Coping with plagiarism in Computer Science teaching laboratories

Adrian West

Department of Computer Science

The Victoria University of Manchester
Oxford Road
Manchester M13 9PL

email: ajw@cs.man.ac.uk
Tel: +44 161 275 6251

July 1995

Abstract

The problem of plagiarism in teaching laboratories is common and can easily reach epidemic proportions, with a highly detrimental effect on both morale and motivation across the student body. Having to police it however is a daunting prospect.

The University of Manchester has a large Computer Science department and hence is particularly vulnerable to plagiarism in its laboratories. Under this pressure we have developed an automated method for detecting it which automatically processes student work and keeps track of laboratories on a per-exercises basis. Staff can peruse the most suspicious cases which are presented visually and regions of similarity can be investigated. An advantage over manual approaches is that detection is independent of the date on which the work was submitted.

In this paper the system is presented and its method of application described, we discuss some of our experiences with it, and what direction the project is currently taking.

Introduction

Computing, perhaps more than most practical subjects, affords ease of plagiarism within teaching laboratories. Programs are easily copied and even thinly disguised work is likely to evade detection in large classes. At Manchester, one of the largest Computer Science departments in the UK the problem has on occasion reached epidemic proportions, with students developing advanced techniques of code re-use well ahead of the official curriculum. The effect is demoralising for the more conscientious students as their efforts appear wasted, and weaker students succumb to the pressure as they see their comrades keeping up with work through copying. The pressure on students to copy can of course be due to the fact that the work is too difficult, but the fact that many appear to complete on time masks that reality making effective laboratory management difficult.

In a large department with many staff and postgraduate students marking an exercise over a period of time – sometimes years in the case of resit students, – it is impractical to address the problem informally. Most of the effort is already consumed with marking and student assistance, and few relish the task of periodically sifting through piles of listings looking for correlations. Clearly we would prefer to put our efforts into good teaching rather than policing plagiarism, and so we were moved to seek a way of automating the task as much as possible.

Scope for automated detection

The detection of similarities is rather meaningless if the exercises are extremely short, or if they have been specified in such a way that there is little scope for a difference between solutions. In these cases other means of deterring the blind copying of work must be sought. One method we have used is to introduce "playing" cards with questions about the exercise. As part of the marking process students pick cards and answer the questions for a small number of marks. The questions should be relevant and interesting to a student who has done the exercise, but cause those who have copied with no understanding to flounder. In practice this appears to work well.

For most of our exercises however the students produce sufficient code that the idea of detecting plagiarism from examining the code alone is sensible. In this discussion we assume a few tens or hundreds of lines in a language such as c, pascal, or SML.

Even here of course, at some level of effort a student will be able to disguise a copied program beyond all possible recognition – even by experts for whom the two programs are brought to explicitly to attention. However, in order to do so, a student must have understood the operation of the program to a very high degree, as most lines of code will need to be reordered or heavily disguised, and in such a manner as not to be obviously perverse. This effort is usually far beyond what was called for in the exercise.

On this basis our general thesis is: an automated detector will be valuable if it is sufficiently difficult to fool, that the effort and comprehension required to disguise a program of a similar order to that of having written original code.

In considering a tool which would help us seek this laudable goal we take two steps. Firstly the general algorithmic difficulty of comparing programs, and secondly the practical management of the system as applied to the laboratories.

Detecting similarities

In practice, even the simple expedient of handing in exercises a few days apart, or ensuring that different staff mark the work, can provide effective immunity for even the least talented plagiarists. If student work can be made readily available, even a simple algorithm will detect many of these blatant cases. We have indeed experienced instances of students presenting work that they do not know how to run, and in once case with an email header still attached saying that this was a good way to get through the lab.

A simple approach

One approach that has been used by colleagues at short notice is to concatenate all the programs for an exercise, and then textually sort the lines. The result is replete with tell-tale signatures of such blatant copying, and the longer – more significant – lines will have risen to the top.

Although the above is simple, it has been effective, and gives a concrete example against which to judge more ambitious attempts. In its favour it is language independent, catches blatant copying, and indeed any attempt at disguise which leaves significant numbers of lines unchanged. The main charges against it however are that

- Being format sensitive it is easily fooled by common plagiarist practices, such as renaming variables or passing the code through a “pretty print” program.
- Secrecy asto the mechanism would perforce be an important factor in its success. Once the general approach is known plagiarists could readily see how to fool it.

- Whilst far better than manually comparing all listings, there is still a reasonable amount of work involved for the person maintaining the system.

Whilst we could trivially normalise out formatting (white space) dependency, in order to cope with anything more sophisticated we will have to comprehend the syntax of the programs, and parse them into their component leximes. Once this is done differences in variable names can be ignored, though comparing the structure of the programs remains difficult.

