# A Study of Self-Plagiarism in Computer Science

University of Arizona TR03-02

**Christian Collberg** **Stephen Kobourov** **Joshua Louie**
**Thomas Slattery**
Department of Computer Science,
University of Arizona, Tucson, AZ 85721.
{collberg,kobourov,jdlouie,thomass}@cs.arizona.edu

## Abstract

We present a web spider that crawls through the web sites of the top fifty Computer Science departments, downloading research papers to search for instances of self-plagiarism by Computer Science professors. Instances of self-plagiarism for each author are reported so that they may be investigated in order to determine if they are truly fraudulent papers.

## 1 Introduction

Self-plagiarism is the use of ones own previously published materials in the creation of a new published material without crediting the previous paper as a source. The use of self-plagiarism allows for a bloated number of research papers to be produced without doing additional work to create new papers. As a result, fundamentally identical papers can be created and passed off to different journals all for the purpose of increasing the academic recognition of the researcher. However, such practices do not benefit the research society as a whole, in that many more papers are produced, with less new and exciting material to spur on new ideas. Instead the pool of papers becomes cluttered with papers on the same topics, but with different names.

The purpose of this experiment is to find out if there are any professors at top Computer Science universities that engage in this practice. The basic concept is to run a web spider to traverse the top 50 computer science departments and find the faculty pages. For each faculty member, download each professor's papers. After converting them to text, run a text analysis program to check for self-plagiarism and report any offending professors and their papers. Those reported would have to be checked by hand to ensure that the similarities are, in fact, due to academic dishonesty. See Figure 1.

## 2 Related Work

CORA, a Computer Science Research Paper Search Engine [2], most closely resembles the type of spider that we are using. CORA made use of smart spiders to crawl computer science web sites and record the papers located within. The major difference between that spider and the one we are using is that our spider is designed to dynamically search for one professor's work at a time, and not crawl the entire site before hand to get all the information first.

The Stanford Copy Analysis Mechanism (SCAM) [3], is a comparison utility for detecting identical documents or documents with a high degree of overlap. SCAM uses a registration server to which original documents can be registered by their authors. Attempts to register illegal copies of already registered documents can be detected. Additionally, web crawlers can be used to search for documents and compare them against registered documents in a manner somewhat similar to our system.

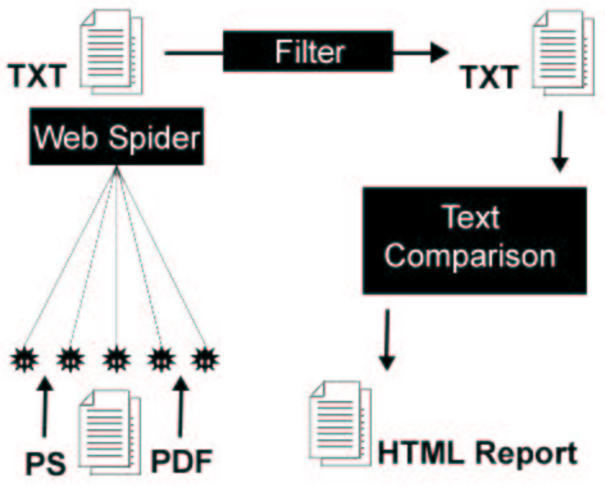


Figure 1: Overview of the system.

## 3 The Web Spider

The programming language WebL [1] was used in the development of the web spider. WebL's manual defines itself as a "language and system is designed for rapid prototyping of Web computations. It is well-suited for the automation of tasks on the WWW1." It has modules to enable the programmer to quickly develop a web spider. The production of a web spider that successfully locates professors' pages and downloads the papers has to be constrained heavily to escape several major problems that come when looking for papers:

**Remaining on the professor's page:** When examining a page, what is an acceptable page to go to and what is not?

**Research papers only:** When preparing to download a file, it there anyway to know if it is a research paper or not?

**Seemingly infinite link graph:** When traversing links on a single professor's web site, when should the spider give up?

**Slow downloads:** If downloading a 30Mb paper, with a transfer rate of less than 1Kb, should the spider bother downloading it?

Ideally speaking, a web spider would start at the homepage of the professor and traverse all links downloading all the research papers for that professor. If seen as graph problem, this would be feasible if and only if the graph had one source, the professor's home page, and no edges going outside of his page. In the real web, a professor's page has many links that extend to other pages ranging from the classes he teaches to interesting pages all over the web. Attempting to try all links would result in the spider going far away from its target area and the time to process one professor could be seemingly infinite along with downloading hoards of irrelevant papers.

