Manber. Deducing similarities in java sources from bytecodes. In *Proceedings of Usenix Annual Technical Conference*, pages 179–190, June 1998.

[2] J. A. W. Faidhi and S. K. Robinson. An empirical approach for detecting program similarity and plagiarism within a university programming environment. *Computers & Education*, 11(1):11–19, Jan 1987.

[3] David Gitchell and Nicholas Tran. Sim: a utility for detecting similarity in computer programs. In *SIGCSE '99: The proceedings of the thirtieth SIGCSE technical symposium on Computer science education*, pages 266–270. ACM Press, 1999.

[4] Paul Heckel. A technique for isolating differences between files. *Communications of the ACM*, 21(4):264–268, 1978.

[5] H. T. Jankowitz. Detecting plagiarism in student pascal programs. *The Computer Journal*, 31(1):1–8, 1988.

[6] Daniel R. Lehenbauer and William R. Young. Directed research on plagiarism detection. Rose-Hulman Institute of Technology, 1999.

[7] Alan Parker and James O. Hamblen. Computer algorithms for plagiarism detection. *IEEE Transactions on Education*, 32(2):94–99, May 1989.

[8] Michael J. Wise. String similarity via greedy string tiling and running karp-rabin matching. `http://www.bio.cam.ac.uk/~mw263/ftp/doc/RKR_GST.ps`, December 1993.

[9] Michael J. Wise. YAP3: Improved detection of similarities in computer program and other texts. *SIGCSEB: SIGCSE Bulletin (ACM Special Interest Group on Computer Science Education)*, 28, 1996.