A Teachers Tool for Detecting Possible Cheating in C++ Programming Classes

Justin Bearinger
Student
Valley City State University
Justin.Bearinger@vcsu.edu

Abstract:

Teachers have enough to do in order to just keep up with grading without the work of having to check student programs for cheating. So, I have written a pair of computer programs that can be used to aid the detection of cheating, if any, in C++ programming classes.

This paper describes and illustrates the difficulties encountered in trying to produce a set of distance correlations between computer programs. It states what my programs look for in determining the distance between two programs being compared. It also describes how the distances produced should be interpreted and what interpretation should be placed on the distances. Also provided in the paper are various test results from the testing process to the final product.

Introduction

Grading and evaluating programs is one part of a teacher's duties, another is to determine if students are copying from other's programs. It isn't reasonable to believe that students will never cheat. Some types of classes are inherently easier to catch cheating than others. However, it is difficult in programming classes to remember exactly what one source file looks like from start to finish when grading other programs.

I have created a set of C++ programs that work together to analyze other C/C++ programs for the purpose of attempting to catch cheating in programming classes. The first of the two programs simply scans the source code files of the programs to be compared and produces a vector of measures for a set of predetermined soured code features. The second program takes the vector of measures from the first program and computes the Euclidean distance between the programs being compared.

In order for the first program to scan the source files and create a set of measures there were a number of obstacles to overcome. The first and foremost was the parsing of the source code. Once the problem of parsing was overcome, I then needed to determine which features of the source code should be scrutinized for comparison. Besides determining which features to examine I needed to decide whether or not certain features were more important than others and if or how they should be weighted. Once the first program was completed, I wrote the program to evaluate the counts produced and give a set of distances for the programs that were scanned.

Parsing

The most difficult part of the project was parsing the source code. The full front end of a compiler was more than was needed. However, I had to have some way to break down the scanned code and create the set of counts for the source code features. So, I decided to write a function that would take a character string and turn it into individual words and characters (characters being semi-colons and other forms of punctuation). However, before calling this function, the string had all comments removed. I used a series of string position methods to create the individual words and characters. Once a word or character is extracted from the string it is placed onto a stack that is to be analyzed further into the function.

Some of the things to be counted are reserved words as well as certain punctuation. In order to keep track of these, every word and/or character in the source code needs to be checked. As words and characters are stripped from the string and before they are pushed onto the stack they are compared to see if they are one of the features to be counted. Once the initial stack is created and the reserved words that are easy to detect have been counted, the stack of words and characters is passed further into the function to be parsed by a series of nested **if** statements. The nested **if** statements check the code for things such as variable declarations, function declarations, assignment statements, etc.

The parsing is enabled by a text file containing all the C++ reserved words. This file also includes some of the Borland CBuilder system reserved words, since that is the main system in use on my campus. The file is read and stored into a hash table. The hash table was created specifically for comparing reserved words and their type. The reserved words need a type in order to differentiate them during the parsing procedure. I created three different types and assigned each reserved word one of three types.

The three types are:

- Type 1 reserved words are include the assignable types such as **int**, **double**, **bool**, etc…
- Type 2 reserved words include the scope block producing types such as **if, while**, **for**, etc...
- Type 0 reserved words include all other reserved words

While the function does find most of the variable declarations as well as the other things it is scanning for, it does not provide a perfect parse of the source code. A full blown parser would have been very difficult and time consuming to write and was unnecessary for this program. The emphasis is not to find out if the source code compiles, which should be determined by running the program during grading. Instead the main goal of this program is to accurately and consistently provide a set of measures for comparison between each program being analyzed. So, a perfect parse is not needed, only one that will treat every program the same.

Testing Scheme

Once the parsing problem was solved I needed to decide what features were important enough to record as well as whether or not my parser could handle scanning them. I came up with various source code features and luckily my parser needed only minor adjustments in order to check for all of them. After coming up with the set of features to examine I needed a way to test them.

