Signature extraction for overlap detection in documents

Raphael A. Finkel, Arkady Zaslavsky, Krisztián Monostori, and Heinz Schmidt

raphael@cs.uky.edu; University of Kentucky, Lexington, KY, USA
A.Zaslavsky@monash.edu.au; Monash University, Melbourne, Australia
Krisztian.Monostori@csse.monash.edu.au; Monash University, Melbourne, Australia
Heinz.Schmidt@csse.monash.edu.au; Monash University, Melbourne, Australia

Abstract

Easy access to the Web has led to increased potential for students cheating on assignments by plagiarising others' work. By the same token, Web-based tools offer the potential for instructors to check submitted assignments for signs of plagiarism. Overlap-detection tools are easy to use and accurate in plagiarism detection, so they can be an excellent deterrent to plagiarism. Documents can overlap for other reasons, too: Old documents are superseded, and authors summarize previous work identically in several papers. Overlap-detection tools can pinpoint interconnections in a corpus of documents and could be used in search engines.

We describe a web-accessible text registry based on signature extraction. We extract a small but diagnostic signature from each registered text for permanent storage and comparison against other stored signatures. This comparison allows us to estimate the amount of overlap between pairs of documents, although the total time required is linear in the total size of the documents. We compare our algorithm with several alternatives and present both efficiency and accuracy results.

Keywords: plagiarism document overlap culling digest

1 Introduction

It is increasingly difficult for instructors to verify that the work that students submit is their own. The Web hosts an enormous amount of material that students can easily find and pretend is theirs. Trying to discover such cheating requires enormous effort, well beyond what any instructor is willing to exert. Instructors are apt to even avoid the much smaller problem of detecting copying within a group of students on a single project. Luckily, since students generally submit their work electronically, instructors can apply computerised methods for discovering cheating automatically.

Recent well-publicised cases [Argetsinger, 2001, Benjaminson, 1999] show that this problem is international and significant even at well-respected educational institutions and that instructors are turning increasingly to automated techniques to detect plagiarism.

Detecting overlap can be valuable in other situations as well. A corpus of documents may have internal connections; documents referring to the same technical subject may contain significant overlap, particularly if they descend from a common ancestor. Someone reading one document might wish to see others that are related in such a fashion.

In this paper, we report on SE, an algorithm for extracting signatures from files and storing them for fast comparison with other files. Our algorithm is subject to both false positives and false negatives, so it should be used in conjunction with other software.

It is beyond the scope of this paper to offer more than a few thoughts on issues connecting document overlap with plagiarism. Plagiarism can be defined as taking from others their words and ideas and passing them on as one's own, as original. "Word" in this context does not only include sections of text; it extends to quotes, ideas, graphics and diagrams, charts, tables and figures, and electronic documents. As theft, plagiarism is clearly unethical. Some authors distinguish plagiarism and cheating by either reducing plagiarism to simple similarity (without passing the content on as original) or adding to the above broad interpretation the intention to gain advantage unfairly.

Similarity between documents is positively correlated to plagiarism in the overwhelming majority of cases we have experienced. To diagnose cheating, however, requires information outside the purview of our automatic tools. However, instructors can use our software to build a plagiarism or a cheating case leading to a disciplinary process whose outcome may be an academic or civil penalty. Any such disciplinary process requires a well specified plagiarism policy, regular education and awareness campaigns alerting students to the issues and tools used, and committed honest academics dealing with the extra work resulting from suspected and real plagiarism cases, especially initially when such policies are new.

Confidentiality requires that automatic tools have insufficient access to the original documents to prove cheating. It is better to leave the onus of developing a case for cheating with the instructor, who can embed the case in the local academic and social contexts. For that reason, we do not store even extracts from the original documents, and we identify the original documents and the instructor (document **meta-information**) only by codes provided by the submitting instructor. It is up to the instructor to maintain information that allows those codes to be interpreted later if necessary.

