

## Efficiency of Data Structures for Detecting Overlaps in Digital Documents

Krisztián Monostori

Arkady Zaslavsky

Heinz Schmidt

*School of Computer Science and Software Engineering*

*Monash University, Melbourne*

*AUSTRALIA*

*E-mail: {krisztian.monostori, arkady.zaslavsky, heinz.schmidt}@infotech.monash.edu.au*

### Abstract

*This paper analyses the efficiency of different data structures for detecting overlap in digital documents. Most existing approaches use some hash function to reduce the space requirements for their indices of chunks. Since a hash function can produce the same value for different chunks, false matches are possible. In this paper we propose an algorithm that can be used for eliminating those false matches. This algorithm uses a suffix tree structure, which is space consuming. We define a modified suffix tree that only considers chunks starting at the beginning of words and we show how the algorithm can work on this structure. We can alternatively reduce space requirements of a suffix tree by converting it to a directed acyclic graph. We show that suffix link information can be preserved in this new structure and the matching statistics algorithm still works with those modifications that we propose.*

### 1. Introduction

Digital libraries provide vast amounts of digitised information on-line. Preventing these documents from unauthorised copying and redistribution is a hard and challenging task, which often results in not putting valuable documents on-line [7].

Copy-prevention mechanisms include distributing information on a separate disk, using special hardware or active documents [8]. We believe that these approaches are very cumbersome for genuine users, therefore copy-detection approaches are more practical. Copy-detection does not try to hinder the distribution of documents but rather tries to identify illegal copies.

One of the most pressing areas of copy-detection applications is detecting plagiarism. With the enormous growth of the information available on the Internet users have a handy tool for writing research papers. With the numerous search engines users can easily find relevant articles and papers for their research. These documents are available in electronic form, which makes plagiarism achievable by cut-and-paste or drag-and-drop operations.

Academic organizations as well as research institutions are looking for a powerful tool for detecting plagiarism. There are well-known reported cases of plagiarism. Garcia-Molina et al. [10] report on a mysterious Mr.X who has submitted papers to different conferences, which were absolutely not his work and Kock [13] describes how he found out that someone – named Mr. Plag in the paper – plagiarised one of his papers and submitted it to a journal.

Copy-detection tools aim to find whole and partial copies of documents either on the Internet or in local repositories. These tools can be used by digital libraries to find copies of their copyrighted documents redistributed on the Web or in newsgroups. The tools can also detect possible plagiarised documents, since textual overlap may prove to be plagiarism. Though there are more “sophisticated” ways to plagiarise, we believe that finding textual overlap would identify most cases of plagiarism. As Kock [13] described in his paper, plagiarists may change numbers, figures, and some geographical names but the two texts would still have substantial textual overlap.

This paper is structured as follows. In Section 2 we discuss existing copy-detection schemes and point out how our suffix tree-based algorithm could complement those approaches. Section 3 introduces suffix trees and the matching statistics algorithm and describes how we can make use of the inherent structure of documents and reduce the size of suffix trees. Section 5 introduces another possible way to save space by converting the suffix tree to a directed acyclic graph (DAG) and we will show how our algorithm works on this structure. In Section 6 we describe what we have achieved so far and what is left for future work.

### 2. Related Work

Copy-detection schemes have two fundamental approaches: digital watermarking, and string comparison. The target of digital watermarking is to find illegal copies and track down the user who purchased the electronic document and redistributed it.

In watermarking methods undetectable codewords are placed in documents similar to the method of placing watermarks in bank notes. These codewords can represent a unique document identifier and this identifier can be assigned to a given customer who purchased the document. Codewords can be hidden in the document by slightly altering the layout of the text. These alterations must be reliably decodable yet not perceptible to the user.

