Embedding Copy Detection Within an Automated Submission System For Programming Assignments

Gordon Lingard
University of Technology, Sydney

Abstract To address the problem of computer programming students copying and colluding on assignments, since 2003 a system has been developed and utilised at the University of Technology, Sydney that embeds copy detection information within the logs of a submission system used by the students to submit assignments. This creates a detailed audit trail that allows for the determination of who has created and who has copied code. Beyond that, the information can be mined to see how student networks form to exchange information.

Key Ideas

- Collusion detection – programming assignments
- Combining collusion detection with assignment submission system
- Studying the way collusion networks form. How information is exchanged between students.

Discussion Question 1 The network detection systems were deployed in a third semester programming subject. There is evidence that many students are colluding from the time they begin their degree to complete assignments. By the time they reach their third semester they are so far behind that they have little option but to continue cheating if they are to pass the subject. What can be done to discourage students from doing this at the very beginning of their degree?

Discussion Question 2 Over the last couple of years there been a marked shift in the way students cheat. Instead of copying off each other they are now increasingly using many of the rent-a-coder sites to get their assignments done. For the students cheating the advantage of this is its cheap, easy to use and almost impossible to detect, let alone prove. This is proving to be a very corrupting temptation. What can be done about this?

1 Introduction

Collusion and copying between students doing computer programming assignments has been a significant problem for many years [Dick, M., J. Sheard, et al.]. In an effort to combat this many plagiarism systems have been created that have worked reasonably well to detect copies [Verco, K. L. and M. J. Wise]. However, a problem with such systems is that they only identify assignments that are copies of each other. They cannot tell who has originated the code and who has copied.

This work originally started in 2000 when the author designed and implemented a software system to detect copying between programming assignments. While very successful in finding copies, it was little help in working out if one of the assignments was a copy of the other or whether there had been active collusion between the students to produce the same result. Additionally, if one was a copy of the other it was very difficult to tell who had originated the assignment and who had copied it.

4th Asia Pacific Conference on Educational Integrity (4APCEI) 28–30 September 2009
University of Wollongong NSW Australia
Refereed Paper

In 2003 the core engine of the detection system was included within an online submission system the author had created. The engine created a kind of digital fingerprint of a student's assignment and stored it in a log, along with date and time information. By comparing the logs of two students it was possible to make strong inferences as to who had created the assignment and who had copied it and when they had copied it. Alternatively, it could also give strong indications if the assignments had been jointly created by the students.

The system proved more useful and powerful than originally intended as data mining of the logs gave some information on the way students networked to solve issues with each assignment.

1. This paper is organised in following sections.
2. Introduction
3. Outlines the original copy detection system and how it works.
4. Describes the submission system and its functionality.
5. Details how the core algorithms of the copy detection system were incorporated into the submission system logs.
6. Examines how the information stored in the logs can be mined to examine how fragments of information are passed between the students. We can examine the way students network to collude.
7. Ends with some conclusions and further considerations.

2 The Copy Detection System

Copying between programming assignments submitted by students is difficult to detect for two reasons;

1. The issue of *disguise*. Transformations are applied to a copied program so as to make it not obvious it is a copy. It doesn't take many transformations to make a program look quite different from another and yet be structurally exactly the same and perform in exactly the same manner. Additionally, it takes very little knowledge to apply these transformations.

2. The issue of *combinatorics*. Given there are *n* assignments then the number of possible pairs *p* is $p = n(n-1)/2$. In other words, the number of pairs goes up as an order of the square of the number of assignments. For example, given 100 assignments this creates 4950 possible pairs to check. For 300 assignments the number is 44850. Even worse, if marking is spread across a number of markers then it is quite possible that a copied program will be marked by a different marker and thus be entirely undetected.

These issues of disguise and combinatorics means it rapidly becomes infeasible to check all possible assignments pairs for copying and have any real hope of detecting any of them, let alone all of them. The implication of this is that it is very easy for a student to copy another student's program and have very little chance of being caught. These factors have driven the development of software based detection systems [Faidhi, J. A. W. and S. K. Robinson].