Several constraints were placed to limit where the spider could go. The first was to check for links containing the words *publication*, *paper*, or *research*. These links would most likely have the desired materials. If no links could be found with any of the key words, then searching would be done in a breadth-first search through all of the professor's links. Another limitation put on the spider was that the only links that could be traversed were ones that existed on the same main site. That is, if the professor were at `http://cs.university.edu/~professor`, the spider would only check sites whose links contained the name *university*. Attempting to further constrain this to require the link to contain possibilities such as `cs.university.edu` or `~professor` suffer many losses since some professors work in multiple departments, and others don't place all their pages hierarchically under their initial home page.

The spider also allows a single level below the home page of checking as well. While this limits the sites that the spider can go to, it still has the possibility of entering other professors' sites in the same university. However, this is hard to prevent, as there is no distinctive mark that determines the owner of a page.

Many papers are located on the professors' web sites in usual formats of `.pdf`, `.ps`, and `.doc`. Instead of attempting to determine if a paper is a research paper or not, the spider simply downloads them all. After downloading all the files and converting them to text, a filter program is run to remove all files that did not convert properly and those that are not research papers. The rule for research papers is that they must contain an abstract or introduction, and also a references or bibliography section. This process was the same as used with CORA to find research papers, and experiments showed that it had roughly a 95% accuracy rate. Attempts to discern from the web whether or not papers are research papers is limited since there are no distinguishing features in the link or in the paper name as to the nature of the paper.

Another issue with the web is the fact that a single professor's site may contain many levels of pages, which results in hundreds and possibly thousands of checks and page loads in search of his papers. The spider is set with a timer that records when a professor's site is first being checked. Each time the spider gets ready to download a paper, search a new page or visit another page, it checks the time. If the time is too long, it stops downloading, instead finding new pages and visiting them. The issue of downloading a large paper through a small connection and wasting large amounts of time is hard to check. The web spider thus has the unfortunate problem that it has to wait for the files to be downloaded, and this might waste the time allocated to searching the particular professor's web site. The only safety against this is that the web spider has a collection of threads doing the crawl. Thus, with a few threads, if one gets stuck on a big file being downloaded on a slow connection, the others can continue searching for additional papers.

## 4 Text Comparison

The text comparison utility, which was implemented in Java, is a completely independent module from the web spider. After the web spider has finished downloading and converting all of the papers for a particular professor, the text comparison utility is executed on the directory containing those papers. The utility performs pairwise comparison of all papers in the directory. When finished, it produces an HTML report file in the same directory. The report file lists in descending order all pairs of files with their respective percentages (above an adjustable threshold) of detected similarity. See Figure 2.

### 4.1 Comparison Algorithm

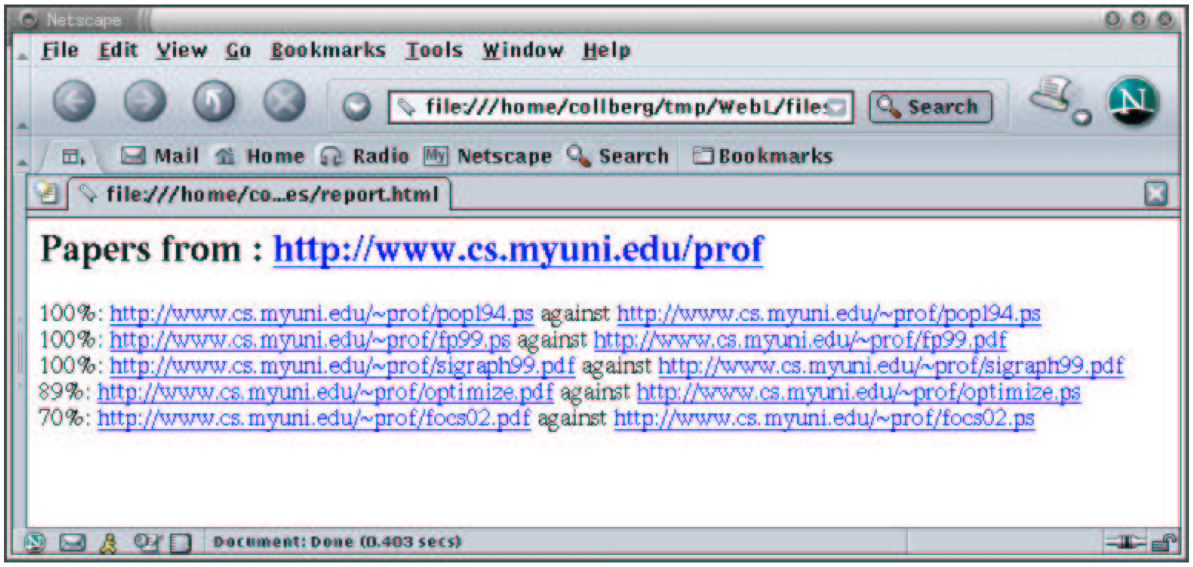
One of the first things to consider was the question of what exactly constitutes plagiarism. Any block of directly copied text is obviously plagiarism, but there are many other cases to consider:

**Cosmetic changes:** Minor cosmetic changes to text, such as the addition or removal of punctuation or spacing should not affect the comparison.

**Reordered text:** Paragraphs or sentences from paper $A$ can be copied but placed in a different order in paper $A$'. If the bulk of the content is the same, however, this should still be detected.

**Reworded text:** Rewording of text without significant change to the meaning, such as the substitution of a few words for synonyms or swapping clauses of a sentence, should be caught. While not nearly as severe as directly copying text, a significant amount of this should still register as plagiarism.

In addition to worrying about what sorts of similarities amount to plagiarism, there are also an almost endless number of criteria upon which two texts can be compared to detect violations. The primary goal of the utility was accuracy in detecting instances of plagiarism, but a certain degree of efficiency was necessary as well. Papers in plaintext format in excess of 200kb were not uncommon, but even the pairwise comparison of 50+ average-sized documents (≈50kb) is a very lengthy $O(n^2)$ operation. More


Netscape
File Edit View Go Bookmarks Tools Window Help
file:///home/collberg/tmp/WebL/files
Search
Mail Home Radio Netscape Search Bookmarks
file:///home/co...es/report.html
Document: Done (0.403 secs)

Papers from : http://www.cs.myuni.edu/prof

100%: http://www.cs.myuni.edu/~prof/popl94.ps against http://www.cs.myuni.edu/~prof/popl94.ps
100%: http://www.cs.myuni.edu/~prof/fp99.ps against http://www.cs.myuni.edu/~prof/fp99.pdf
100%: http://www.cs.myuni.edu/~prof/sigraph99.pdf against http://www.cs.myuni.edu/~prof/sigraph99.pdf
89%: http://www.cs.myuni.edu/~prof/optimize.pdf against http://www.cs.myuni.edu/~prof/optimize.ps
70%: http://www.cs.myuni.edu/~prof/focs02.pdf against http://www.cs.myuni.edu/~prof/focs02.ps

Figure 2: Reports are presented in a standard browser.