2 Signature extraction and comparison

The basic idea is this:

1. Partition each file into contiguous chunks of tokens.
2. Retain a relatively small number of representative chunks.
3. Digest each retained chunk into a short byte string. We call the set of byte strings derived from a single file its **signature**.
4. Store the resulting byte strings in a hash table along with identifying information.
5. If two files share byte strings in their signatures, they are **related**. The closeness of relation is the proportion of shared byte strings.

Each step can be accomplished quite rapidly. Mutual comparison of f files takes $O(f)$ time. Ordinary pairwise methods would require at least $O(f^2)$ time.

2.1 Chunking

The method used to tokenise and partition files depends on the type of data. We have experimented with two methods, one for text files and the other for program files.

For text partitioning, we discard all punctuation and tokenise based on white space. Each token is then hashed with a simple, quick, hash function. The resulting hash value is tested for equality to 0 mod c, where c is a fixed number. We generally set $c = 10$. Any token whose hash value is 0 mod c ends the current partition. We expect, therefore, that the average chunk size will be c tokens long, although a chunk can be as small as one token and as large as the whole file.

This **hashed-breakpoint** chunking method [Shivakumar and Garcia-Molina, 1996], has the good property that an insertion or deletion of tokens in the file only affects adjacent chunks. It has the bad property that if a common word hashes to 0 mod c, most chunks will be quite short.

For programs, we use **syntax-based chunking**, which divides the text into paragraphs, either by placing each subroutine in a paragraph, which requires parsing, or by delimiting paragraphs by blank lines, which requires no parsing. Each token is replaced by a single letter indicating its function. For instance, each keyword has its own single-letter abbreviation, such as `f` for `for`. Other identifiers become *i*, numbers become `0`, and punctuation is retained. White space is removed. Each paragraph therefore becomes a single space-free string.

2.2 Culling

We could store all chunks, but long files lead to a many chunks. Dealing with them all uses space for storing them and time for comparing them against other stored chunks. However, it is not necessary to store all chunks.

A short chunk is not very representative of a text. The fact that two files share a short chunk does not lead us to suspect that they share ancestry. In contrast, very long chunks are very representative, but unless a plagiariser is quite lazy, it is unlikely that a copy will retain a long section of text.

We therefore discard the longest and the shortest chunks. We wish to retain similar chunks for any file. We have experimented with two culling methods. Let n be the number of chunks, m the median chunk size (measured in tokens), s the standard deviation of chunk size, and b a constant, and L the length of an arbitrary chunk.

- **Sqrt.** Retain $\lceil \sqrt{n} \rceil$ chunks whose lengths L are closest to m.
- **Variance.** Retain those chunks such that $|L - m| \leq bs$. Increase b, if necessary, until at least $\sqrt{n}$ chunks are selected. We start with $b = 0.1$.

2.3 Digesting

We could store the chunks themselves, but we choose not to do so for two reasons. First, chunks may be quite large, and we wish to limit the amount of storage required. Second, chunks contain the intellectual property of the author of the file that we are processing. We prefer not to store such property in order to reduce fears that our tools can themselves be used to promote plagiarism or that the database can be used for breaches of confidentiality and privacy.

Instead of storing the chunks, we reduce them by applying a digesting tool. We use the MD5 algorithm [Rivest, 1992], which converts arbitrary byte streams to 128-bit numbers. We retain the d leading hex digits of the MD5 digest; currently, d is set to 10. Retaining more digits costs in storage space and decreases the likelihood of a false positive. However, 1 false positive in $16^{10} \cong 10^{12}$ chunks is not a serious problem.

Given a file F, we call its set of digested retained chunks $d(F)$.