The copy system was originally devised to tackle the problem and worked in the following way

1. The program code is read, statement by statement, and all operators used in each statement are extracted.

   Once all the operators have been extracted from a statement, a complexity metric is applied to the collected operators. This metric is designed to generate a *complexity value* representing how complex and convoluted the logic of the statement is. The larger the value the more complex the statement. The value is weighted by the following factors

   - By how deeply array operator and parentheses are nested.
   - By the use of more difficult operators such as pointer operators.
   - By the number of different operators used.

2. In addition, a *checksum* is generated for the operators. This is a value in the range 0 to 255.

3. The complexity value is multiplied by 1000 and added to the checksum. The result is called a *complexity number*. Thus, the first 3 low order digits of the number are the checksum while the higher order digits represent the complexity value. The checksum helps ensure that each statement generates a unique complexity number. Figure 1 illustrates the result.

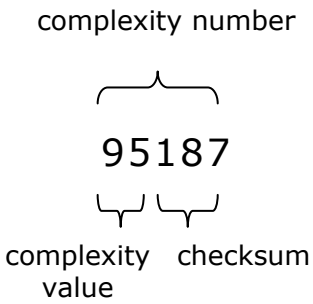

Figure 1 - Break up of Complexity Number

   The result is that a program made up of a number of code statements is reduced to an equivalent number of complexity numbers. The important aspect of the complexity numbers is that they are independent of such things as comments and variable names used in the code. These are the things students typically change in order to disguise the copied code.

4. Given two programs and their associated set of complexity numbers, a measure is made of what percentage of complexity numbers is in common. The closer the percentage overlap is to 100% the more likely the two programs are copies.

Figure 2 show a graph of 148 student assignments compared to each other. This gives 10878 possible pairs of assignments to check. The *Y* axis of the graph shows the number of pairs in log scale while the *X* axis groups the number of assignments pairs falling into percentage blocks of 5%.

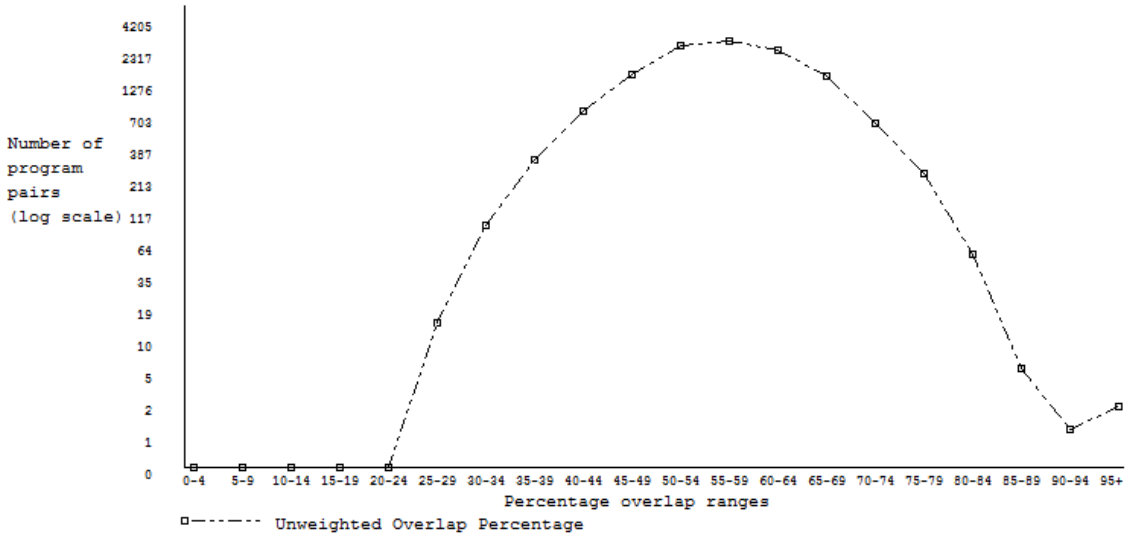

Figure 2 – Percentage of Complexity Numbers overlapping between every assignment pair

By only choosing those assignment pairs falling into the higher percentile brackets, the system allows the user to quickly trim the number of assignment pairs down to a manageable number for individual checking.

This method of looking for copies has these advantages.