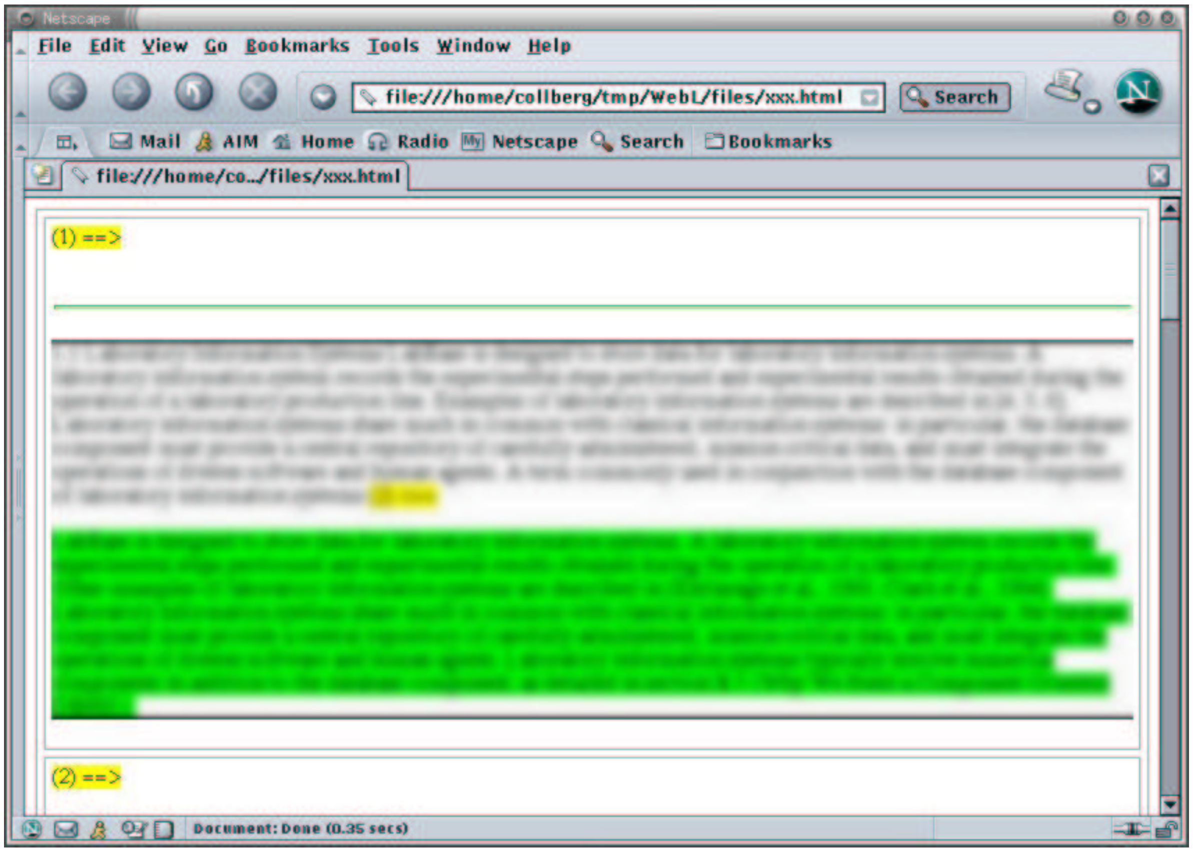

Netscape
File Edit View Go Bookmarks Tools Window Help
file:///home/collberg/tmp/WebL/files/xxx.html
Search
Mail AIM Home Radio Netscape Search Bookmarks
file:///home/co.../files/xxx.html
(1) ==>
(2) ==>
Document: Done (0.35 secs)

Figure 3: The system allows for two papers to be examined in more detail. Similar paragraphs are color coded and presented in a standard browser.

complex algorithms, such as attempting to align the documents, were considered. However, the highly likely possibility that the text will be reordered renders alignment a relatively poor choice for a primary algorithm. Since plagiarized text from one document could appear anywhere and in any order in a second document, a brute-force comparison algorithm seemed to be the best choice. A great deal of optimization is possible to improve the speed of the algorithm, but a certain degree of brute force is necessary to find all occurrences of plagiarism. The final algorithm consists of three main parts:

1. parsing the text documents into paragraphs and sentences in a canonical form;
2. performing a highly optimized, brute-force, pairwise comparison of the parsed documents; and
3. producing an HTML report of the results.

## 4.2 Parsing into Canonical Form

In the first phase, each text file is parsed into a Document object with a list of Paragraph objects, each of which then has a list of Sentence objects. The text file is expected to contain the URL of the original file on the first line, which is extracted first, followed by a separate paragraph on each line. Each line after the first then becomes a Paragraph. Sentences are considered to be anything delimited by the characters `! . ? ;`. Paragraphs with less than four sentences are discarded, since they are most likely just section headings, parts of formulas, or other random and insignificant text. If a Paragraph has at least four sentences, the resulting text for the Sentences is converted to lowercase and then parsed into words. Words are considered to be anything delimited by whitespace or any of the following characters: `! . ? " \ < > : ; [ ] { } ( ) /` Any whitespace or delimiting characters are removed and only the list of words is retained. Words from a short, pre-defined list deemed insignificant to the meaning of a sentence (such as a, an, the, this, and that) are discarded. Similar to Paragraphs, any Sentence with less than four words (after removing insignificant ones) is discarded. Most "sentences" with less than four meaningful words are not actual sentences–those that happen to be real sentences are too short to hold much content and thus are not of much interest for comparison. Rather than storing enormous numbers of small strings, the words are represented as integers. A global hash of words to their corresponding integer values is maintained across the program. As a sentence is parsed, each word is looked up in the table. If it already exists, that value is used. Otherwise, the word is inserted into the table with a new, unique value. Each Sentence then maintains a list of its unique words (as integers in sorted order); the sum of the integer values of all its words; and the original sentence, as a string in canonical form with a space between each word.