2.4 Storing

We store $d(F)$ in a pair of hash tables called `keyData` and `nameData`. Both are implemented as Perl [Wall and Schwartz, 1992] DB files tied to Perl hash data structures. The name of the file F indexes `nameData`, where we store a record containing information on the submitter of the file, the date, and the number of chunks in $d(F)$. We do not restrict the names by which submitters identify themselves; it is wise for each submitter to pick a unique and unguessable personal identifier if secrecy is important. Likewise, the submitter may obscure file names if desired.

Each digest in $d(F)$ indexes `keyData`, where we store a record containing a list of all the file names that share this particular digest.

2.5 Comparing

To compare a file F to the database, we use all digests in $d(F)$ to index `keyData`. We count how many of these digests are found in other files in the database. A single pass over $d(F)$ allows us to calculate $d(F) \cap d(G)$ for all files G whose digests overlap $d(F)$. For all files G with non-zero intersection of digests with F, we compute three measures of similarity.

- **Asymmetric similarity.**

$$a(F,G) \;=\; \frac{|d(F) \cap d(G)|}{|d(F)|}$$

- **Symmetric similarity.**

$$s(F,G) \;=\; \frac{|d(F) \cap d(G)|}{|d(F)| + |d(G)|}$$

- **Global similarity.**

$$g(F) \;=\; \frac{|d(F) \cap (\cup_G\, d(G)|}{|d(F)|}$$

A similarity of 0 indicates no overlap; a similarity of 1 indicates complete overlap of the digests, which implies significant overlap of the files. Asymmetric and symmetric similarities differ in their treatment of files of disparate lengths. If file F is a small excerpt of a much larger file G, then $a(F,G) = 1.0$, but $s(F,G) \ll 1.0$. Global similarity indicates the degree to which file F overlaps with all other files in the database.

2.6 Web access

Our web site accepts several types of query.

- **Pairwise test.** The user submits a *tar* or *zip* file containing a directory. For each F in that directory, we enter $d(F)$ into a temporary database. In a second pass over each F, we calculate $s(F,G)$ for all pairs for which there is overlap of digests. (We only use the symmetric similarity in order to reduce the amount of information presented to the user.)

  We convert similarities to distances by setting $distance(F,G) = \lceil -b \log(s(F,G)) \rceil$ where b is selected so that the distances are reasonably displayed in a browser by a graph where edges between F and G are $distance(F,G)$ pixels apart; a typical value

of b is 100. We then display those files with non-zero similarities in a graph in the user's browser along with our modified version of Sun Microsystems' `Graph.java`, which uses a hill-climbing technique to attempt to display the graph on the plane with distances set as specified. The user can then see which files are related along with some feeling for their similarity.

We limit the number of files displayed to the 20 with the highest similarity in the interests of readability and display speed.

We also collect the 20 most-similar files to submit to more computationally expensive methods, such as MDR [Monostori et al., 2000].

The user can set the threshold for the number of top-ranked files to analyse in more detail.

- **Verification.** The user submits a directory, as before. For each F in that directory, we consult the global database to derive $s(F, G)$ for all files G showing significant overlap of digests. We display these results as before in a graph and collect the most-similar files for submittal elsewhere.

- **Registration.** Again the user submits a directory, along with a personal identification string and a directory-name string, which we append to all the file names in the directory. As mentioned earlier, we do not restrict the nature of these strings (although we may remove special characters that interfere with other processing). All elements of $d(F)$ for all files in the directory are inserted into the **registry**, which is a permanent database.

3 Experience

We have tested signature extraction with hashed-breakpoint chunking using several bodies of texts. The first body is the set of 2591 RFC documents of the Network Working Group. Nine of these documents are quite short, saying only "This RFC was never issued." Several files just indicate that the document is only available in a different format. However, most of the documents are at least 10KB in length; the largest is almost 500KB, and the combined documents occupy 112MB. We do not expect plagiarism as such in this corpus of documents. Instead, we use overlap detection to discover families of related documents.

We find that the Sqrt method of culling does not store enough chunks for short files to allow reasonable testing. The Variance method, however, tends to do much better. Even small files are represented by a significant number of chunks, because the variance in chunk length tends to be high.