1. It is difficult for students to disguise their assignment. Disguise methods, such as changing variable and function names or swapping function placement, are completely defeated.

2. The system is reasonably fast. For example, running the program on 300 assignments takes about 10 seconds, and most of this time is spent extracting the complexity numbers rather than doing the comparisons.

3 The Submission System

The submission system was a set of programs the author had put together over a number of years to handle student assignments. It was designed to run under UNIX as a command line facility. It incorporated an extensive configuration file which meant it was a very flexible tool and could call any other programs to aid in processing the assignments.

The submission system was designed to accept coding assignments, test if they compiled and then run a suite of tests against the compiled code. Additionally, a number of other programs were run to analyse the source code to look for poor programming practices. The results from the tests and the analysis became part of the final mark for the assignment. The students were given immediate feedback about the functionality of their programs and they could re-submit as many times as they liked till the due date. Because students could submit repeatedly, they used the submission system as a kind of development environment.

This was a main consideration when the submission system was developed. That is, to create a formative assessment tool that the students could use to gain feedback and learn from their mistakes without penalty. While a separate issue

from the focus of this paper, the quality of student assignments considerably improved due to the use of these tools.

Additionally, every time a student submitted their assignment they would receive a receipt in the form of a file containing date and time, a hashed checksum of the assignment and a hashed checksum of the entire receipt. This made the receipts almost impossible to forge and constituted proof on their part that they had submitted their assignment. Any change to the receipt or file would make the checksums incorrect thus making the receipt invalid.

The system didn't store every submission the students made as they would often resubmit over a hundred times as they refined their assignment in response to the feedback and testing programs. Given a couple of hundred students this would require a fairly significant amount of memory. It would store the last 3 submissions received. This would later change with addition of the copy detection logs.

4 Embedding the Copy System within the Submission System

In the final semester of 2002 there was a case where two assignments were clearly identical. There was no chance this could have been coincidental. Nevertheless, each student was able to present a convincing case to the Faculty that they had created the assignment by themself. One was misrepresenting themself with a very high level of sophistication and it became increasingly clear that more evidence was needed to prosecute cases of cheating.

In response, elements of the copy system were incorporated within the submission system. These elements were used to create a digital fingerprint of an assignment.

4.1 Digital Fingerprints

Given that the original copy detection system reduced an assignment to a set of complexity numbers, the 6 largest unique numbers were used to create a digital fingerprint of the code. These numbers nominally represent the most complex parts of the code. As such they are usually unique to any piece of code and are probably the hardest for a student to change without a detailed understanding of how they function. As students who cheat usually lack this knowledge they tend to be left alone. This means the fingerprint is usually unique to a piece of code while being resistant to minor changes to it.

The digital fingerprints allowed the implementation of another useful feature in the submission system. Since a change in any of the complexity numbers in the fingerprint meant a significant change in the code, any submission that generated a new fingerprint that had not been previously seen from the students would be stored. The submission system was now storing the last three submissions and every major revision to the code.

4.2 Submission Logs

Every time a student submitted their assignment, the system would call a small program that would generate the digital fingerprint of their code and store it in a log file along with time and date.

Figure 3 illustrates the steps of the submission process and adding fingerprints to the logs

