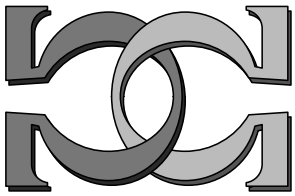

CDMTCS
Research
Report
Series

Plagiarism Detection with State of the Art Compression Programs

Cristian Grozea
University of Bucharest
Romania

CDMTCS-247
August 2004

Centre for Discrete Mathematics and
Theoretical Computer Science

Plagiarism Detection with State of the Art Compression Programs

Cristian Grozea
Department of Computer Science
University of Bucharest
Str. Academiei 14, 70109 Bucharest, Romania
chrisg@phobos.ro

Abstract

This note documents a new approach to plagiarism detection, based on Algorithmic Information Theory. It uses results from the author's Ph.D. Thesis.

Introduction

The story: In the second term of 2004 I had a course on web programming (the client side: html, javascript). I assigned to each of the 60 students the same set of 19 projects (some of them simple). Some of the students found this number too high. So, they tried to cheat by copying entire solutions or only part of those from their colleagues.

Because they know the examiners are usually browsing the sources and maybe in order to make more difficult the automatic fraud detection, they tried to hide by doing quick transformations that didn't change the program behavior. Some of those were: reformatting the source (changes in white space), renaming the variables and functions used, adding comments (those were sometimes intended to help them answer the examiner's questions during source examination).

I received around 400 projects. Each project had one or more files. Some of those were pictures. I have deleted all pictures, as I was only interested in checking the programs. Unfortunately, each student named each project in a different way, so it was no way to pair those automatically. As a result around 80.000 pairs of files had to be compared.

I needed a method (as fast as possible - with just one second per file compare you get almost 22 hours for all pairs) to compare two files A and B in order to find out if they have "too much in common".

It's hard to define what "having too much in common" means. Other approaches define sometimes the minimum length that a substring should have in order to be a proof of plagiarism [3]. Because of the transformations applied by the students globally over the file (such as reformatting text and variable renaming) you couldn't find such lengthy common strings. A preprocessing step to eliminate the white space wouldn't have helped too much with variable renaming. A preprocessing idea I have considered was to rename all variables as x or similar, but then a syntactic parser would have been needed and much more programming, so we didn't proceed with this approach.

The editing distance of strings or derivate algorithms looked nice for a while, but they would still have to be programmed and it is not clear at all if one can derive an edit distance that take search / replace operations into account.

Information

Let's forget about editing for now. What I really wanted to know was if the files A and B had too much information in common. The *information* is the key word.

From an informational point of view, I needed a way to determine if the new information in B when you already know (have) A is much lower than expected (or the other way around, with A and B exchanged).

Formally I consider that A and B have too much in common when
$K(B|A)/K(B) < d$ or $K(A|B)/K(A) < d$ where K is the Kolmogorov complexity and d is a fixed threshold.

The complexity of A, $K(A)$ is a measure of the quantity of information in A, the relative (conditional) complexity $K(A|B)$ is a measure of the quantity of the information needed to describe A when you have B.

It is difficult to give the right threshold d by theoretical reasoning, so we obtained the value 60% by doing experiments and checking that the number of false positives matches the number of false negatives.

Everything looks appealing and simple (hopefully to the reader too), but there was a very big problem. You cannot compute $K(\cdot)$ and as a result you cannot compute $K(\cdot|\cdot)$ either. K is not computable, being able to compute it is equivalent with being able to solve the Halting Problem. As a result, you cannot find out how much information a

given string (the content of a file) contains.

Then what about using an approximation of K, some computable function K' with let's say $K - c < K' < K + c$ for some constant c or $K/c < K' < K * c$. Nothing like this exists, there is a stronger result that proves that you cannot find computable lower bounds for K.

Let's now look at this from a different perspective. $K(x)$ is the length of the shortest compressed form of x, is the best compression you may imagine (if we ignore some fixed additive constant). This ideal compression cannot be practically achieved and you cannot even go close to it. But you can still compress, using the current state compression programs. The better it compress, the better results we expect to have, as always when dealing with approximations.

There is nothing like "the best compression program in the world". Each has its strengths and limits. Some are better in some data domains (image compressors, for example). We are going to consider the text compression, so we will limit our attention to text compression algorithms and programs. We intend to approximate $K(x)$ by the length of the result of the compression of x with an actual compression program f, $len(f(x))$.

The second big problem is that the off-the-shelf compression programs are not able to do contextual compression (compress A given B). For some of them, if you are willing to go to the algorithm, change it and reimplement it, this can be done, but the task seems difficult to do. And by doing that we would loose two of the advantages of the approach here: using already written (and carefully tested for years) programs as well as simplicity.

A simple theoretical analysis shows that we can limit ourselves to using the absolute Kolmogorov complexity K.

We have $K(XY) < K(X) + K(Y|X) + O(1) = K((X,Y)) + O(1) < K(XY) + min(K(len(X)), K(len(Y))) + O(1)$, where $len(X)$ is the length of the file/string X.

Also $K(len(X))$ is very small (1..2 bytes) for any file/word X used here, because the length of any file was under $64k$.

