Detecting Plagiarism in Large Programming Classes

Tom Stokke
Computer Science Department
University of North Dakota
tstokke@cs.und.edu

Abstract

Plagiarism is a problem for all academic disciplines. Many businesses have been created to help find and deter the copying of work in a university setting. But programming presents special issues in determining the originality of work that make the methods used by other disciplines ineffective. While looking for common words or borrowed phrases may work in other disciplines it fails in programming.

The approach used here to determine the originality of a program is to compare the ideas put forth by the program rather than looking at the text itself. By breaking down each line(s) into its most basic concept, such as a *for loop*, *while loop*, *assignment* statement, or *print* statement we can eliminate the text and focus on the underlying ideas or concepts. By then comparing the logic and flow of the program an attempt to determine uniqueness can be performed.

Introduction

Plagiarism is a huge problem on college campus's everywhere. Borrowing a section of code for a computer program or a paragraph or two for a report is an all too common problem; a problem that has created a discipline of its own. A simple Google search will turn up dozens of businesses specializing in determining the likely originality of reports and papers. But programming presents some special issues in determining originality of the work that most plagiarism sites don't or can't address. While looking for common words or borrowed phrases may work in other disciplines it fails in programming.

All programs created in the same language work with the same basic set of commands. Even if using different languages there is still a limited vocabulary used by programming languages. Every program is literally guaranteed to have a duplicate amount of text strictly because of the limited command set. Variable names are one of the few pieces of text available to determine "uniqueness". But variable names are easily changed without changing the program. Several global search and replace commands can alter significant amounts of text in a program without the user having any idea what the program actually accomplishes. Indention and spacing schemes are also easy to alter, creating a very different visual look to a program without altering any of the logic behind the program. When dealing with large number of programs something truly unique or unusual, whether in the code or the output must occur for the similar programs to be noticed.

The approach taken to determine the originality of a program is to compare the ideas put forth by the program rather than looking at the text itself. I try to compare the logic and flow of control rather than only looking at variable names and commands. Breaking down each line into its most basic concept, such as a *for loop*, *while loop*, *assignment statement*, or *print statement* allows the program to eliminate the text and focus on the ideas. By then comparing the logic and flow of the program an attempt to determine uniqueness can be performed. A combination of several different techniques will be used to compare the "concepts" of the programs including assigning different point levels to different commands based upon their perceived importance, the length of runs of identical code and placing more importance on code that is nested.

It should be noted that the goal of this program is the help find the students that put little or no effort into the copying of an entire program. This program should avoid reporting of programs just because they have "some" similar code. It is to be expected that there will be small sections of code that will be similar if not the same in all programs. If code is given in class or the book it should be expected that it would be used in assignment. Do we not encourage students not to 'reinvent the wheel'? A whole other group of students is capable of writing the program but will spend more time altering someone else's code than the time it would have taken to write the program themselves. Those students are not the goal of this program, if for no other reason than it's unlikely that any program will be able to "catch" those students.

Source code preparation

The initial step was to create a file format independent of any programming language. This file format needs to represent the ideas and concepts put forth by the program while eliminating the details used to implement those ideas. The main obstacles to deal with include

- Various use of whitespace
- Various use of carriage returns
- Altering physical order of subroutines[1]
- Altering of variable and subroutine names

The output format needs be general enough to work with any programming language; Java, Visual Basic and Perl are three of the languages used at UND and all have their own idiosyncrasies. Perl has an implied starting spot in a program (the first line of code not actually in a declared subroutine), Java has a specified starting position (subroutine *main*) and most if not all Visual Basic programs turned in at UND have no specific starting position[2]; a completely event driven program. Some languages may require that subroutines be placed before the main code or that subroutines physically occur before being called (or have a forward declaration), others have no such requirements so there are numerous approaches that can be taken to alter the appearance of code that must be addressed.

The initial set of testing programs came from a Perl programming class. The only modifications to the original source code are the removal of any comments that could identify an individual student. It is assumed that all programs will at least compile and are capable of starting execution.

To remove any extraneous white space and carriage returns PerlTidy, a source code formatter for Perl programs, is used to speed up the process of consistently formatting the Perl programs before analysis. Among other options PerlTidy allows the programmer to dictate the location of opening and closing brackets, to remove all comments, to remove blank lines and to generate a consistent indention scheme.

The order of the main subroutine and/or any subroutines will not be addressed at this step of the program but will be addressed when two files are being compared.