Brassil et al. [4] study three different methods: line-shift coding, word-shift coding, and feature coding. Line-shift coding vertically shifts the locations of text lines. Word-shift coding horizontally shifts words within their lines. It makes use of the fact that most of the texts are left- and right-justified and the space between words is calculated by some algorithm. Feature coding alters the actual bitmap representation of a given character depending on the codeword. These methods also study decoding performance in the presence of noise. Noise in their case is the alteration of the text layout because of the inaccuracy of scanners and photocopiers. We consider digital watermarking as a supplementary tool in copy-detection, which is useful when finding the originator of the illegal copy. Digital watermarking works at a coarse granularity and it would be hard to detect smaller overlapping chunks, such as sentences, or paragraphs.

In string-comparison based algorithms we can define the basic problem as with given two strings T and P with the length of m and n respectively, we have to divide T and P into substrings  $T=t_1s_1t_2s_2t_3s_3\dots t_k s_k t_{k+1}$  and  $P=p_1q_1p_2q_2p_3q_3\dots p_r q_r p_{r+1}$  where for each  $s_y$  there is an  $x$   $s_y=q_x$  and  $\Sigma(|s_y|)$  is maximal. As an example compare  $T=abcbdacda$  and  $P=aabaca$ . One possible partition is  $T=() (ab) (cdb) (c) (adc) (a) ()$  and  $P=() (a) () (ab) (a) (c) (a)$ . T is partitioned into  $t_1=()$ ,  $s_1=(ab)$ ,  $t_2=(cdb)$ ,  $s_2=(c)$ ,  $t_3=(adc)$ ,  $s_3=(a)$ ,  $t_4=()$  and P is partitioned into  $p_1=()$ ,  $q_1=(a)$ ,  $p_2=()$ ,  $q_2=(ab)$ ,  $p_3=(a)$ ,  $q_3=(c)$ ,  $p_4=(a)$ .  $s_1=q_2$ ,  $s_2=q_3$ ,  $s_3=q_1$ .  $\Sigma(|s_x|) = 3$ , which is not maximal in this case because the partition giving the maximum for  $\Sigma(|s_x|)$  is  $T=() (ab) () (c) () (b) () (ca) (d) (ca) ()$  and  $P=() (a) () (ab) (a) (ca) ()$ . The maximum overlap is a single value, but different partitions can result in the same maximal overlap value, that is, the partition is not unique.

We analysed four research prototypes that aim to solve the above-defined maximal overlap problem. These research prototypes include the SCAM (Stanford Copy Analysis Method) system [8], the Koala system [12], the “shingling approach” of [5], and the file system clustering method of the sif tool [14]. The general approach to define the maximal overlap is to split up the text into chunks, calculate some hash function on those chunks, select certain chunks from all the chunks to be stored in an index. When a document has to be compared to documents in the repository, the suspicious document, that we want to compare to others, has to be chunked using the

same strategy as used in registration. The overlapping chunks then must be identified. Given these chunks, a decision function decides whether a given document is plagiarised, is a partial or exact copy of other documents depending on the objective of the system. In the following subsections we compare these systems, based on the chunking primitives they use, how they select chunks to keep in the repository, and their decision functions.

## 2.1. Possible Chunking Primitives

The SCAM project [9] studied different chunking primitives and compared them based on their accuracy, performance, and security, that is how hard it is to fool the system by inserting some extra words, or changing sentences, etc. The finest granularity of overlap is character-overlap while the other extreme is to consider the whole document as a chunk. In this case we can only identify exact document copies and partial copies cannot be identified. Of course, two documents that both have all the characters in the English alphabet would produce a maximal overlap of the length of the document in case of character-overlap.

We define a threshold for the shortest character sequence that we consider overlap. The Koala system [12] uses 20 consonants, which translates into 30-45 characters. The “shingling approach” [5] considers 10 consecutive words, which is approximately 50-60 characters. The sif tool [14] uses 50 bytes (they use bytes rather than characters since their main focus is not only textual files but binary files, too). The SCAM system [9] studies words, 5 consecutive words (app. 25-30 characters), 10 consecutive words (app. 50-60 characters), sentences, and the whole document. Our algorithm, which is detailed in the next section, uses 60 character chunks.