A statistical approach

A second approach developed by a colleague takes a courageous approach to identifying similar programs. From the tokenised sources, statistics are gathered based on such quantities as average identifier length, nesting depth, size of procedures and so forth. The intention is twofold. Firstly to produce a signature, or finger print, for a student from which significant deviation may usefully be noted, and secondly to determine relationships between the statistics that are sensitive to plagiarism. A fascinating proposal is to use a genetic algorithm to build the equations that define those relationships. Whilst from a pragmatic viewpoint this is rather speculative venture, it would at least have the advantage that – even given knowledge of the algorithm – it would be hard to know what would fool it from moment to moment, and what would not. A related drawback is that it is hard to know why the system believes something to be copied, and harder to know when it is just plain wrong. However the idea is certainly worthy of some investigation.

Our current approach

Currently we strike what appears to be a happy compromise. We tokenise the input stream, normalising out formatting and variable names, and then endeavor to find matching sequences of tokens anywhere between the pair of programs under scrutiny. Figure 1 shows the effect.

An “x” represents a point where the same token appears in both programs, and “x”s forming diagonals correspond to matching token sequences. The current detector seeks unbroken diagonals although clearly we can go further if we wish. Broken or displaced diagonals correspond to the insertion, deletion or rearrangement of tokens and would therefore also be of interest.

For each pair of programs examined the detector returns a list of matching sequences described by their length and position within the source programs. Longer matches are regarded as more significant when it comes to ordering the list of suspicions for an exercise. The resultant output is shown in figure 2 with names changed.

Each matching sequence is described by the source file starting positions in parenthesis, followed by the length of the sequence in numbers of source lines in both programs. The general pattern for such a file is for a number of highly suspicious cases at the top, falling rapidly to a noise level as we progress down the list. Where skeleton code is given to the students for an exercise this appears as common to all, so the pattern is generally just displaced to the right by these common matching sequences. This seems satisfactory, but a more elegant approach would be to explicitly factor out supplied code.

Given this list of matches it is a fairly straight forward matter to present the programs on the screen, and to step through the matching regions as shown in figure 3. The highlighting of matching regions has been omitted for clarity of reproduction.

The controls allow stepping between the pairs of programs in the results list of figure 2 and the matching regions in each pair. In this example the two programs are very lightly disguised.

Where groups of people have copied the same program it would also be useful to see the transitive closure that should exist in the results file, though this is fairly evident.

Having this visual presentation available greatly eases the task of administering the system. It provides an easy confirmation that the programs are indeed copied, and markedly reduces the effort needed to present convincing evidence to the students. This latter point is genuinely beneficial as there is less scope for the unpleasant and damaging debates that arise when evidence is not so readily visible.

Managing the system

Having a workable system for comparing two programs is only a part of the system that is required to manage plagiarism in the laboratories. It is also necessary to have some means of collecting the students exercises and marshaling the comparisons.

When students have their work assessed, their programs are usually demonstrated to a postgraduate student or member of staff. Having done this, the students then run a script which time stamps their work and mails it to an archive. A potential weak link is that the student could send something other than the program demonstrated. If we believe this to be a problem we encourage the demonstrators to have the source program compiled for the demonstration, and to ensure that is the one sent. This may appear extreme, though at least it is an available option. Some heuristics are possible to check that the submission is of the kind expected, and it will be a record that we have permanently available.

Submitting work to the archive is a requirement in order to have the mark registered. Previously it has not been practical to administer this policy though this has been remedied by integrating the archive with our wider laboratory management system ARCADE, described elsewhere in these proceedings. Amongst other things ARCADE emails students with their marks and progress reports and informs them about work they have had marked which has not been received in the archive.

When programs arrive in the archive they are tokenised and checked against all those already present. In this way, work which is submitted several months late is still processed routinely. Peculiarities in submissions – for example if it does not parse, are currently flagged, though automatically alerting the student to a problem is the next logical step.

The archive serves an additional purpose in providing a record of the students work. If marks go astray and the students have lost their listings and purged their filestore, we have the archive to fall back on. We therefore encourage use of the archive for all laboratory courses, even where plagiarism detection is not appropriate.

At the end of the year the archive is kept as a way of checking for solutions handed down across the years, and to check for students who resit a year handing in their earlier compatriots work, as from time to time appears.

results, conclusions and further work

The plagiarism detector has been running for some four years now. Curiously, on interviewing the students that it caught, most were genuinely struggling with the courses though had felt reluctant to seek the help of staff. The detector has therefore been helpful in identifying students who were in genuine need of assistance. A common occurrence is for weaker students to be helping each other rather more than they had realised, or cared to admit to themselves, and again the net results have been helpful.