A complete pairwise test of the RFC body (using text-file chunking) takes about 21 minutes on an UltraSparc 10/440MHz for the first pass, which builds the two hash files. The resulting files have length 286KB (`nameData`) and 5.0MB (`keyData`). The second pass, to compare each document against the database, takes about 23 minutes.

The result of the pairwise test is a list of 13365 file pairs that show non-zero similarity. As expected, all the trivially short files have similarity 1.0 (both symmetric and asymmetric). File pairs with symmetric similarity between 0.9 and 1.0 include (2264,2274), (1596,1604), and (1138,1148); in each case, the second RFC is an update of the first with substantial retained text. A file pair with symmetric similarity 0.8 is (2059,2139). Again, the latter RFC obsoletes the former; here, the update is not quite so trivial. The pair (1048,1084), with symmetric similarity 0.7, represents a still less-trivial update.

An increasing number of pairs is found as the symmetric similarity drops. Not counting the trivially identical files, there are 5 pairs with similarity in $[0.9,\ 1.0)$, but 24 with similarity in $[0.8,\ 0.9)$, 30 in the next group, 60 in the next group, then 87, 168, 301, 2046, 2865, and 1083. The fact that a great number of files show at least one chunk in common with another is not surprising; RFC files tend to follow a standard format and use a standard descriptive tone.

To check the accuracy of signature extraction, we also tested certain file pairs with a much more expensive but highly accurate MDR method that uses suffix trees to find the exact amount of overlap between file pairs [Monostori et al., 2000]. MDR takes about a week of computer time to find pairwise similarity within a directory of all RFC files; we therefore tested it only on some combinations of RFC files mentioned above. MDR computes the global similarity between each file and all the given files, so we prepared datasets with only two elements in order to derive the asymmetric similarity. Table 1 compares the MDR method with signature extraction (SE) and the overlapping-chunks method (OV, discussed later), showing both asymmetric similarity measures for each pair of files.

This table shows that signature extraction provides adequate accuracy. It tends to underestimate the similarity between very similar files (near the top of Table 1) and to overestimate the similarity between very dissimilar files (near the bottom of the table). Both of these tendencies can be explained by the sampling nature of signature extraction. Files that are highly, but not completely, similar are likely to have a few chunks that cross the boundaries where they differ. These chunks will not match, so even though there are few such boundaries, the similarity measure will be artificially low. In files that are very dissimilar, if even one chunk is derived from a region of similarity and that chunk survives culling, it will contribute to the similarity measure.

A second body of files is the set of 154 sonnets by Shakespeare. No sonnets showed any similarity under our methods. In contrast, MDR found that sonnets 36 and 96 happen to have identical final couplets. There was no other similarity. Sonnets are most likely too short to compare effectively with signature extraction. The small amount of similarity in this case was beneath the threshold of visibility.

A third body of files involves program submissions for a graduate-level operating-systems class at the University of Kentucky. We analyzed a dataset of approximately 5MB (the first assignment, 65 students) and 6 MB (the second assignment) using syntax-based chunking. Each analysis took about 30 seconds of computation on a 1GHz Pentium III computer running Linux and generated about 100KB of hash table. In the first assignment, each submission included several files containing a sorting program written three different ways in C (doing a mergesort via recursion, pthreads and forked processes, respectively). The output of the overlap detector listed pairs of files with significant commonality among the chunks. We discarded pairs where both files belonged to the same student. Two students submitted copies of the standard data set, showing similarity 1.0. The most significant remaining overlap involved two C program files with symmetric similarity of 0.40. Visual inspection of the files involved revealed variable renaming and program reordering, but a clear similarity of code. The students turned out to be good friends who had worked together to some extent. The other sets of pairs had similarity of 0.20 or less; visual inspection revealed no significant overlap.