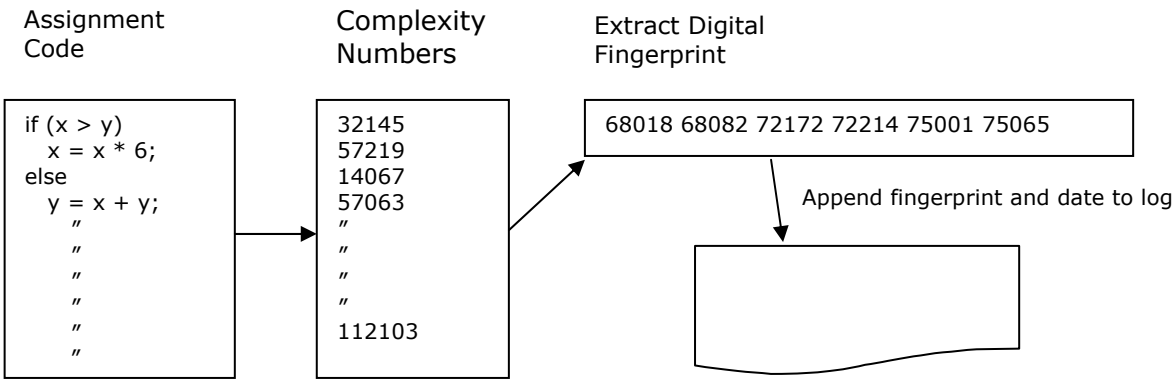

Figure 3 – Steps to adding fingerprints to logs

Table 1 is a partial listing of an actual submission log file for a student.

Sat May 24 02:30:55 assign2.c studentx, bytes = 2156 68018 68082 72172 72214 75001 75065	Fri May 30 11:53:46 assign4.c studentx, bytes = 3065 68082 72172 72214 75001 75065 205069
Sat May 24 02:34:54 assign2.c studentx, bytes = 2156 68018 68082 72172 72214 75001 75065	Fri May 30 11:59:22 assign4.c studentx, bytes = 3065 68082 72172 72214 75001 75065 205069
Mon May 26 12:18:11 assign2.c studentx, bytes = 2166 68082 72172 72214 75001 75065 115028	Fri May 30 12:06:58 assign4.c studentx, bytes = 3065 68082 72172 72214 75001 75065 205069
Mon May 26 12:20:32 assign2.c studentx, bytes = 2166 68082 72172 72214 75001 75065 115028	Fri May 30 12:15:36 assign4.c studentx, bytes = 3009 68082 72172 72214 75001 75065 205069
Mon May 26 17:06:31 assign2.c studentx, bytes = 2186 68082 72172 72214 75001 75065 115028	Fri May 30 12:54:46 assign4.c studentx, bytes = 3009 68082 72172 72214 75001 75065 205069
Tue May 27 16:35:56 assign2.c studentx, bytes = 2221 68082 72172 72214 75001 75065 115028	Fri May 30 20:06:13 assign7.c studentx, bytes = 3155 72172 72214 75001 75065 84253 205069
Thu May 29 23:15:49 assign3.c studentx, bytes = 3421 75065 85058 115028 115048 140160 205069	Fri May 30 23:36:03 assign8.c studentx, bytes = 3962 72214 75001 75065 84253 150027 205069
Fri May 30 11:49:08 assign4.c studentx, bytes = 3068 68082 72172 72214 75001 75065 205069	Sat May 31 00:04:49 assign8.c studentx, bytes = 3969 72214 75001 75065 84253 150027 205069
Fri May 30 11:50:27 assign4.c studentx, bytes = 3065 68082 72172 72214 75001 75065 205069	

Table 1 – Log of submissions by an individual student for their assignment

Each log entry consists of 3 lines.

1. Date and time
2. Assignment file name, student login, size of assignment file
3. Digital fingerprint

Each log entry also contained other information, such as whether the assignment compiled or not, how many tests were passed, what the feedback results were. These have been left out of Table 1 for clarity.

Going through the log file is a fascinating exercise. On Monday, May26 studentx made a significant incremental change to their code. Again this happened on Thursday May 29 and Friday May 30. We see a clear trail of studentx developing their code till they reached their final submission. A student who is copying the work of another will not have this sort of audit trail. From this it would be possible to strongly infer who created the assignment and who copied it. Actual use of the logs bore this out.

An important point to note is the fact that students could repeatedly submit assignments. The testing infrastructure associated with the submission system practically guaranteed that students would submit many times as they developed and refined their assignments. If the submission system was a once only system then there would be no audit trail of development.