In order to test the results produced by my programs, I needed an experimental set of programs. I used a common programming project in which four different C++ classes participated. Each class was numbered 1-4 for purposes of confidentiality (Hill 2004). I took the best program from each group as well as one that I wrote myself. Each program did the same thing, but each was from a different university, so there was little chance for plagiarizing between them.

I made a copy of each of these five programs, applying cheating tricks such as renaming variables, cutting and pasting code in different places, renaming functions, changing the comments, changing the indentation, as well as several other things. Once I created a copy of each, I then made a copy of each copy employing the same tricks resulting in a test group of fifteen programs. The goal was to obtain small distances between originals and copies and large distances for the rest.

I ran my programs on this set of programs several times using different source code features to process the distance computations. After many test runs on the test group I came up with a set of features that I felt gave the best distance results. However, the results produced by these did not seem to be quite as refined as I was hoping for. Upon further thought I decided it was necessary to apply a weight to the features, so that I could enforce which source code features were more important than others (the features measured are discussed in the next section). For example, I completely changed the indentation in one of the copies and it created a large distance between two programs I knew should have a close correlation. So, I decided to multiply the final indentation count by 0.1 since that count produced large values. However, I still feel that indentation should be part of the overall evaluation.

Finally, I came up with a set of thirteen features some of which are more important than others for determining the distances between programs. Although some features are not as differentiating as others I still consider them important enough to be measured. The following table illustrates the results of the final thirteen features.

Table 1: Test Group Final Results

LC #3 & LC #2 have the distance 0	**a9 #3 & a9 #1 have the distance 22.02**
LC #3 & LC #1 have the distance 1	**a9 #2 & a9 #1 have the distance 22.03**
LC #2 & LC #1 have the distance 1	**AJ #2 & AJ #1 have the distance 30.03**
a9 #3 & a9 #2 have the distance 2.1	**AJ #3 & AJ #1 have the distance 37.14**
JW #3 & JW #2 have the distance 4	**PG #3 & PG #1 have the distance 59.8**
PG #3 & PG #2 have the distance 12.19	**PG #2 & PG #1 have the distance 71.93**
JW #2 & JW #1 have the distance 12.69	**PG #2 & JW #1 have the distance 860.63**
AJ #3 & AJ #2 have the distance 15.4	**PG #2 & JW #2 have the distance 861.32**
JW #3 & JW #1 have the distance 16.52	**PG #2 & JW #3 have the distance 861.4**

Table 1 shows the results of the final source code features on the test group I created. Each programs' copies had variances in the way they were created in order to provide a wider range of difficulties for my programs to overcome. The results show the sudden jump from a distance of 71.93 to 860.63. All the results are displayed in ascending order. The jump from 71.93 to 860.63 implies that all previous programs are suspiciously closer than the rest. The last result of the group (not shown) is **LC #1 & a9 #1 have the distance 16447.2** which shows just how much difference there can be between this particular set of programs.

Source Code Features Measured

The first part of the thirteen features consists of counts of C++ reserved words. The other part of this set is specific to the code usage such as *declarations* and *assignments*.

The Thirteen Features are:

1) A count of every time the reserved word **int** is used
2) A count of every time the reserved word **double** is used
3) A count of every time the reserved word **bool** is used
4) A count of every time the reserved word **while** is used
5) A count of every time the reserved word **for** is used
6) A count of every time the reserved word **if** is used
7) A combined count of all other reserved words used
8) A count of semi-colons, gives an idea of how many executable statements
9) A count of assignment statements
10) A count of variable declarations
11) A count of the total amount of comments
12) A count of the various compound statements within the entire source code file
13) A count of the total indentation throughout the source file

The counts are stored inside a vector that is subscripted using an enumeration. As stated previously, the goal is not perfect counts but rather consistent counts. Every program's counts are produced in exactly the same way. Once all the source code files for a program have been scanned and the counts have been generated weights are applied and appended to a text file. Once the information for a program has been written to a file the next program for comparison can be scanned and evaluated. The table below shows the method used for applying the weights.