To solve the issue of altered variable and subroutine names all identifiers will be removed. Only the basic command itself (such as *while*, *for*, *assignment*) will be retained. To determine the basic command each line of a program is tested against a set of regular expressions. Associated with each regular expression is a corresponding generic statement. A sample of the regular expressions and their associated commands is shown in Table 1. The use of a separate file will allow the program to work with any language and also lets the instructor determine what commands they are interested in and how specific they wish to be. Separate if statements can be created for an if statement with an equal's comparison, an if with a not equal comparison, and so on or just a simple if statement can be used without regard to the boolean expression. A generic call subroutine statement has been included for any commands that do not match other regular expression. As new material is introduced in class extra lines can be added to the regular expression file to recognize those new commands.

Table 1: Regular Expressions

Regular Expression	**Generic statement**	**Points**
^\s*my\s+	Declaration	5
^\s*\$\w+\s*(=|\+=|\-=|*=|\/=|**=)	Scalar Assignment	7
^\s*\$\w+\s*\[	Array Element Assignment	9
^\s*\$\w+\s*\{	Hash Key Assignment	9
^\s*print[\s|\"|\;]	Print	3
^\s*\{\s*$	Open Bracket	0
^\s*\}\s*$	Closing Bracket	0
^\s*foreach\s+	Foreach	17
^\s*sub\s+\w+	Subroutine Heading	15
^\s*for(\s|\()	For	15
^\s*if\s*\(	If	13

The entire program is read, analyzed and then written to a separate file. If the regular expressions are applied correctly both versions of source code from Table 2 generate the same generic code shown in Table 3.

Table 2: Functionally identical source code

```
@studentInformation = <STDIN>;
for ($ID = 0;
     $ID <= $#studentInformation;
     ++$ID)
{
      If ($studentInformation[$ID] >= 90)
            {
            ++$letterA;
            }
      elsif ($studentInformation[$ID] >= 80)
            {
            ++$letterB;
            }
      elsif ($studentInformation[$ID] >= 70)
            {
            ++$letterC;
            }
      elsif ($studentInformation[$ID] >= 60)
            {
            ++$letterD;
            }
      else
            {
            ++$letterF;
            }
       }
print $letterA, $letterB, $letterC, $letterD,
      $letterF;
```

```
@grades = <DATA>;
for ($sNo = 0; $sNo <= $#grades; ++$sNo) {
   if ($grades[$sNo] >= 90) {
      ++$A;
   } elsif ($grades[$sNo] >= 80) {
      ++$B;
   } elsif ($grades[$sNo] >= 70) {
      ++$C;
   } elsif ($grades[$sNo] >= 60) {
      ++$D;
   } else {
      ++$F;
   }
}
printf "%5d %5d %5d %5d %5d\n", $A, $B, $C,
$D, $F;
```

```
   Table 3: Desired output from
           both programs

Array Assignment
For
   If
      Scalar assignment
   Else if
      Scalar assignment
   Else if
      Scalar assignment
   Else if
      Scalar assignment
   Else
      Scalar assignment
Print
```

Criteria for determining likely originality – or lack of

To eliminate programs that don't require further examination by the instructor some form of metrics was need to compare the various programs. Each generic command in the regular expression file (Table 1) is given a value. The initial values are created by perceived importance and amount of thought needed to implement and/or alter. *Print* statements may be easily added to a program without worrying about affecting the programs logic but altering an *if* statement or a *for loop* requires more understanding of the program. *If* statements are deemed to be of more importance than *print* statements and therefore receive a higher point count. Less used statements receive a high point count because of their rarity of use. In Perl the *unless* statement receives more points than the *if* statement although it is functionally equivalent. Opening and closing brackets receive no points as they are just delimiters for the code itself.

Nesting of code creates complexity. An *if* statement placed inside an *if* statement inside a *for loop* is significantly more complex than a simple *for loop* followed by an *if* statement followed by an *if* statement. To account for this in the metrics any nested code receives the value assigned for that particular statement plus the value assigned to the nesting statement. So an *if* statement receives 13 points and a *print* statement that is the body of the *if* statement receives 16 "nesting points" (3 for the *print* and 13 for nesting within the *if* statement). This procedure is followed regardless of the level of nesting so a line of code nested several levels deep can generate a large nesting point value. Code within a subroutine is considered nested

A cumulative total is determined from the summation of the nesting points for each line. Given a couple of sample generic programs a breakdown of points, both per line and nesting points are shown in Table 4. After the summation of the nesting points is determined for the two programs the smaller total is divided by the larger total and converted to a percentage. Given the programs in Table 4 our results would indicate that there is a 92.26% (370 / 401) "likeliness percent "that the two programs are the same. Through testing, with the point values currently assigned for each command, a percent of 70% of higher indicates the need for further examination of the original source code.