4.3 Comparing Two Sets of Logs

There is a lot of detail in each log file. This can be simplified down by only looking at when the digital fingerprints change. Table 2 shows the major digital fingerprints changes of two students listed side by side in ascending date order.

date summary	StudX	StudY
Fri Oct 28 17:42:24	56053 60020 60193 65350 70426 96435	
Fri Oct 28 19:09:52	55378 56053 60020 60193 65350 70426	
Fri Oct 28 22:05:52	56053 60020 60193 65350 70426 96435	
Mon Oct 31 18:50:24	70426 72017 82024 85265 96435 182097	
Mon Oct 31 18:57:42	65350 70426 72017 85265 96435 182097	
Tue Nov 1 00:12:05	60020 60193 65350 70426 72017 96435	
Tue Nov 1 01:15:38	56053 60020 60193 65350 70426 96435	
Tue Nov 1 14:04:30	56053 60020 60193 65350 70426 96435	
Fri Nov 4 13:47:36		27033 28182 36020 45423 45491 55378
Fri Nov 4 13:49:18		26223 27033 28182 45423 45491 55378
Fri Nov 4 14:22:32		36060 36183 45423 50482 60426 84235
Fri Nov 4 14:23:38		36183 45423 50482 55378 60426 84235
Fri Nov 4 14:25:18		36183 45423 50482 55378 70426 84235
Fri Nov 4 14:42:29		56053 60020 60193 65350 70426 84235
Fri Nov 4 15:15:53		56053 60020 60193 65350 70426 96435
Fri Nov 4 15:16:12		60020 60193 65350 70426 84235 96435
Fri Nov 4 16:42:48		60020 60193 65350 70426 84235 96435

Table 2 – Comparison of two students logs showing date and time of major code revisions

StudX has developed their code well before StudY and have finalised their submissions on November 1. We can also see they have taken five days to incrementally develop and complete the assignment. On the other hand, StudY does not start submitting till three days after StudX finishes. The digital fingerprints are quite different to StudX until 14:42 when 4 of the 6 numbers abruptly change to numbers that have been generated by StudX. In fact, the submission of StudY at 15:15 has exactly the same digital fingerprint as the final

submission by StudX. Investigation of the code subsequently showed the submissions were copies and the information provided by the logs made a compelling case as to who had originated the code and who had copied it.

In addition, by storing each submission that generated a new digital fingerprint it was easy to go back to the code submission by StudY at 14.42 and see exactly what they had done.

4.4 History of Using the Plagiary Logs

The new system was tested in the first semester of 2003. The subject had 320 students enrolled with two programming assignments to complete. In the first assignment the copy detection program found a total of 13 students involved in copying while in the second assignment a further 12 were found. The logs proved invaluable in presenting misconduct cases as to who had originated assignments and who had copied.

Since then, the inclusion of the logs and their analysis has put the Faculty in a much more solid position when making determinations of misconduct cases. Furthermore, given the weight of evidence the logs create, most students admit their guilt immediately and are far less inclined to appeal the results.

5 Collusion Networks

The logs were initially used to provide corroborative evidence in cases of copying. However, they can be mined for further information, in particular searching for complexity numbers that appear to travel between groups of students. In other words, examining the logs can be used to determine if there are low levels of collusion between groups of students. This would be in the form of networks of colluding students.

The networks are constructed by going through the submission logs and comparing every student log to every other student log. The idea is to look for the total number of complexity numbers that are shared between each set of students. The larger the number then the more likely there has been collusion.

A number of factors need to be taken into account to determine if we have a genuine network of information being passed around.

- The amount of information (in complexity numbers) shared between students. The more shared numbers the more likely this isn't coincidence.
- The quality of the numbers, in terms of how well the submitted program performs. Numbers associated with programs that are fully working tend to be larger than those from programs that do not work at all.
- The commonness of the numbers. Do they represent material found in lecture notes, etc or do they represent uniquely created pieces of code that have been passed around? To this end, larger numbers will more likely represent specialised code rather than generic code taken from the lecture notes. There needs to be a cut off factor if numbers are used by too many students as they will probably represent code from the lecture notes.
- What weighting should be given to groups of numbers that transfer all at once as opposed to numbers that dribble across from one student to another student one at a time? Groups of numbers that cross from one student to

another will more likely be due to collusion rather than random chance.

Tweaking these parameters can produce endless results. Table 3 is an example extracted from the logs of 144 students who made submissions. The extracted data used the following criteria on exchanged numbers

1. No more than 15% of the students used the number. This was the cutoff percentage chosen. Any higher than this and it was assumed the numbers were generated from code found in the lecture notes

2. The number had to be associated with code that scored at least 5 out of 8 in the tests. Again, this meant the numbers were associated with code that was mostly working rather than code that wasn't.