Table 2: Weight Applying and Enumeration

```
enum
{Int,Dubl,Bool,While,For,Rest,Semi,Assignmnt,Decl,Comment,Blocks,Indent,If};

  vec_counts[While] = vec_counts[While] * 5;
  vec_counts[For] = vec_counts[For] * 5;
  vec_counts[If] = vec_counts[If] * 5;
  vec_counts[Indent] = vec_counts[Indent] * 0.1;
```

The table shows two pieces of code directly out of my program. The top line shows the enumeration I created of the thirteen features. Instead of trying to remember which number I used as a subscript inside the vector (vec_counts) for each source code feature, I can simply refer to them by name. I decided that the **for**, **while**, and **if** features were more important than the others and the indentation feature should have less importance than the others.

Directory Processing

Once I had finished my programs I needed a fast and easy way to read in multiple programs for comparison, a single program can have several source code files in it and there can be many student programs in a single class. Therefore, executing the program manually for many programs is undesirable. In order to achieve the goal of analyzing several programs fairly effortlessly, I needed a way to process an entire directory of folders each containing a program. Fortunately I was able to acquire a substantial donation in this particular matter. My advisor, Curt Hill, had already written a directory processing program for his own purposes. Professor Hill allowed me to take a copy of his directory processing program and manipulate it to work with my program. After the initial manipulation of the directory program was done, I cut and pasted the code from my program into the appropriate parts of the directory program.

Once my code had been embedded into the directory program, I was able to process an entire class of student programs very quickly and with little effort. In order for the newly modified directory program to work, a grader simply creates a directory tree. The directory tree for this is created by starting with an initial folder which in itself contains a folder for each student which should be named appropriately. The name of the directory should indicate the student. Inside each student directory should contain all the files a student turns in for a particular programming assignment.

After a directory has been created, the executable file for the directory program can be started. Once the executable has started, simply open the directory to start in and it scans every subfolder looking only for the source code files and then creates the source code feature counts for each program. Once the executable is done running a file named Cheat_counts.txt containing all the counts can be found on the C: drive.

Distance Calculations

Once the Cheat_counts.txt file has been created it is time for the second program to calculate the distances between all of the programs. The Cheat_counts.txt file is simply the text representation of several vectors containing the various counts obtained by the directory scanning program. A vector of counts is created for each student program and is represented on its own line in the Cheat_counts.txt file. The distance calculating program reads the text file in and puts each of the vectors into another vector creating a two dimensional vector.

Once the two dimensional vector is complete the program can calculate the Euclidean distances between any two programs. Every student program is represented by a vector or "point" inside the containing two dimensional vector. Each student program vector is subtracted from another student's vector. Then each difference inside the vector is squared and all the squares are added. Finally, the distance program takes the square root of the sum of all the squares to determine a distance between the two vectors containing each students set of counts.

There is no set distance that can be declared as an indication of cheating. This is because there are too many variances between program assignments. The major variance is the size of the programs being compared. Very large programs with many lines of code will inherently have greater distance results than shorter simpler programs. The idea behind my distance evaluation programs is to show the distance between all the programs scanned. If there are programs that have a much shorter distance correlation than the rest of the programs then it is safe to assume that those programs should be reviewed more closely. However, it does not imply that the programs are examples of plagiarizing, it just shows that they are abnormally closer than the rest. The following table shows an example of distance correlations from university #4 (VCSU) for the common programming project.

Table3: Results from University #4's Programming Class.
Program 7 & Program 3 have the distance 7.3
Program 6 & Program 5 have the distance 20.84
Program 6 & Program 1 have the distance 408.32
Program 5 & Program 1 have the distance 408.69
Program 6 & Program 2 have the distance 410.23
Program 5 & Program 2 have the distance 411.9
Program 4 & Program 1 have the distance 605.89
Program 2 & Program 1 have the distance 816.94
Program 8 & Program 7 have the distance 852.88

Table 3 illustrates two sets of programs that have an abnormally close distance when compared to the rest. All distances are displayed in ascending order by the program. The abnormality of the first two sets of distances and the third closest distance of 408.32 suggest that the first two sets are cheats. After manual inspection the first two sets did appear to be cheats. In fact, one of the authors for the second pair, with the distance of 20.84, had been suspected of plagiarism on other assignments by the class instructor.