In the second assignment, there were more pairs of files showing suspiciously large overlap. The makefiles (used by a the *Make* program-construction utility) for 3 students showed similarity 1.0; they were clearly copies. There was a similarity of 0.73 between two C program files. Visual inspection found essentially identical code, with significant identifier renaming, for a large segment of the files. There were other file pairs with similarity reaching down

RFC 1	RFC 2	MDR 1	MDR 2	SE 1	SE 2	OV 1	OV 2
1596	1604	99	99	91	92	94	94
2264	2274	99	99	96	95	94	94
1138	1148	96	95	93	92	91	89
1065	1155	96	91	71	68	84	79
1048	1084	94	91	73	67	87	82
2059	2139	92	90	77	83	83	81
1084	1395	86	84	58	64	79	75
1497	1084	82	87	38	42	73	79
1600	1410	72	77	52	48	58	61
2497	2394	19	17	33	27	16	15
2422	2276	18	3	23	6	15	2
2392	2541	16	12	27	17	13	10

Table 1: Asymmetric similarities: MDR, signature extraction, and overlapping chunks

characteristic	SE	OV	increase factor
time (seconds)	28	128	4.6
space (KB)	40	5200	130

Table 2: Resource requirements: signature extraction and overlapping chunks

to 0.30 that most likely indicated some amount of copying.

These experiments show that overlap detection is clearly both inexpensive and effective. It seems to work even for fairly short files (the makefiles were less that 1K in length).

4 Alternatives

Many other approaches to similarity checking have been proposed, based both on storage of extracts and storage of entire documents.

4.1 Extract storage: Overlapping chunks

Our method stores extracts for later retrieval. We have chosen to extract digests of chunks we hope are representative. One alternative is to store digests of all chunks of b (say 10) consecutive tokens, with each token beginning an (overlapping) chunk. This method requires space proportional to the length of the file. It is likely to be less subject to false negatives, and it is likely to give a more accurate measure of similarity. We have implemented this "overlapping-chunks" method and found that it is indeed more accurate (see the OV columns in Table 1) than our SE method, but it is also far more expensive, as shown in Table 2.

4.2 Extract storage: Multidimensional numeric values

Instead of storing information derived from chunks of the text, we can store statistical information about the text. For example, we could compute stylistic measures such as the average number of syllables in words, the frequency of passive constructions, and the number of dependent clauses. Content-based measures could include the number of uses of words from various technical vocabularies. These measures could be normalised based on the length of the text to derive a vector of numbers that represents the text.

The signature of file F is $P(F)$, the k-dimensional vector of numbers derived from F by some statistical method. The **multidimensional similarity** of two files F and G is defined as

$$m(F,G) = distance(P(F), P(G))$$

for some k-dimensional metric $distance(\cdot,\cdot)$.

A registry of signatures can be arranged in a k-d tree [Friedman et al., 1977]. (An alternative structure has been suggested for finding approximate nearest neighbours when k is high [Arya et al., 1994].) The k-d tree structure has several attractive properties:

- The vectors in a k-d tree need not have the same dimensionality. The metric $distance(x, y)$ can be designed to ignore dimensions that do not occur in both x and y. Nonetheless, it seems appropriate to segregate vectors derived by different statistical measures into different trees.
- K-d trees can be built incrementally. As with binary search trees, they tend to be better balanced if all the vectors are available from the start, but they do not become terribly unbalanced except in pathological cases when vectors are added incrementally.
- K-d trees can be stored efficiently in files even with incremental addition. We have built a Perl module for k-d trees that uses an ordinary text file with random access for the data structure.
- The incremental cost of adding a vector is $O(k \log n)$, where n is the number of vectors already in the tree. The cost of finding the nearest neighbour to a vector (possibly itself not in the tree) is also $O(k \log n)$, assuming $distance(\cdot,\cdot)$ takes time $O(k)$.