3. There must be at least 4 numbers exchanged between the students. Such a large set of numbers would indicate a real level of collusion rather than a coincidental development of the same single piece of code.

Subnet 1	stud10 0	Subnet 8	stud23 3
stud01 4	stud09 2	stud19 5	stud25 5
stud02 0	stud11 4	stud20 0	stud27 0
stud02 4	stud10 0	stud21 0	stud27 0
stud03 0	stud11 5	stud20 4	stud28 4
Subnet 2	Subnet 5	stud21 0	stud27 0
		stud22 8	stud29 4
stud04 0	stud12 4		
stud05 5	stud13 0	stud21 0	stud29 4
		stud23 5	stud30 0
Subnet 3	Subnet 6		
		stud24 4	Subnet 9
stud06 0	stud14 0	stud25 0	
stud07 7	stud15 5		stud31 4
		stud25 1	stud32 0
stud06 0	stud14 0	stud20 4	
stud08 6	stud16 6		stud33 4
		stud20 4	stud32 0
stud07 0	Subnet 7	stud23 1	
stud08 6			stud34 4
	stud17 5	stud23 0	stud32 0
Subnet 4	stud18 0	stud26 6	
			stud35 4
stud09 4		stud25 2	stud32 0

Table 3 – Groups of complexity numbers exchanged between students

Each set of students lists the number of complexity numbers that each student appeared to receive from the other student. For example, in Subnet 1, stud01 4 and stud02 0 indicates that 4 numbers (and the code they represent) appeared to transfer from stud02 to stud01 but none went in the other direction.

There appear to be 9 groups of students sharing, with one subnet (subnet 8) containing 12 students while the rest are much smaller in size. The results are displayed diagrammatically in Figure 4.

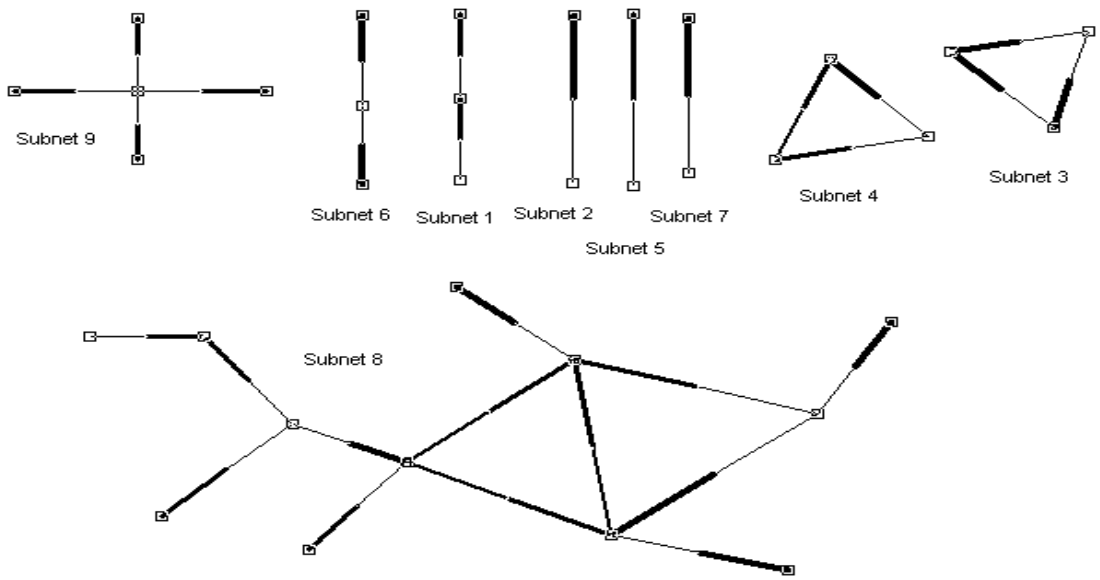

Figure 4 – Display of collusion networks

In the figure each square node represents a student and each link between nodes represents complexity numbers that appear to have passed from one student to another. The links are split into two halves. The half that is closest to each student represents the number of complexity numbers that student appeared to receive. A line of thickness 1 means they received no numbers. A line of thickness 2 means they received either 1 or 2 numbers. Thickness 3 means 3 or 4 numbers and thickness 4 means 5 or more numbers. In other words, the lines indicate the direction of flow of information. For example, in subnet 1 we can see information went from stud03 to stud02 and from stud02 to stud01.