The following table shows an example of distance correlations that are not as obvious as the results in table 3.

Table 4: Results from University #3's Programming Class.
Prgrm 37 & Prgrm 23 have the distance 0
Prgrm 40 & Prgrm 28 have the distance 1.35
Prgrm 8 & Prgrm 19 have the distance 6.5
Prgrm 45 & Prgrm 37 have the distance 7
Prgrm 45 & Prgrm 23 have the distance 7
Prgrm 38 & Prgrm 26 have the distance 26.51
Prgrm 35 & Prgrm 18 have the distance 29.53
Prgrm 14 & Prgrm 11 have the distance 29.94
Prgrm 32 & Prgrm 9 have the distance 31.76
Prgrm 33 & Prgrm 12 have the distance 38.45
Prgrm 46 & Prgrm 25 have the distance 44.89
Prgrm 35 & Prgrm 20 have the distance 44.93
Prgrm 21 & Prgrm 11 have the distance 48.07

Table 4 illustrates the difficulty in deciphering distance correlations within program variations. The initial jump isn't as obvious as table 3. However, the jump from 7 to 26.51 is slightly more intriguing than the other distance differences. The initial jump seems to be less gradual than others, implying only that the first five pairs should definitely be looked at closer and maybe a couple of the others. The rest of the distance results (most not shown) gradually become larger at the same rate as the ones shown. Upon closer manual inspection the first pair proved to be identical programs, even the comments were the same. The next four pairs only had differences involving commenting. The rest of the programs, under close manual inspection, appeared to be legitimate, original programs. The close correlations of all the programs come from the class using a specific set of functions and layout.

Once in a while two programs can have source code feature counts that are similar enough to create a close distance correlation and yet not be cheat programs. The following two tables show examples of how occasional results can be misleading.

Table 5: Results from University #2's Programming Class.
SM & JA have the distance 6.48
MD & HF have the distance 13.96
CG & BC have the distance 20.3
RM & BC have the distance 200.5
TP & BI have the distance 200.91
RM & CG have the distance 204.15

Table 5 shows three sets of programs that have an abnormally closer distance than the rest. However, after manually inspecting the programs in table 5 only the first two sets appeared to be cheats. The third set, with a distance of 20.3, was difficult to decipher manually because the two programs seemed to follow a similar but not identical style part way through. However, one of the programs was incomplete and the code for it seemed to end prematurely, leading to the conclusion that they are not examples of plagiarism.

Table 6: Results from University #1's Programming Class.
a15 & a12 have the distance 1
a14 & a11 have the distance 13.3
a6 & a10 have the distance 17.8
a8 & a2 have the distance 25.2
a7 & a10 have the distance 256.55
a7 & a6 have the distance 256.58
a13 & a11 have the distance 515.73

Table 6 shows an abnormality between four sets of programs. After closer manual inspection the first three sets of programs did appear to be examples of plagiarism. However, the fourth distance of 25.2 under manual inspection showed that the two programs seemed to be legitimate non cheating programs. After inspecting the counts for programs a8 and a2 the reserved word counts for the programs were very close. Normally the counts for reserved words between non cheating programs are quite different due to the way that particular count is produced. Two programs with minimal usage of reserved words could produce a misleading distance correlation.

The following figures display the cheating information from all the programs scanned.