Overlap detection is accomplished by storing signatures of all texts in a single k-d tree. The nearest neighbour to each text is the best candidate for an overlapping document, and the distance between the two texts is a measure of similarity. Other methods such as MDR can then be applied to investigate the connection between the files.

We have experimented slightly with this multidimensional-numeric (MN) approach. A very simple statistical analysis measures the percent of words in a file that are the common words `the`, `and`, `but`, and `so`, yielding a 4-dimensional vector. Our data set was 22 of the RFC files. For each file we determined the other file closest in 4-space using the Euclidean norm. Our MN results, scaled by 10^7, are shown in Table 3, along with the SE symmetric similarity. In most cases, if F's closest neighbour is G, then G's closest neighbour is F. Table 3 indicates exceptions to that situation with $\bullet$. It took 2 seconds to complete the nearest-neighbour pairwise analysis compared to 27 seconds for the SE analysis.

The first few rows of Table 3 show that the MN method has promise. The top 7 pairs are accurately discovered, in the sense that in each case, SE found no closer match for any of the files involved. A more comprehensive statistic might do even better, particularly for the files lower in the table.

The MN approach suffers in general from the fact that it treats files in their entirety. It is unlikely to suspect a file that includes a direct quotation from another unless the quotation is a large fraction of the whole. Our SE method, on the other hand, might well notice such quotation, although it might assign it low significance.

RFC 1	RFC 2	MN distance	SE
2264	2274	0.1644192	95
1138	1148	0.5555986	92
1596	1604	1.160099	92
1395	1084	10.94293	58
2139	2059	47.32077	77
• 1048	1084	98.74357	67
1155	1065	384.5729	71
• 1497	1084	487.4635	42
2394	2392	745.3948	18
• 2422	1155	1323.221	0
2541	2276	1388.167	6
1600	1410	1770.094	48
• 2497	1600	1974.232	0

Table 3: Nearest-neighbour in 4-space

4.3 Complete storage: Compression

Web search engines such as Google [Google, 2001] store entire documents. Their data structures are tuned not to similarity detection but rather fast retrieval based on a limited set of keywords.

Given complete storage, various sophisticated algorithms can be applied to determine similarity. One is the MDR method mentioned above. Another class of methods is based on **compression**: Given two files, F and G and a compression algorithm $compress$ (such as `gzip`), symmetric similarity is defined as

$$s(F,G) = 2 - \frac{2|compress(F+G)|}{|compress(F)| \ + \ |compress(G)|} \ ,$$

which yields a value in $[0,\ 1]$. (We use "+" to represent file concatenation.) The idea is that an excellent compression algorithm will "notice" similarities between F and G and will not use much extra space to represent regions of overlap. At one extreme, we would expect that $compress(F\ +\ F)$ will have about the same length as $compress(F)$, leading to a similarity of 1. At the other extreme, if F and G are completely unrelated, we would expect that $compress(F+G)$ would have length equal to the sum of the lengths of F and G compressed individually, leading to a similarity of 0. If F and G have some similarity (for instance, they are documents written in the same language), we would expect a value of similarity between 0 and 1.

We have experimented with several compression algorithms, including `gzip` [Deutsch, 1996], `bzip2` [Seward, 2000], and `ppm*` [Cleary and Teahan, 1997]. We find reasonable correlation between the computed symmetric similarity and that computed by MDR, as shown in Table 4. The entries marked with "*" show a failing of `gzip`: When the files get too large, it appears to "forget" information about the first before it is able to use it on the second, leading to very poor joint compression and therefore artificially low similarity measure. Both `bzip2` and `ppm*` appear immune to this problem with the file sizes involved.

The results show that compression tends to underestimate the similarity between very similar files (those near the top of Table 4) and overestimate the similarity between dissimilar files (near the bottom). The former tendency is due to the imperfection of compression. Even when we test RFC 2300 against itself, `ppm*` gives only 92% symmetric similarity. The latter tendency is due to **residual similarity** between unrelated files; after all, they are in the same language and certainly use much of the same vocabulary, although words are arranged differently. MDR is insensitive to extremely short similar regions; compression methods are not.