While a visually powerful tool, the network diagram of Figure 4 does have a number of limitations that must be understood.

1. The diagram shows that numbers are moving from student to student. It doesn't show which numbers though.
2. The diagram is static and doesn't show how the networks dynamically evolve over time.
3. A more fundamental problem is that certain links that the log analysis generates are almost certainly false. The subnet 3 in Figure 4 illustrates this nicely. The values for the subnet are

stud06	0	stud06	0
stud07	7	stud08	6

stud07	0	stud08	6

   Assume that six of the numbers that stud07 and stud08 receive are the same. The problem is we don't know if stud08 received these numbers directly from stud06 or indirectly via stud07. This means that one of the links, 6→8 or 7→8, is probably not real and there is no way to determine this from analysing the logs. This issue would have significant impact on the actual shape of larger networks such as subnet 8.

Nevertheless, given these limitations, the networks that appear are fascinating to observe. Invariably they are small, involving 2 to 3 students, but larger ones like subnet 8 do occur. It appears that large networks of students colluding is not common and they keep to smaller groups of 2 or 3 students.

6. Conclusions and Further Work

A program copy detection system was described along with a submission system for students to submit their programming assignments. From this a methodology was introduced that extracted information from the detection system in the form of a digital fingerprint and imbedded this within the submission logs. The logs then created a detailed audit trail of how students developed their assignments and gave powerful evidence on who created assignments and who copied.

There is an immense amount of information contained within the logs, which opens up all sorts of possibilities for study. Some of these would include

- Investigating patterns of how students develop and submit assignments
- Investigating patterns of how students collaborate or collude
- Investigating whether the methodology can be applied to other domains such as fraud detection or insider trading in trading markets.

Acknowledgements

I thank two anonymous referees for helpful comments.

7. References

Collberg, C., G. Myles, et al. (2004). Cheating Cheating Detectors. TR04-05. Tucson, Department of Computer Science, University of Arizona**:** 7.

Dick, M., J. Sheard, et al. (2002). Addressing student cheating: definitions and solutions. Annual Joint Conference Integrating Technology into Computer Science Education, Aarhus, Denmark, ACM Press.

Faidhi, J. A. W. and S. K. Robinson (1987). "An Empirical Approach For Detecting Program Similarity And Plagiarism Within A University Programming Environment." Computer Education **11**(1): 11-19.

Gitchel, D. and N. Tran (1999). Sim: A Utility For Detecting Similarity in Computer programs. Proceedings of the 30th SIGCSE Technical Symposium on Computer Science Education, New Orleans, Louisiana, US.

Halstead, M. (1977). Elements of Software Science, Elsevier.

Jones, E. L. (2001). "Metrics Based Plagiarism Monitoring." Journal of Computing Science Colleges **16**(4): 253 - 261.

Parker, A. and J. O. Hamblen (1989). "Computer Algorithms for Plagiarism Detection." IEEE Transactions on Education **32**(2): 94 - 99.

Prechelt, L., G. Malpohl, et al. (2000). Finding Plagiarism Among a Set of Programs with JPlag, Fakultat fur Informatik, Universitat Karlsruhe, Germany**:** 23.

Schleimer, S., D. S. Wilkerson, et al. (2003). "Winnowing: Local Algorithms for Document Fingerprinting." 76 - 84.

Verco, K. L. and M. J. Wise (1996). Software for Detecting Suspected Plagiarism: Comparing structure and Attribute-Counting Systems. 1st Australian Conference on Computer Science Education, Sydney, Australia.

Wise, M. J. (1996). YAP3: Improved Detection of Similarities in Computer Program and Other Texts. Proceedings of the 27th SIGCSE Technical Symposium on Computer Science Education, Philadelphia, USA, ACM Press.