Figure 1: Cheat Counts

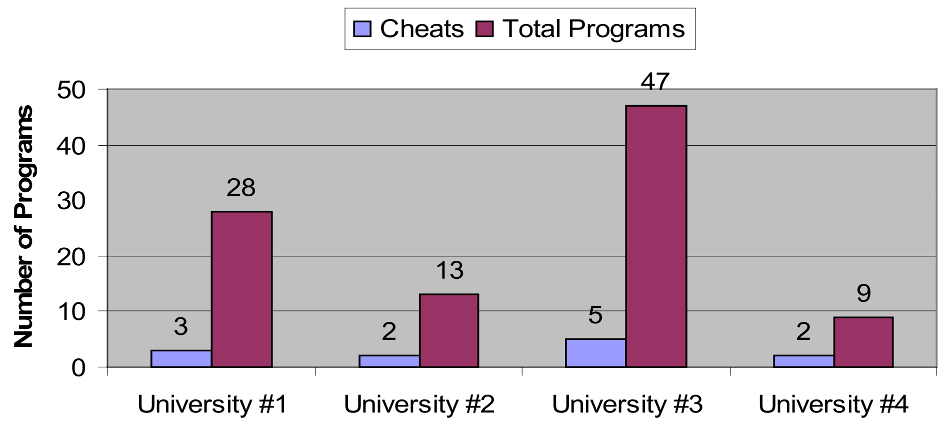

Figure 1 illustrates the total number of programs for each university class and how many cheats were found in each class.

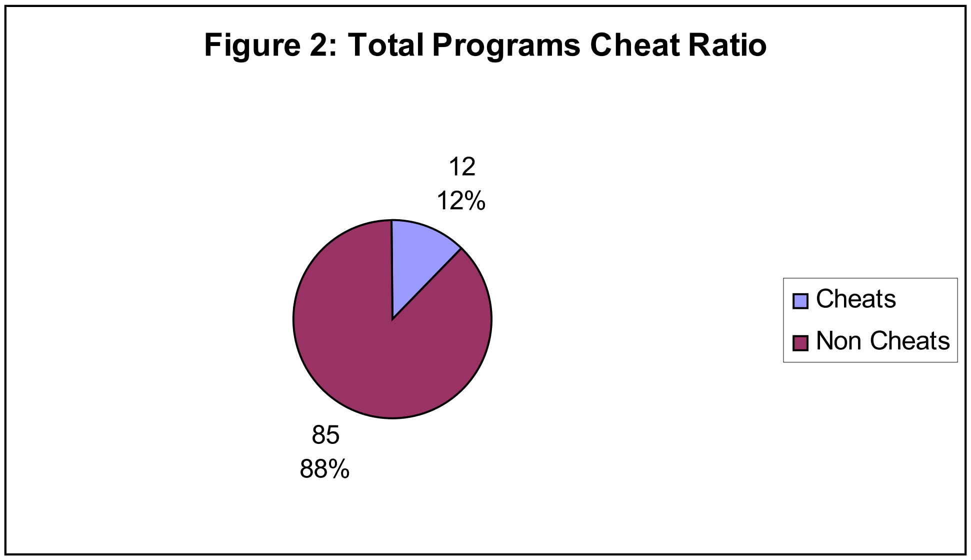

Figure 2 illustrates the total number of cheats and non cheats for all four universities (Hill 2004).

Conclusion

While cheating in programming classes remains difficult to catch I believe that my programs can be used to remove some of the time consuming, manual analyzing out of the process. Though my programs are not designed to absolutely pinpoint cheating it does give an excellent idea of where particular attention should be placed. My programs were written to evaluate C/C++ programs, Borland programs in particular. Borland source code files have the extension .cpp which is what the directory program looks for. However, the only things that would need to be changed are the search for another source code extension (instead of .cpp) and a few minor changes to the reserved words text file used.

Given the similarities between C/C++ source code and Java source code it is conceivable that my programs could also be converted to perform their tasks on Java programs. The simple syntax that my parsing function checks for in C/C++ code is nearly identical to that of Java code. Again I conclude that the only things that would need to be changed are the source code extension search and the reserved words text file.

Acknowledgements

I would like to acknowledge the assistance of Curt Hill in the creation of the finer details within the distance generating program as well as the contribution of his directory processing program.

References

Hill, Curt, Brian Slator, and Lisa Daniels (2004). Using and Validating Programming Land. Submitted for publication at Computers and Advanced Technology in Education. (CATE 04). www.iasted.org/conferences/2004/hawaii/cate.htm.