Few hardened criminals have been identified however. The reason for this is almost certainly that they have avoided submitting their work, and in practice we had never enforced this effectively. Now that we have integrated the archive with ARCADE this will be remedied, as marks will not be awarded if the work is not sent.

Even so, the task of chasing up students and discussing their copied work is still time consuming, and has often been neglected though the results themselves were readily available. Based on our experiences of automated detection, our next experiment is to see if we can find an effective way of informing students automatically as soon as copied work is detected. In this way we hope to discourage the practice as soon as it arises so that it will no longer be a significant issue in our laboratories.

Acknowlegments

I would like to thank the many people who have contributed to the plagiarism detection project over the years. Koshor Mistry who designed and coded the first version and Simon Peterkin who developed the X interface. Malcolm Schute for his work and discussions about Pfinger – his statistically based detector. John Latham for integrating the current version into ARCADE, and for many discussions on new approaches to administering this activity.

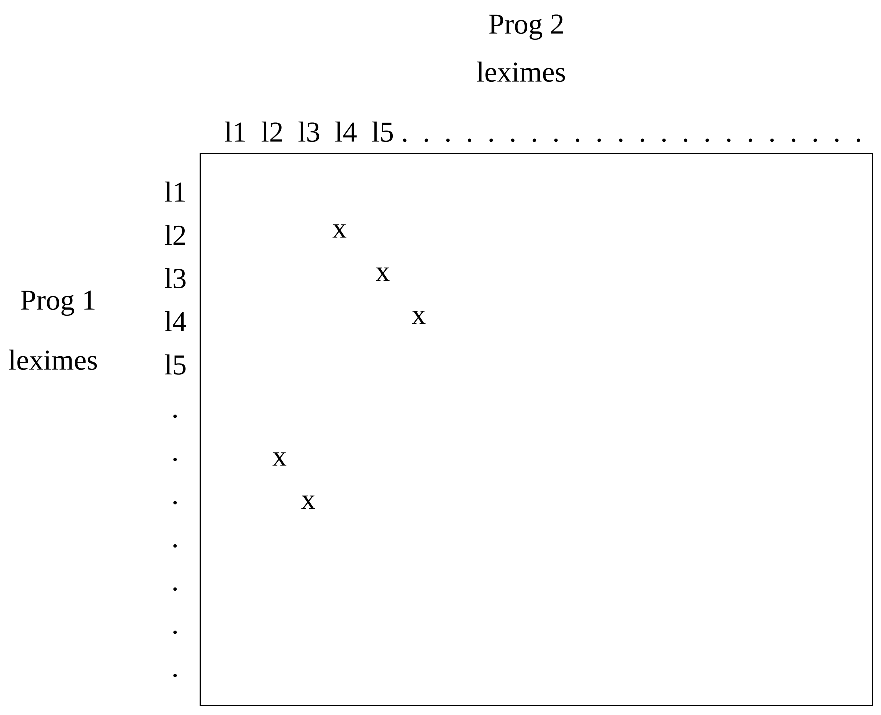
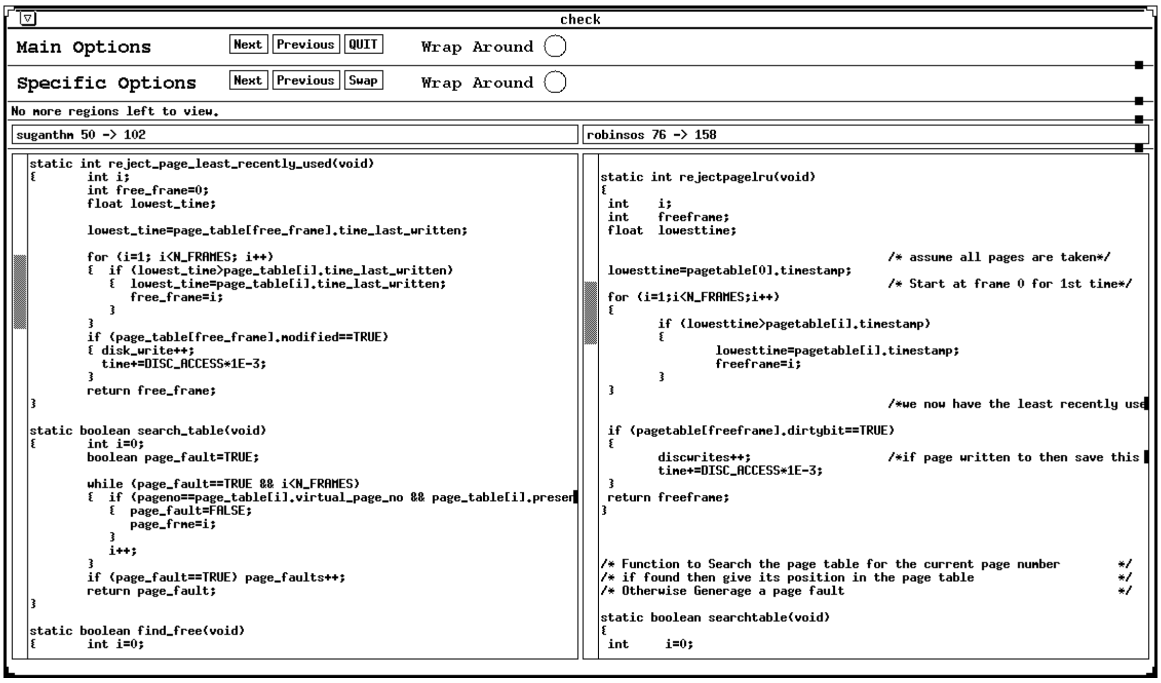