Table 4: Analysis of Generic Code

Program1	**Points / Line**	**Nesting Points**		**Program3**	**Points / Line**	**Nesting Points**
Array Assignment	9	9		Array Assignment	9	9
For	15	15		Array Assignment	9	9
If	13	28		For	15	15
Scalar assignment	7	35		Print	3	18
Else if	17	32		If	13	28
Scalar assignment	7	39		Scalar assignment	7	35
Else if	17	32		Else if	17	32
Scalar assignment	7	39		Scalar assignment	7	39
Else if	17	32		Else if	17	32
Scalar assignment	7	39		Scalar assignment	7	39
Else	15	30		Else if	17	32
Scalar assignment	7	37		Scalar assignment	7	39
Print	3	3		Else if	17	32
				Scalar assignment	7	39
				Print	3	3
Overall Total Points		370				401

The next test was to actually compare the generic code of two programs. It's possible that two completely different programs, just through coincidence, can generate similar point totals. The Perl diff package was used to compare two generic files. The diff function computes 'intelligent' differences between two files / lists and is based on the *NIX diff command. More specifically diff computes the smallest set of additions and deletions necessary to turn the first sequence into the second, and returns a description of these changes[3]. By comparing the number and location of the differences against the original generic code a metric of copied code can be determined. Using the generic code from Table 4 diff returns the lines that need to be added or removed from Program1 to make it identical to Program3. diff will indicate that we need to insert 'Array Assignment' at line 2 and insert 'Print' statement at line 4. The two added lines have a total nesting point value of 27. By calculating a percentage of the common code over the entire program ((401 – 27) / 401) we come up with 93.26% likeliness that the two programs are the same.

To add a bit of efficiency only the main routines are initially tested. If the main programs are not sufficiently similar the testing of the subroutines is skipped. If the main programs display a sufficient degree of similarity then the subroutines are checked. Checking the subroutines pose several difficulties. Perl programs allow subroutines to physically appear either before or after they are invoked[4]. Visual Basic or Java subroutines can appear in random order. This allows the simple act of cutting and pasting of subroutines to drastically alter the look of a program without implementing any change at all. Analysis of the subroutines to match up corresponding subroutines will be required.

If the subroutines are tested each subroutine in the first program is compared to every subroutine in the second program using both total points and the difference percentage. The matching of subroutines is accomplished using a greedy algorithm that initially selects the best match available. Those two subroutines are removed from consideration and the process is repeated until all subroutines are paired. Any remained unmatched subroutines are ignored. Overall totals of total nesting points and differences for the main routines and all subroutines are calculated. If a likeliness percent of 70% or greater is calculated the two programs are reported as requiring further attention. Testing for the entire class requires an n^2 process of at least an initial comparison of each main program against every other main program.

Conclusion

This program was designed to find programs that require further scrutinizing by the instructor, not as a concrete statement regarding the uniqueness of a program. The goal is only to provide a means of sorting through a large number of programs and singling out programs that deserve further attention. It's essential that the instructor will still have the final say in determining the degree of similarity between any two programs. The current cut-off criterion of 70% seems to minimize the number of false positives while still finding the programs that require further attention.

Future work to be done includes

- Regular expressions for Visual Basic and Java
- A tokenizer for each implemented language to guarantee that whitespace/carriage returns cannot effect the quality of the process
- More analysis in determining the best point value for each command
- More efficient algorithm for matching subroutines
- Alternative methods of handling subroutines

Notes

1. For the purposes of this paper the term subroutine will be used generically for called named blocks of code regardless of the language. This includes subroutines, functions, value-returning methods and void methods.

2. Visual Basic programs can specify a subroutine to run at the start of the program execution but that is rarely, if ever used in student programs at UND.

3. Algorithm::Diff help file.

4. Perl functions can physically appear either before or after they are invoked but it may require a change of syntax depending upon the number of arguments, if any, passed to the function. Using prototyping in a subroutine may also affect the ability to move the subroutines.