It is obvious that the finer is the granularity the greater could be the number of false matches. That is two documents, which are identified as related, in fact could be quite different. On the other hand the coarser the granularity the greater the chance to miss some documents (false negatives). The empirical value between 40 and 60 characters looks to be a good compromise.

These systems also differ in the way, how chunks overlap. This issue is important because simply dividing the text into say 60-character long non-overlapping chunks is not enough. Chunk boundaries can easily be shifted by inserting or removing a word. In the SCAM system, a hashed breakpoint chunking is introduced, which identifies chunk boundaries based on a hash value calculated for each word. Manber [14] proposes to use strings as anchors, and build 50-byte chunks around those anchors. The problem with using anchors is that it is hard to find a set of anchors that achieves uniform distribution regardless of the subject of the text. In case of sentence chunking, the sentence boundaries are the natural anchors,

however as it is pointed out in [9], sentence boundaries are not always obvious to detect. All other approaches consider overlapping chunks and they select only a portion of them to store based on an algorithm described in the next subsection.

There is another natural boundary in documents, namely word boundary. We may not be interested in overlapping chunks that start in the middle of a word. The “shingling approach” [5] uses 10-word chunks, and we also make use of this feature of natural language texts in our algorithm.

## 2.2. Selection Strategy

Heintze [12] points out that the space-requirement for full fingerprinting is too large and some selection method must be used to keep a representative fingerprint of the possible chunks in a document. Full fingerprinting requires less space in the case of the “shingling approach” [5], which makes use of word boundaries. If we only consider chunks starting at the beginning of words then the number of possible chunks (full fingerprinting) is in the order of the number of words rather than the number of characters. However, we have to note that the sif tool [14] targets a broader area of copy-detection and they consider binary files besides text files. In case of binary files we do not have such natural chunk boundaries as word boundaries in text files.

The selected chunks are called the fingerprint of the document, because they represent the whole document. The “shingling approach” [5] and the sif tool [14] use Rabin’s fingerprinting algorithm to select representative fingerprints. Heintze [12] proposes to keep chunks with the least hash values. The SCAM system [9] keeps all chunks except for stopwords in case of word chunking but as expected their index is 30-60% of the size of the original documents, which is prohibitive in such large collections as the Internet.

Another issue when selecting representative fingerprints is whether to have a fixed number of chunks for each document or let the number of chunks be proportional to the size of the document [12][5]. In case of fixed-size fingerprints it is hard to define whether a given document is contained in another document, one of the crucial problems in copy-detection.

## 2.2. Decision Function

Given the overlap between two documents we have to decide whether the documents are related or not. The decision function for this problem is usually a simple threshold. If  $V(A)$  is the fingerprint of document A, that is a representative set of chunks in document A, and the same holds for B then our resemblance measure is:

$$(1) \text{ Resemblance}(A, B) = \frac{|V(A) \cap V(B)|}{|V(A) \cup V(B)|}$$

In the SCAM system [9] when words are the chunking primitives this resemblance measure is not descriptive because similar documents do not necessarily share the same number of postings for each word. Garcia-Molina et al. [9] propose a measure that is similar to the cosine measure [3] used in information retrieval applications.

## 2.4. Reducing False Positives

The main problem of the approaches analysed in the previous subsection is that they use a hash function on chunks. Given the large number of possible chunks and the limited space to store a representative number for that chunk it is probable that different chunks produce the same hash value. The probability increases if we index more and more documents. As a result these approaches report a certain number of false positives [9], which should be eliminated.

In this paper we propose an algorithm, which is based on Chang’s matching statistics [6] algorithm that eliminates these false positives. This algorithm uses a suffix tree, which is a very space consuming data structure, thus it can only work on a limited number of documents, such as documents reported by one of the methods described above. The documents identified in the first phase are called candidate documents and our algorithm compares the suspicious document to the candidate documents.

## 3. Identifying Overlap with the Matching Statistics Algorithm

This section introduces suffix trees and describes how the matching statistics algorithm can be used to identify overlap and in Section 4 we show how a suffix tree can be converted into a directed acyclic graph and how the matching statistics algorithm can be used on that structure.