Figure 1: comparison matrix

```
flintstf  rubbleb  - (50,76)=53 83 (109,169)=45 56 (6,12)=38 56 (175,254)=30 27
mousem    mousemi  - (125,139)=52 55 (6,6)=44 49 (91,103)=26 26 (69,78)=19 22
brucel    chanj    - (73,160)=25 30 (134,48)=23 31 (156,78)=16 22 (175,103)=9 9
blofelde  bondj    - (190,136)=25 28 (125,102)=20 18 (94,73)=27 25 (36,39)=10 8
batm      robin    - (77,59)=34 35 (34,24)=14 17 (138,132)=14 12 (60,44)=8 6
joker     penguine - (88,54)=20 22 (16,17)=17 18 (149,201)=16 13 (45,149)=10 10
sinbad    colosus  - (142,132)=32 40
cayote    roadrun  - (170,208)=16 17 (109,135)=20 17 (133,158)=22 25 (37,45)=6 8
tom       jerry    - (69,76)=21 49 (30,32)=9 11
eigor     baron    - (83,45)=9 8 (107,82)=11 8 (203,138)=15 18 (12,6)=20 20
droopy    sleepy   - (100,63)=26 18 (126,82)=14 11 (68,47)=17 10
dozy      happy    - (173,168)=21 29
grumpy    busy     - (85,105)=34 17
snowwhit  giant    - (110,167)=19 27
kongk     godzila  - (103,60)=23 22
.....
```

Figure 2: output of matching sequences



```
No more regions left to view.
```

```
suganthm 50 -> 102
```

```
static int reject_page_least_recently_used(void)
{       int i;
        int free_frame=0;
        float lowest_time;

        lowest_time=page_table[free_frame].time_last_written;

        for (i=1; i<N_FRAMES; i++)
        {  if (lowest_time>page_table[i].time_last_written)
           {  lowest_time=page_table[i].time_last_written;
              free_frame=i;
           }
        }
        if (page_table[free_frame].modified==TRUE)
        { disk_write++;
          time+=DISC_ACCESS*1E-3;
        }
        return free_frame;
}

static boolean search_table(void)
{       int i=0;
        boolean page_fault=TRUE;

        while (page_fault==TRUE && i<N_FRAMES)
        {  if (pageno==page_table[i].virtual_page_no && page_table[i].presen
           {  page_fault=FALSE;
              page_frme=i;
           }
           i++;
        }
        if (page_fault==TRUE) page_faults++;
        return page_fault;
}

static boolean find_free(void)
{       int i=0;
```

```
robinsos 76 -> 158
```

```
static int rejectpagelru(void)
{
 int    i;
 int    freeframe;
 float  lowesttime;

                                        /* assume all pages are taken*/
 lowesttime=pagetable[0].timestamp;
                                        /* Start at frame 0 for 1st time*/
 for (i=1;i<N_FRAMES;i++)
 {
        if (lowesttime>pagetable[i].timestamp)
        {
                lowesttime=pagetable[i].timestamp;
                freeframe=i;
        }
 }
                                        /*we now have the least recently use

 if (pagetable[freeframe].dirtybit==TRUE)
 {
        discwrites++;                   /*if page written to then save this
        time+=DISC_ACCESS*1E-3;
 }
 return freeframe;
}



/* Function to Search the page table for the current page number        */
/* if found then give its position in the page table                    */
/* Otherwise Generage a page fault                                      */

static boolean searchtable(void)
{
 int     i=0;
```

Figure 3: viewing tool