## 4.3 Comparison Algorithm

After all documents have been parsed, all pairs of documents are "scored" for the level of similarities found between them. All pairs of paragraphs in the two documents are scored against each other by a pairwise comparison of their respective sentences. The results then filter up, with paragraphs earning points based upon the number of similar sentences and their levels of similarity and the document earning points based upon the amount of detected plagiarism in each paragraph with a total score above a certain threshold.

Sentences are compared for similarity on two levels. Sentences that are identical earn the maximum score possible; sentences that are highly similar to one another earn a score somewhere between 50-100% of the possible score, depending on the amount of similarity. Comparing sentences for equality is easy and highly optimized. Before even looking at the words in a sentence, the ”sums” of the two sentences–calculated while parsing–are compared. If they are not the same, it is known immediately that the sentences are not identical. While occasional overlap of the values of these sums does occur between sentences that are different, it is rare enough to eliminate almost all unnecessary comparisons. Only if the sums and word counts of two sentences are identical are the actual strings compared.

Sentences are considered similar if the intersection of their sets of unique words is the same size or only slightly smaller than the sets themselves. Since the lists of unique words are maintained in sorted order, binary searching can be used to efficiently calculate the size of the intersection of the sets. This comparison is also optimized to be performed only when significant similarities may exist. Sentences that have a major discrepancy in the sizes of their unique word sets are ignored, as are any sentences that have very small sets of unique words.

### 4.4 Reporting Results

After pairs of papers have been scored against each other, they are given a percentage to represent the approximate level of plagiarism found between the two. The score for a document represents roughly the number of sentences worth of plagiarism found in the document. The percentage divides that number by the average length of the papers and multiplies by 100. Once all pairs of papers have been compared, the list of papers and their respective percentages is sorted in descending order and the results are written out to an HTML file in the directory. The top of the file gives the URL for the root page from which the papers were downloaded. Following that is the list of percentages with links to the corresponding papers in their original format.

## 5 Future Work

In the future, the spider will probably be able to find better papers if the entire computer science site is crawled. All papers would be downloaded and converted. Those would be filtered and then would be parsed and sorted into the appropriate professor's directories. The spider would implement a better schema such as reinforcement learning, with fewer time and traversal constraints. At its current state, a number of valid papers are missed as well as a number of invalid papers gathered.

There is a great deal that could still be added to the comparison utility. It currently has a very good balance of accuracy and efficiency, although there are still optimizations that could be made to increase the speed. The greatest improvement, though, would be to allow the criteria for comparison to be adjusted, so that the user could make the decision between speed and an even higher level of accuracy. There are an almost limitless number of statistics that could be examined to more accurately–or quickly–spot possible plagiarism. Some interesting criteria could include

- the number of words that occur very infrequently in each document, yet occur in both, and

- the percentage of overlap between the unique word sets for both complete documents.

Another improvement to the comparison utility would be to allow more thorough comparison and scoring across all documents, rather than simple pairwise comparison. It is probably of greater value, for instance, to see the total amount of plagiarism in a document from five others than to see five separate comparisons.

## Conclusions

We have presented a web spider and a text comparison utility designed to detect self-plagiarism among Computer Science academics.

## Acknowledgments

Christian Collberg was partially supported by the NSF under grant CCR-0073483. Stephen Kobourov was partially supported by the NSF under grant ACR-0222920. Joshua Louie and Thomas Slattery are in the undergraduate program in the Department of Computer Science at the University of Arizona.

## References

[1] T. Kistler and H. Marais. WebL – A programming language for the web. In *Proceedings of WWW7*, pages 259–270. Elsevier, 1998.

[2] Jason Rennie and Andrew Kachites McCallum. Using reinforcement learning to spider the Web efficiently. In Ivan Bratko and Saso Dzeroski, editors, *Proceedings of ICML-99, 16th International Conference on Machine Learning*, pages

335–343, Bled, SL, 1999. Morgan Kaufmann Publishers, San Francisco, US.

[3] Narayanan Shivakumar and Héctor García-Molina. SCAM: A copy detection mechanism for digital documents. In *Proceedings of the Second Annual Conference on the Theory and Practice of Digital Libraries*, 1995.