Given the inequalities above we can approximate $K(Y|X)$ by $K(XY) - K(X)$. In other words, if the compressor is ideally good, by subtracting the length of the compression of X from the length of the compression of X concatenated with Y you get a good estimation of the new information that Y brings when you already know X.

Is this still true if you are a using a regular compression program instead of the ideal one? Let's first decide what would be reasonable to have:

- $K(X|X)$ should be close to 0, no matter what X contains. Is this still valid if we use

an actual compression algorithm f? This would mean that $len(f(XX)) - len(X)$ is close to 0. There are few compression programs that compress well this kind of redundancies. From the experiments we performed for our Ph.D. thesis [2] with several compression programs, including *gzip* and *winzip*, we identified two programs, *bzip*2 and *rar* as being close to this desired behavior. *bzip*2 is a freeware based on a published algorithm and it is open-source, so we preferred it to the shareware closed-source *rar*.

- $K(X|Y)$ should be sensibly less than $K(X)$ when X and Y are not independent, X being derived from Y. If you want, this is the same problem as above, just that more general. The practical results with *bzip*2 show that it is able to detect most of the redundancy we wanted it to detect.

- $K(X|Y)$ should equal $K(X)$ when X and Y are independent. This one is easily handled by most compression algorithm, even by the ones that do not comply with the two conditions above.

There was some fine-tuning of the procedure done during the experiments. For example, we observed that the very small files gave mostly false positive results. This is very much in line with the asymptotic nature of the theoretical results used here (just note that the constants behind O(1) have been determined by Chaitin for many results and they are not really big [1]). So we eliminated also the very short files, from which we did not expect any spectacular proof of plagiarism anyway.

The program

The program is a set of Perl scripts. Stripped of any optimizations (like caching) the main test that compares two files is the following.

```
#extract the 'information quantity' - using the compressor
sub information
{
my $f=shift;
my $l;
`cp -f "$f" ../t`;
`bzip2 -f ../t`;
$l=filelen('../t.bz2')-37; #here is where we subtract the constant
return $l;
}
```

```
#compare two files
sub compare
{
$threshold=0.6;
my $f1=shift;
my $f2=shift;
my $i1=information($f1);
my $i2=information($f2);
my $l1=filelen($f1);
my $l2=filelen($f2);
‘cat "$f1" "$f2" >../t12‘;
my $i12=information(’../t12’);
my $novelty1=($i12-$i2)/$i1;
my $novelty2=($i12-$i1)/$i2;
if ($novelty1<$threshold || $novelty2<$threshold)
{
print "possible plagiarism for $f1 $f2 \n" ;
}
}
```

Please note the constant 37 used in the function *information*. It is the length of the header used by the compression program bzip2. You can find what value this constant has for your compression program by compressing random files of few bytes. All tend to have roughly the same length, which is this additive constant.

Results

The results of applying the hereby-described procedure were very good. Almost 100 projects had common parts with projects of other students.

The degree of plagiarism ranged from exact copies to subtle similarities. The most interesting pairs of files are the ones with the similarity just above the threshold we fixed (60%). In some cases it is hard to tell if the program is actually copied or rewritten (a student confessed about rewriting in another location one of its programs, from scratch) or just a natural similarity of two programs that solve the same problem in the same way (same algorithm). We had a case where what the procedure detected was the use of the same sorting algorithm.

Further work: refining the method

The formula $K(X) + K(Y) - K(XY)$ would give a direct estimation of the *common information* of X and Y. Then we could detect plagiarism with a simpler test that implies no relativization: test if the common information of X and Y is greater than some *threshold*.

This would correspond vaguely to the common string based methods - that try to find a common string longer than some fixed threshold. Our method would keep its advantage of finding common parts that are not simple identical copies of each other.

This approach could also increase the speed of the plagiarism detection, as we could concatenate all the files of a project (or even of a student) into a single file, before testing. This would decrease the calls to the compressor, which would be beneficial, as calling the external compressor proved to be a slow operation. On the other side, the time complexity of the compressor could have an adverse effect. Some of them are $O(n^2)$ on each block. Also, if the file would get too large, the compressor won't be able any longer to detect all the redundancies if the distance between them would be to large.

Another possible advantage is that it wouldn't miss the students that, finding out that we ignore the short files would split their programs into very small files.

This method would also resist to large blocks or random or useless texts (maybe in comments), intentionally inserted to lower the relative significance of the common part. These random blocks are used by spammers against statistical tests.

The main disadvantage would be that would not be so easy to say which file takes after which one. Anyway a careful analysis done by a real person is needed in cases of positives, because of the implications the accuse of plagiarism has.

Conclusion

The compression proved to be an effective way of detecting the redundancies that are inherent to plagiarism.

The strength of this technique is that it is practical (we have already applied it) and not deeply involved into the internals of some text matching or text compression algorithm, because it doesn't really matter what compressor you use, as long it is good enough. The better the compressor you use, the better the accuracy you get.

References

[1] G.J. Chaitin, *The Limits of Mathematics,* Springer-Verlag, Heidelberg, 1998.

[2] C. Grozea, *Contributions to the AIT*, Ph.D. Thesis, Bucharest University, 2003.

[3] S. Schleimer, D.S. Wilkerson, A. Aiken, Winnowing: Local Algorithms for Document Fingerprinting, *SIGMOD* 2003.