On a complete pairwise test of the sonnets database, `gzip` finds symmetric similarities between 14% and 28%; the most similar pair is in fact Sonnets 36 and 96, which share the closing couplet. `Ppm*` finds symmetric similarities between 9% and 25%; Sonnets 36 and 96 show 24% similarity.

Unfortunately, a pairwise test within a directory requires $O(n^2)$ compressions, each of which can be quite expensive. Although `ppm*` is apparently a better compression algorithm than `gzip` or `bzip2`, it is also far more costly, especially on long files, taking over 18 minutes to compress RFC 2300 against itself, in comparison to less than a second for `gzip` and `bzip2`. Even with the fastest compressors, comparing a file against a registry of many files is out of the question.

5 Comparison to other methods

Some proposed methods for pairwise overlap detection are quite different from our approach. For example, JPlag [Prechelt et al., 2000] is a web service for plagiarism detection among programs. It attempts a full comparison of any pair of programs by tiling the token stream of one with maximal substrings of the token stream of the other. The algorithm is similar to that of diff, except that tokens are used instead of lines. While tokenisation compresses programs considerably and also results in the formation of equivalence classes (for example all identifiers become a single token), the remaining task is still compute-intensive in general.

Other proposed methods for web-based overlap detection are related to our SE method. They all divide the document into chunks by some kind of chunking and then select a representative set of chunks. They use different chunking and culling strategies.

SCAM [Shivakumar and Garcia-Molina, 1996] accepts various chunk sizes, ranging from the finest grain (individual words) to the coarsest (the entire document). We adopt SCAM's hashed-breakpoint chunking strategy. SCAM applies no culling; instead, it stores all chunks in the database. A SCAM database therefore requires 30−60% of the size of the document set, compared to approximately 5% for SE.

Koala [Heintze, 1996] uses overlapping chunks of 20 consecutive consonants and proposes two different culling strategies: (1) Retain a fixed number of chunks for each document. (2) Compute the relative frequency of each chunk and retain least-frequent chunks. Overlapping chunks lead to a large amount of data, as we have seen in Section 4.1. Furthermore, Koala does not use natural chunk boundaries, such as word boundaries.

Shingling [Broder et al., 1997] considers 10-word overlapping chunks. It retains chunks based on Rabin's fingerprint [Rabin, 1981]. Shingling has the advantage that it only considers semantically delimited chunks, that is, those that starting at the beginning of a word, but overlapping can lead to a large number of chunks. The culling strategy can retain consecutive chunks, the second of which only provides marginal additional information.

Sif [Manber, 1994] considers 50-byte overlapping chunks. It delimits with bytes instead of words because it is aimed at a wider class of documents than text files. Culling is based on anchors around which chunks are built. An anchor is an arbitrary byte sequence, such as "ante". The problem of anchors is finding the right anchor for a given text and using the same anchor when analysing a potentially similar text.

Our SE method is similar in spirit to these methods, but our culling methods have superior space requirements. Unfortunately, we do not have access to the document sets used by other researchers, but we do compare our results to a those generated by MDR, which is based on exact comparison of texts. This comparison shows that our culling does not sacrifice accuracy.

RFC 1	RFC 2	MDR 1	MDR 2	gzip	bzip2	ppm*
1596	1604	99	99	*4	79	75
1048	1084	94	91	89	70	73
1084	1395	86	84	87	68	72
1600	1410	72	77	*3	58	61
2497	2394	19	17	27	26	28
2422	2276	18	3	6	9	10
2392	2541	16	12	22	20	22

Table 4: Similarities: MDR, gzip, ppm*

6 Conclusions

Our results show that signature extraction (SE) for detecting document overlap is effective. Our algorithm is based on (1) hashed-breakpoint chunking, (2) culling by the variance method, (3) retaining only 10 hex digits of the MD5 digest, (4) storing in a Perl database, (5) computing symmetric similarity. Although each of these components has reasonable alternatives, our SE implementation is fast and accurate enough for initial checking of documents for possible plagiarism. Those document pairs that appear suspiciously similar can be further examined by other methods such as MDR. SE is also a potential tool for a search engine, particularly one that can direct the reader to other documents in a repository that are similar to a given one.