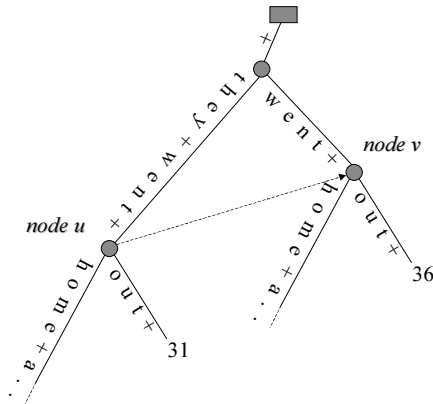
### 3.1. Suffix Trees

Suffix tree is a data structure, which represents all suffixes in a string. Suffix trees are also referred to as position trees, subword trees and complete inverted files [1][2]. Apostolico [2] gives a brief overview of straightforward applications of suffix trees: multiple search of different patterns in a fixed set, finding the first occurrence of  $w$  in  $x$  where both  $w$  and  $x$  are strings, finding all occurrences of  $w$  in  $x$ , finding the number of



T than  $v$  has a label  $\beta$ . This definition is similar to the original definition but it considers that only suffixes starting at beginning of words are represented in the tree.

Inherently the matching statistics algorithm can use the suffix links of our modified suffix tree. We can use the same heuristic as we used on the construction of the modified suffix tree. We start matching only from beginnings of words (recall our definition of word) and when we finish traversing a path due to a mismatch, we can follow the suffix links and continue matching from that node where we jumped to because having found a matching chunk of  $\alpha\beta$  we will follow by matching  $\beta$ .



**Fig2. Modified suffix tree**

As an example let us suppose that we want to compare the string ‘+They+went+home+before+midnight’ to the suffix tree of figure 2. We traverse down the tree until we find ‘+They+went+home+’, which gives a score 16 for the matching statistic at *0th* position. Then we have to find the matching statistic value at position 5, that is the beginning of the next word. We follow the suffix link, which places us on the path of ‘+went+home+’, so we can continue matching from this position, which in our case is impossible, so we set the matching statistics for the *5th* position to 11.

With this representation, the size of the suffix tree can be reduced by the ratio of the average word size in the document (approximately 5 - 6 times). The building algorithm also benefits from this size reduction and we are still able to use the matching statistics algorithm, which is also accelerated because of neglecting the middle of words.

### 3.3. Performance Analysis

We tested our algorithm on a test machine with the following configuration: Intel Pentium II 433MHz, 128M RAM, Windows NT Workstation. Below, the 3 common scenarios are described along with performance details of

our algorithm.

**Scenario 1:** the suspicious document is genuine. There can be accidental overlaps but they do not add up above a given percentage (10%).

**Scenario 2:** the suspicious document is heavily plagiarised, that is 60-70% percent of the document is copied from 8 different documents and the order of the chunks is mixed up.

**Scenario 3:** the most part of the suspicious document is genuine, but there is a huge chunk (1 page), which is copied from another document.

The files in scenario 1 and 2 are 11K and 14K respectively, while the file in scenario 3 is a 1.67M document. We compare these three documents to 19 documents, which are 3.5M altogether and of course contain those 8 documents, which are used in Scenario 2 and the one used in Scenario 3. We used the modified algorithm for analysing the documents. This algorithm correctly reported the files in Scenario 2 and 3 as plagiarised while the document in Scenario 1 as genuine.

The running time for calculating matching statistics in Scenario 1 and 2 were almost the same (see table 1). But Scenario 3 took more time. It contradicts with the linear time bound of the matching statistics algorithm, which states that the running time of the algorithm is independent from the size of the tree and only depends on the strings compared to the tree, which are the candidate-documents of 3.5M in our case. We found two issues, which influence the running time of the algorithm in practice. Firstly, the suffix tree in Scenario 1 and 2 can fit into the cache, which considerably speeds up the algorithm. Secondly, it is true that the so-called skip/count steps [11] are bounded by 3m but the bigger the tree the more the skip/count steps, which adds up to the running time. It is still true that the algorithm has linear time worst-case bound but in reality the case is worse if the tree is bigger.

**Table 1. Performance analysis details**

Scenario	Size	msi	msi <sub>overall</sub>
1	309,192	530	7
2	228,716	531	6
3	41,573,992	1923	-

Legend:

Size: Size of suffix tree (bytes)

msi: Calculating msi by document (ms)

msi<sub>overall</sub>: Calculating overall msi (ms)



suffix link in the suffix tree? If there is a suffix link in the suffix tree representation between *node p* and *node q* and *node p* and *node q* are merged in the conversion process it means that their subtrees are isomorphic. During the matching statistics algorithm we follow the suffix link to find the value of  $ms(i+1)$  having found  $ms(i)$  but if the two subtrees are isomorphic then it is sure that no further matches will be found, so  $ms(i+1)$  is simply  $ms(i)-1$ . We know that we are in a subtree that was previously connected by suffix links if there is an offset on the path that we followed from the root. The sum of the offsets determines how many suffix links we could have followed without finding a different subtree, so if this value is  $x$  we already know the matching statistics  $ms(i+1)$  through  $ms(i+x)$ . After that we follow the algorithm as if we have already defined  $ms(i+x)$  and we would like to define  $ms(i+x+1)$ .

It is also possible that after merging *node p* and *node q* some suffix links under *node p* and *node q* are lost. These suffix links are not needed at all in the DAG representation because traversing further down from *node p* will place us under *node q* and this case is equivalent to the case discussed above.

There is one more case that we have to discuss. It is possible that the destination of a suffix link is removed from the tree. There is still an equivalent node of that destination node but they are only equivalent regarding their subtrees. If we divert the suffix link to that equivalent node we have to check how it affects the matching statistics algorithm. We can place an offset on the suffix link similar to the one we place on edges. The offset value is the number of suffix link hops between the original destination node and the equivalent node. Following a diverted suffix link is equivalent to following the original suffix link in the original suffix tree then following the suffix link from that node etc. Matching statistics will always increase by one on the way because subtrees are equivalent. If we place offset on suffix links the algorithm is the same as the algorithm with the offset on edges.

After discussing how suffix links are converted we describe how our algorithm works in practice. In the first step it makes a depth-first traversal to identify an equivalent node for each node if any. This can be achieved in linear time. After this suffix links are diverted. If a suffix link points to a node, which has an equivalent node the suffix link will point to the equivalent node with an offset of 1. It is also possible that the equivalent node has an equivalent node, too. In this case the offset is incremented by one and the suffix link is diverted to the latter node. This process can be followed until a node is found without an equivalent node. Normal edges can be diverted in the same way as suffix links the only difference is that if an edge is diverted the subtree of the node that was originally pointed by the edge has to be

removed. Instead of removing it immediately we add this node to a list of nodes to be removed. The reason for this is that by removing a node we might break a path of equivalent nodes and at a stage we might address some invalid memory range. After all edges and suffix links are diverted we have to remove subtrees of nodes in the list. The pseudo code for the algorithm:

```

Find number of leaves under each node
For each node
    Identify equivalent node
End For
For each node
    Divert suffix link to equivalent node
    Define offset value for suffix link
    For each edge running out of current
        node
            If equivalent exist
                Divert edge
                Define offset value for edge
                Add next node to remove_list
            End If
        End For
    End For
End For
For each node in the remove_list
    Remove subtree
End For

```

## 5. Conclusion and Future Work

In this paper we have described a copy-detection algorithm that can be used as a final filter on existing copy-detection algorithms that have a problem of reporting many false positives. This algorithm uses a suffix tree structure, which is space-consuming. We proposed two ways of reducing the space requirement. One of them is considering word boundaries and only inserting suffixes that start at the beginning of words and the other one is to convert the tree into a directed acyclic graph. We showed how the matching statistics algorithm works on these structures and how it is possible to preserve suffix links, which are crucial for the matching statistics algorithm.

In the future we will test how these two approaches could be merged. We will study whether it is possible to convert the modified suffix tree into a modified directed acyclic graph. We are also developing a prototype system MatchDetectReveal (MDR) [15], which will implement an algorithm for identifying candidate documents and the result set will be fed to the matching engine using suffix trees and directed acyclic graphs. Other components of the system have already been developed including the visualiser-component, and an enhanced version of the matching engine that uses a local cluster for parallel comparison of documents.

## Acknowledgement

Support from Distributed Systems Technology Centre (DSTC Pty Ltd) for this project is thankfully acknowledged.

## References

- [1] Aho A. V., Hopcroft J. E., Ullman J. D. The Design and Analysis of Computer Algorithms (Addison-Wesley, Reading, MA, 1974).
- [2] Apostolico A. The Myriad Virtues of Subword Trees, in A. Apostolico and Z.Galli. Combinatorial Algorithms on Words. pp. 85-96, Springer-Verlag Heidelberg, 1985.
- [3] Baeza-Yates R., Ribeiro-Neto B., Navarro G. Indexing and Searching in Baeza-Yates R., Ribeiro-Neto B. Modern Information Retrieval, ACM Press, 1999.
- [4] Brassil J., Low S., Maxemchuk N., and O'Gorman L. Technical report, AT&T Bell Laboratories, 1994. URL <ftp://ftp.research.att.com/dist/brasil/docmark2.ps>.
- [5] Broder A.Z., Glassman S.C., Manasse M.S. Syntactic Clustering of the Web. Sixth International Web Conference, Santa Clara, California USA. URL <http://decweb.ethz.ch/WWW6/Technical/Paper205/paper205.html>
- [6] Chang W.I., Lawler E.L. Sublinear Approximate String Matching and Biological Applications. Algorithmica 12. pp. 327-344, 1994.
- [7] Garcia-Molina H., Shivakumar N. The SCAM Approach To Copy Detection in Digital Libraries. D-lib Magazine, November, 1995.
- [8] Garcia-Molina H., Shivakumar N. SCAM: A Copy Detection Mechanism for Digital Documents. Proceedings of 2nd International Conference in Theory and Practice of Digital Libraries (DL'95), June 11 - 13, Austin, Texas.
- [9] Garcia-Molina H., Shivakumar N. Building a Scalable and Accurate Copy Detection Mechanism. Proceedings of 1st ACM International Conference on Digital Libraries (DL'96) March, Bethesda Maryland, 1996.
- [10] Garcia-Molina H., Shivakumar N. SCAM: A Copy Detection Mechanism for Digital Documents. Proceedings of 2nd International Conference in Theory and Practice of Digital Libraries (DL'95), June 11 - 13, Austin, Texas, 1995.
- [11] Gusfield D. Algorithms on Strings, Trees, and Sequences. Computer Science and Computational Biology. Cambridge University Press, 1997.
- [12] Heintze N. Scalable Document Fingerprinting. Proceedings of the Second USENIX Workshop on Electronic Commerce, Oakland, California, 18-21 November, 1996. URL <http://www.cs.cmu.edu/afs/cs/user/nch/www/koala/main.html>
- [13] Kock. N. A case of academic plagiarism. Communications of the ACM, July 1999, Vol. 42, No.7, p.96-104, 1999.
- [14] Manber U. Finding similar Files in a Large File System. Proceedings of the 1994 USENIX Conference, pp.1-10, 1994.
- [15] Monostori K., Zaslavsky A., Schmidt H. MatchDetectReveal: Finding Overlapping and Similar Digital Documents, Information Resources Management Association International Conference, 21-24 May, 2000, Anchorage, Alaska, USA
- [16] Monostori K., Zaslavsky A., Schmidt H. Parallel Overlap and Similarity Detection in Semi-Structured Document Collections. 6th Annual Australasian Conference on Parallel And Real-Time Systems (PART '99)
- [17] Ukkonen E. On-Line Construction of Suffix Trees. Algorithmica 14. pp 249-260, 1995.