Our SE method is available for testing at `http://www.csse.monash.edu.au/projects/plague/registry.shtml`. The Perl module for k-d trees is available at `ftp://ftp.cs.uky.edu/cs/software/kd.pm`.

References

[Argetsinger, 2001] Argetsinger, A. (2001). Technology exposes cheating at U-Va. *The Washington Post*.

[Arya et al., 1994] Arya, S., Mount, D. M., Netanyahu, N. S., Silverman, R., and Wu, A. (1994). An optimal algorithm for approximate nearest neighbor searching. In Sleator, D. D., editor, *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 573–582, Arlington, VA.

[Benjaminson, 1999] Benjaminson, A. (1999). Internet offers new path to plagiarism, UC-Berkeley officials say. *Daily Californian*.

[Broder et al., 1997] Broder, A. Z., Glassman, S. C., Manasse, M. S., and Zweig, G. (1997). Syntactic clustering of the Web. *Computer Networks and ISDN Systems*, 29(8–13):1157–1166.

[Cleary and Teahan, 1997] Cleary, J. and Teahan, W. J. (1997). Unbounded length contexts for PPM. *Computer Journal*, 40(2/3):67–75.

[Deutsch, 1996] Deutsch, L. P. (1996). *RFC 1952: GZIP file format specification version 4.3*. Internet Activities Board.

[Friedman et al., 1977] Friedman, J. H., Bentley, J. L., and Finkel, R. A. (1977). An Algorithm for Finding Best Matches in Logarithmic Expected Time. *ACM Transactions on Mathematical Software*, 3(3):209–226.

[Google, 2001] Google (2001). http://www.google.com.

[Heintze, 1996] Heintze, N. (1996). Scalable document fingerprinting. In *Proceedings of the second USENIX Workshop on Electronic Commerce: November 18–21, 1996, Oakland, California*, pages 191–200, Berkeley, CA, USA.

[Manber, 1994] Manber, U. (1994). Finding similar files in a large file system. In *Proceedings of the Winter 1994 USENIX Conference: January 17–21, 1994, San Francisco, California, USA*, pages 1–10, Berkeley, CA, USA.

[Monostori et al., 2000] Monostori, K., Zaslavsky, A. B., and Schmidt, H. (2000). Document overlap detection system for distributed digital libraries. In *Proceedings of the 5th ACM Conference on Digital Libraries (DL-00)*, pages 226–227, New York, NY.

[Prechelt et al., 2000] Prechelt, L., Malpohl, G., and Philippsen, M. (2000). Finding plagiarisms among a set of programs with JPlag. Submitted to Journal of Universal Computer Science; http://wwwipd.ira.uka.de/jplag.

[Rabin, 1981] Rabin, M. O. (1981). Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University.

[Rivest, 1992] Rivest, R. L. (1992). *RFC 1321: The MD5 Message-Digest Algorithm*. Internet Activities Board.

[Seward, 2000] Seward, J. (2000). http://sources.redhat.com/bzip2.

[Shivakumar and Garcia-Molina, 1996] Shivakumar, N. and Garcia-Molina, H. (1996). Building a scalable and accurate copy detection mechanism. In *Proceedings of the 1st ACM Conference on Digital Libraries (DL'96)*, Bethesda, Maryland.

[Wall and Schwartz, 1992] Wall, L. and Schwartz, R. L. (1992). *Programming Perl*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA.