Winnowing: Local Algorithms for Document Fingerprinting

Saul Schleimer
MSCS
University of Illinois, Chicago
saul@math.uic.edu

Daniel S. Wilkerson
Computer Science Division
UC Berkeley
dsw@cs.berkeley.edu

Alex Aiken
Computer Science Division
UC Berkeley
aiken@cs.berkeley.edu

ABSTRACT

Digital content is for copying: quotation, revision, plagiarism, and file sharing all create copies. Document fingerprinting is concerned with accurately identifying copying, including small partial copies, within large sets of documents.

We introduce the class of *local* document fingerprinting algorithms, which seems to capture an essential property of any fingerprinting technique guaranteed to detect copies. We prove a novel lower bound on the performance of any local algorithm. We also develop *winnowing*, an efficient local fingerprinting algorithm, and show that winnowing's performance is within 33% of the lower bound. Finally, we also give experimental results on Web data, and report experience with MOSS, a widely-used plagiarism detection service.

1. INTRODUCTION

Digital documents are easily copied. A bit less obvious, perhaps, is the wide variety of different reasons for which digital documents are either completely or partially duplicated. People quote from each other's email and news postings in their replies. Collaborators create multiple versions of documents, each of which is closely related to its immediate predecessor. Important Web sites are mirrored. More than a few students plagiarize their homework from the Web. Many authors of conference papers engage in a similar but socially more acceptable form of text reuse in preparing journal versions of their work. Many businesses, notably in the software and entertainment industries, are based on charging for each digital copy sold.

Comparing whole document checksums is simple and suffices for reliably detecting exact copies; however, detecting partial copies is subtler. Because of its many potential applications, this second problem has received considerable attention.

Most previous techniques for detecting partial copies, which we discuss in more detail in Section 2, make use of the following idea. A *k-gram* is a contiguous substring of length k. Divide a document into k-grams, where k is a parameter chosen by the user. For example, Figure 1(c) contains all the 5-grams of the string of characters in Figure 1(b). Note that there are almost as many k-grams as there are characters in the document, as every position in the document (except for the last $k-1$ positions) marks the beginning of a k-gram. Now hash each k-gram and select some subset of these hashes to be the document's *fingerprints*. In all practical approaches, the set of fingerprints is a small subset of the set of all k-gram hashes. A fingerprint also contains positional information, which we do not show, describing the document and the location within that document that the fingerprint came from. If the hash function is chosen so that the probability of collisions is very small, then whenever two documents share one or more fingerprints, it is extremely likely that they share a k-gram as well.

For efficiency, only a subset of the hashes should retained as the document's fingerprints. One popular approach is to choose all hashes that are *0 mod p*, for some fixed p. This approach is easy to implement and retains only $1/p$ of all hashes as fingerprints (Section 2). Meaningful measures of document similarity can also be derived from the number of fingerprints shared between documents [5].

A disadvantage of this method is that it gives no guarantee that matches between documents are detected: a k-gram shared between documents is detected only if its hash is *0 mod p*. Consider the sequence of hashes generated by hashing all k-grams of a file in order. Call the distance between consecutive selected fingerprints the *gap* between them. If fingerprints are selected *0 mod p*, the maximum gap between two fingerprints is unbounded and any

```
A do run run run, a do run run
```
(a) Some text from [7].

```
adorunrunrunadorunrun
```
(b) The text with irrelevant features removed.

```
adoru dorun orunr runru unrun nrunr runru
unrun nruna runad unado nador adoru dorun
orunr runru unrun
```
(c) The sequence of 5-grams derived from the text.

```
77 72 42 17 98 50 17 98 8 88 67 39 77 72 42
17 98
```
(d) A hypothetical sequence of hashes of the 5-grams.

```
72 8 88 72
```
(e) The sequence of hashes selected using *0 mod 4*.

Figure 1: Fingerprinting some sample text.

SIGMOD 2003, June 9-12, 2003, San Diego, CA.

matches inside a gap are not detected.[1] In experiments with selecting fingerprints *0 mod p* on HTML data taken from the Web, we found that gaps between fingerprints chosen *0 mod p* can be quite large. In fact, there are passages much longer than the size of the average Web page in which no hashes are *0 mod p* for reasonable values of p (Section 5).

In this paper we give the first efficient algorithm for selecting the fingerprints from a sequence of hashes that guarantees that at least part of any sufficiently long match is detected. Define a *window* of size w to be w consecutive hashes of k-grams in a document (w is a parameter set by the user). By selecting at least one fingerprint from every window our algorithm limits the maximum gap between fingerprints. In fact, our algorithm is guaranteed to detect at least one k-gram in any shared substring of length at least $w + k - 1$.

Central to our construction is the idea of a *local* algorithm (Section 4), which we believe captures the essential properties of any document fingerprinting algorithm which guarantees that matches are detected. An algorithm is local if, for every window of w consecutive hashes $h_i, \ldots, h_{i+w-1}$, the algorithm decides to select one of these hashes as a fingerprint and this choice depends only on the window's contents $h_i, \ldots, h_{i+w-1}$. Note that some other of the hashes in this window may also be selected due the action of the algorithm in one of the (potentially many) overlapping windows.

With respect to a given input distribution, let the *density* of a fingerprinting algorithm be the expected proportion of hashes it selects as fingerprints. In Section 3 we introduce a particular local algorithm, the *winnowing* algorithm. We analyze its performance on random (independent uniformly distributed) data. We show that, with a given window size w, the density is asymptotically $2/(w + 1)$. We also prove a lower bound of $1.5/(w + 1)$ on the density of local algorithms. Thus the winnowing algorithm is within 33% of optimal.

We also report on experience with two implementations of winnowing. The first is an purely experimental framework for comparing actual performance with the theoretical predictions (Section 5). Our analysis of all fingerprinting algorithms, including that of winnowing, is based on the assumption that the sequence of hashes generated by hashing k-grams is independent and uniformly random. On such random data the performance of our system matches theoretical predictions perfectly. We have found, however, that there are situations where real data does not generate sufficiently random sequences of hashes. In particular, there are clusters of low-entropy strings on the Web, such as

$$000000000000000000000000\ldots$$

or more complex patterns such as

$$abbaabbaabbaabbaabbaabba\ldots$$

The important characteristic of such strings is that they have few distinct k-grams, and therefore few distinct hashes of k-grams. In the case of a long string of a single character, there is only a single k-gram (and hash). The straightforward winnowing algorithm selects far more fingerprints than predicted on such strings, but a simple modification of the algorithm reduces the density of fingerprints[2]. Note that other fingerprint-selection approaches are exposed to the same problem. For example, choosing hashes that are *0 mod p* means that on the string "11111111 ..." either all or none of the hashes of the string are selected as fingerprints.

We also report on experience with MOSS, a widely-used service for detecting plagiarism, primarily in programming assignments (Section 5.2).[3] Document fingerprinting has worked extremely well in this setting. Because the basic idea of hashing k-grams makes minimal assumptions about the form of the input, it is easy to incorporate fingerprinting for a new data format without disturbing the underlying hashing engine. False positives appear to be non-existent, and the infrequent reports of false negatives (instances of shared substrings missed by the system) have always been tracked back either to implementation bugs or user error.

[1] See Section 3.1 for a discussion of the expected gap size.

[2] See Definition 3 "Robust Winnowing" in Section 5.1

2. BACKGROUND AND RELATED WORK

Not all copy-detection algorithms are based on selecting fingerprints of k-grams. To give some basis for discussing different techniques, we list several criteria that a copy-detection algorithm should satisfy. These requirements are based on our own experience (mostly in the context of plagiarism detection), but echo many of the goals outlined in papers on related techniques.

2.1 Desirable properties

We believe a copy-detection algorithm should have three properties:

1. *Whitespace insensitivity* In matching text files, matches should be unaffected by such things as extra whitespace, capitalization, punctuation, etc. In other domains the notion of what strings should be equal is different—for example, in matching software text it is desirable to make matching insensitive to variable names.

2. *Noise suppression* Discovering short matches, such as the fact that the word *the* appears in two different documents, is uninteresting. Any match must be large enough to imply that the material has been copied and is not simply a common word or idiom of the language in which documents are written.

3. *Position independence* Coarse-grained permutation of the contents of a document (e.g., scrambling the order of paragraphs) should not affect the set of discovered matches. Adding to a document should not affect the set of matches in the original portion of the new document. Removing part of a document should not affect the set of matches in the portion that remains.

All schemes that we know of handle property (1) in essentially the same way. A first pass over the data transforms it to eliminate undesirable differences between documents. For example, whitespace and punctuation are removed, all letters are converted to lower case, or all variable names are replaced by the identifier "V". The exact details vary from one type of document to the next, but the essential feature is that semantic information about the document type is used to eliminate unimportant differences between documents.

Schemes based on fingerprinting k-grams satisfy requirement (2) by choosing k to be sufficiently long that common idioms of the language have length shorter than k. An important assumption of this class of algorithms is that there is some threshold k such that matches with length shorter than k are almost always uninteresting and matches with length longer than k are almost always interesting. In our experience, there is such a sharp threshold (Section 5).

The most interesting requirement is (3). Before describing methods for addressing requirement (3), we first give a short history of the use of hashes of k-grams in copy detection.

[3] `http://wwww.cs.berkeley.edu/~aiken/moss.html`

2.2 Karp-Rabin String Matching

Karp and Rabin's algorithm for fast substring matching is apparently the earliest version of fingerprinting based on k-grams [10]. Their problem, which was motivated by string matching problems in genetics, is to find occurrences of a particular string s of length k within a much longer string. The idea is to compare hashes of all k-grams in the long string with a hash of s. However, hashing strings of length k is expensive for large k, so Karp and Rabin propose a "rolling" hash function that allows the hash for the $i+1^{\text{st}}$ k-gram to be computed quickly from the hash of the i^{th} k-gram. Treat a k-gram $c_1 \ldots c_k$ as a k-digit number in some base b. The hash $H(c_1 \ldots c_k)$ of $c_1 \ldots c_k$ is this number:

$$c_1 * b^{k-1} + c_2 * b^{k-2} * \ldots + c_{k-1} * b + c_k$$

To compute the hash of the k-gram $c_2 \ldots c_{k+1}$, we need only subtract out the high-order digit, multiply by b, and add in the new low order digit. Thus we have the identity:

$$H(c_2 \ldots c_{k+1}) = (H(c_1 \ldots c_k) - c_1 * b^{k-1}) * b + c_{k+1}$$

Since b^{k-1} is a constant, this allows each subsequent hash to be computed from the previous one with only two additions and two multiplications. Further, this identity holds when addition and multiplication are modulo some value (e.g., the size of the largest representable integer), so this method works well with standard machine arithmetic.

As an aside, this rolling hash function has a weakness. Because the values of the c_i are relatively small integers, doing the addition last means that the last character only affects a few of the low-order bits of the hash. A better hash function would have each character c_i potentially affect all of the hash's bits. As noted in [5], it is easy to fix this by multiplying the entire hash of the first k-gram by an additional b and then switching the order of the multiply and add in the incremental step:

$$H'(c_2 \ldots c_{k+1}) = ((H'(c_1 \ldots c_k) - c_1 * b^k) + c_{k+1}) * b$$

2.3 All-to-all matching

The first scheme to apply fingerprinting to collections of documents was developed by Manber, who apparently independently discovered Karp-Rabin string matching and applied it to detecting similar files in file systems [12]. Rather than having a single candidate string to search for, in this problem we wish to compare all pairs of k-grams in the collection of documents.

The all-to-all nature of this comparison is a key difficulty in document fingerprinting. To illustrate, consider the problem of all-to-all matching on ASCII text. Since there is a k-gram for every byte of an ASCII file, and at least 4-byte hashes are needed for most interesting data sets, a naive scheme that selected all hashed k-grams would create an index much larger than the original documents. This is impractical for large document sets, and the obvious next step is to select some subset of the hashes to represent each document. But which hashes should be selected as fingerprints?

A simple but incorrect strategy is to select every i^{th} hash of a document, but this is not robust against reordering, insertions and deletions (requirement (3) above). In fact, prepending one character to a file shifts the positions of all k-grams by one, which means the modified file shares none of its fingerprints with the original. Thus, any effective algorithm for choosing the fingerprints to represent a document cannot rely on the position of the fingerprints within the document.

The scheme Manber chose is to select all hashes that are *0 mod p*. In this way fingerprints are chosen independent of their position, and if two documents share a hash that is *0 mod p* it is selected in both documents. Manber found this technique worked well.

In [8], Heintze proposed choosing the n smallest hashes of all k-grams of a document as the fingerprints of that document. By fixing the number of hashes per document, the system would be more scalable as large documents have the same number of fingerprints as small documents. This idea was later used to show that it was possible to cluster documents on the Web by similarity [6]. The price for a fixed-size fingerprint set is that only near-copies of entire documents could be detected. Documents of vastly different size could not be meaningfully compared; for example, the fingerprints of a paragraph would probably contain no fingerprints of the book that the paragraph came from. Choosing hashes *0 mod p*, on the other hand, generates variable size sets of fingerprints for documents but guarantees that all representative fingerprints for a paragraph would also be selected for the book. Broder [5] classifies these two different approaches to fingerprinting as being able to detect only *resemblance* between documents or also being able to detect *containment* between documents.

2.4 Other techniques

Instead of using k-grams, the strings to fingerprint can be chosen by looking for sentences or paragraphs, or by choosing fixed-length strings that begin with "anchor" words [4, 12]. Early versions of our system also used structure gleaned from the document to select substrings to fingerprint. The difficulty with such schemes, in our experience, is that the implementation becomes rather specific to a particular type of data. If the focus is on English text, for example, choosing sentences as the unit to hash builds in text semantics that makes it rather more difficult to later use the system to fingerprint, say, C programs, which have nothing resembling English sentences. In addition, even on text data the assumption that one can always find reasonable sentences is questionable: the input may be a document with a large table, a phone book, or Joyce's *Finnegans Wake* [9]. In our experience, using k-grams as the unit of hashing is much more robust than relying on common-case assumptions about the frequency of specific structure in the input.

There are approaches to copy detection not based on fingerprinting. For example, in SCAM, a well-known copy-detection system, one of the ideas that is explored is that two documents are similar if the distance between feature vectors representing the two documents is small. The features are words, and the notion of distance is a variation on standard information-retrieval measures of similarity [14].

Baker considers the problem of finding near-duplication in software and develops the notion of *parameterized matches*, or p-matches. Consider two strings, some letters of which are designated as parameters. The strings match if there is a renaming of parameters that makes the two strings equal. For example, if we take the parameters to be variable names, then two sections of program text could be considered equal if there was a renaming of variables that mapped one program into the other. Baker gives an algorithm for computing p-matches and reports on experience with an implementation in [2] and in a subsequent paper considers how to integrate these ideas with matching on k-grams [3].

There is an important distinction to be made between copy-detection for discrete data and for continuous data. For discrete data, such as text files and program source, after a simple suppression of the uninteresting pieces of documents, exact matching on substrings of the remainder is a useful notion. For continuous data, such as audio, video, and images, there have been a number of commercial copy-detection systems built but relatively little has been published in the open literature (an exception is [13]). The problems here are

more difficult, because very similar copies of images, for example, may have completely different bit representations, requiring a much more sophisticated first step to extract features of interest before the matching can be done.

Further afield from copy-detection, but still related, is Digital Rights Management (DRM). DRM systems seek to solve the problem of the use of intellectual property by preventing or controlling copying of documents. DRM schemes are encryption-based: the valuable content is protected by encrypting it and can only be used by those who have been granted access in the form of the decryption key. However, regardless of the copy-prevention technology chosen, users must ultimately have access to the unencrypted data somehow—otherwise they cannot use it—and as discussed in Section 1, it seems to be nearly a natural law that digital content is copied. We find ourselves in agreement with [4]: for at least some forms of digital media copy-prevention systems will have trouble ultimately succeeding. We suspect that in many environments the best one can hope for is efficient copy detection.

3. WINNOWING

In this section we describe and analyze the winnowing algorithm for selecting fingerprints from hashes of k-grams. We give an upper bound on the performance of winnowing, expressed as a trade-off between the number of fingerprints that must be selected and the shortest match that we are guaranteed to detect.

Given a set of documents, we want the find substring matches between them that satisfy two properties:

1. If there is a substring match at least as long as the *guarantee threshold*, t, then this match is detected, and
2. We do not detect any matches shorter than the *noise threshold*, k.

The constants t and $k \leq t$ are chosen by the user. We avoid matching strings below the noise threshold by considering only hashes of k-grams. The larger k is, the more confident we can be that matches between documents are not coincidental. On the other hand, larger values of k also limit the sensitivity to reordering of document contents, as we cannot detect the relocation of any substring of length less than k. Thus, it is important to choose k to be the minimum value that eliminates coincidental matches (see Section 5).

Figures 2(a)-(d) are reproduced from Figure 1 for convenience and show a sequence of hashes of 5-grams derived from some sample text.

Given a sequence of hashes $h_1 \ldots h_n$, if $n \geq t - k$, then at least one of the h_i must be chosen to guarantee detection of all matches of length at least t. This suggests the following simple approach. Let the *window size* be $w = t - k + 1$. Consider the sequence of hashes $h_1 h_2 \ldots h_n$ that represents a document. Each position $1 \leq i \leq n - w + 1$ in this sequence defines a *window* of hashes $h_i \ldots h_{i+w-1}$. To maintain the guarantee it is necessary to select one hash value from every window to be a fingerprint of the document. (This is also sufficient, see Lemma 1.) We have found the following strategy works well in practice.

DEFINITION 1 (WINNOWING). *In each window select the minimum hash value. If there is more than one hash with the minimum value, select the rightmost occurrence. Now save all selected hashes as the fingerprints of the document.*

Figure 2(e) gives the windows of length four for the sequence of hashes in Figure 2(d). Each hash that is selected is shown in boldface (but only once, in the window that first selects that hash). The

```
A do run run run, a do run run
```
(a) Some text.

```
adorunrunrunadorunrun
```
(b) The text with irrelevant features removed.

```
adoru dorun orunr runru unrun nrunr runru
unrun nruna runad unado nador adoru dorun
orunr runru unrun
```
(c) The sequence of 5-grams derived from the text.

```
77 74 42 17 98 50 17 98 8 88 67 39 77 74 42
17 98
```
(d) A hypothetical sequence of hashes of the 5-grams.

```
(77, 74, 42, 17)  (74, 42, 17, 98)
(42, 17, 98, 50)  (17, 98, 50, 17)
(98, 50, 17, 98)  (50, 17, 98,  8)
(17, 98,  8, 88)  (98,  8, 88, 67)
( 8, 88, 67, 39)  (88, 67, 39, 77)
(67, 39, 77, 74)  (39, 77, 74, 42)
(77, 74, 42, 17)  (74, 42, 17, 98)
```
(e) Windows of hashes of length 4.

```
17 17 8 39 17
```
(f) Fingerprints selected by winnowing.

```
[17,3] [17,6] [8,8] [39,11] [17,15]
```
(g) Fingerprints paired with 0-base positional information.

Figure 2: Winnowing sample text.

intuition behind choosing the minimum hash is that the minimum hash in one window is very likely to remain the minimum hash in adjacent windows, since the odds are that the minimum of w random numbers is smaller than one additional random number. Thus, many overlapping windows select the same hash, and the number of fingerprints selected is far smaller than the number of windows while still maintaining the guarantee. Figure 2(f) shows the set of fingerprints selected by winnowing in the example.

In many applications it is useful to record not only the fingerprints of a document, but also the position of the fingerprints in the document. For example, we need positional information to show the matching substrings in a user interface. An efficient implementation of winnowing also needs to retain the position of the most recently selected fingerprint. Figure 2(f) shows the set of [*fingerprint, position*] pairs for this example (the first position is numbered 0). To avoid the notational complexity of indexing all hashes with their position in the global sequence of hashes of k-grams of a document, we suppress most explicit references to the position of k-grams in documents in our presentation.

3.1 Expected Density

Recall that the *density* of a fingerprinting algorithm is the expected fraction of fingerprints selected from among all the hash values computed, given random input (Section 1). We now analyze the density of winnowing, which gives the trade-off between the guarantee threshold and the number of fingerprints required.

Consider the function C that maps the position of each selected fingerprint to the position of the first (leftmost) window that se-

lected it in the sequence of all windows for a document. We say we are *charging* the cost of saving the fingerprint to the indicated window. The charge function is monotonic increasing — that is, if p and q are the positions of two selected fingerprints and $p < q$, then $C(p) < C(q)$.

To prove this, assume fingerprints are selected at distinct positions p and q where $p < q$ but $C(p) > C(q)$. Then both positions p and q are in both windows. Let h_p be the hash at position p and let h_q be the hash at position q. There are two possibilities: If $h_p = h_q$ then, as $p < q$, the window $C(p)$ was not charged for p nor for q, as $C(q) < C(p)$. If $h_p \neq h_q$ then one of $C(p)$ or $C(q)$ was not charged. These both contradict the hypothesis. We conclude that the charge function is monotonic increasing.

To proceed further recall that the sequence of hashes we are winnowing is random. We assume that the space of hash values is very large so that we can safely ignore the possibility that there is a tie for the minimum value for any small window size. We examine the soundness of this assumption in Section 5.

Consider an indicator random variable X_i that is one iff the i^{th} window W_i is charged. Consider the adjacent window to the left W_{i-1}. The two intervals overlap except at the leftmost and rightmost positions. Their union is an interval of length $w+1$. Consider the position p containing the smallest hash in that union interval. Any window that includes p selects h_p as a fingerprint. There are three cases:

1. If $p = i - 1$, the leftmost position in the union, then W_{i-1} selects it. Since $p \notin W_i$, we know W_i must select a hash in another position, q. This hash is charged to W_i since W_i selected it, W_{i-1} did not select it, and the charge function is monotonic increasing. Thus in this case, $X_i = 1$.

2. If $p = i+w-1$, the rightmost position in the union interval, then W_i selects it. W_i must be charged for it, as W_i is also the very leftmost interval to contain p. Again, $X_i = 1$.

3. If p is in any other position in the union interval, both W_{i-1} and W_i select it. No matter who is charged for it, it won't be W_i, since W_{i-1} is further left and also selected it. Thus in this case, $X_i = 0$.

The first two cases happen with probability $1/(w + 1)$, and so the expected value of X_i is $2/(w + 1)$. Recall that the sum of the expected values is the expected value of the sum, even if the random variables are not independent. The total expected number of intervals charged, and therefore the total number of fingerprints selected, is just this value times the document length. Thus the density is

$$d = \frac{2}{w+1}.$$

3.1.1 *Comparison to* 0 mod p *at same density*

Here we compare the *0 mod p* algorithm and winnowing at the same density. That is, we take $p = 1/d = (w + 1)/2$. For a string of length $t = w + k - 1$ consider the event that the *0 mod p* algorithm fails to select any fingerprint at all within it. (Recall that winnowing would never fail to do so.) We now compute the probability of this event for one given string. Please note that for two overlapping such strings these events are not independent. Thus the probability we compute is not a good estimate for the fraction of all such substrings of a text that do not have a fingerprint selected using the *0 mod p* algorithm.

Again we assume independent uniformly distributed hash values. Also we assume large w; in our experiments $w = 100$ (see Section 5.1). Thus, the probability that the guarantee fails in a given sequence of text of length t, i.e. that no hash in a given sequence of w hashes is *0 mod p*, is

$$(1-d)^w = \left(1 - \frac{2}{w+1}\right)^w \approx e^{\frac{-2w}{w+1}} = e^{-2+\frac{2}{w+1}} \sim 13.5\%.$$

3.1.2 *Comparison to* 0 mod p *with guarantee*

One may be tempted to try modifying the *0 mod p* algorithm to give a guarantee. There is one straightforward solution that we know of: In the event that a gap longer than the guarantee threshold threatens to open up, select *all* hashes as fingerprints until the next hash that is *0 mod p*.

Let the *Safe* 0 mod p algorithm be as follows. Partition hashes into:

- *Good* if a hash is *0 mod p*.
- *Bad* if it and the $w - 1$ hashes to its left are not Good, and
- *Ugly* otherwise [11].

Select all non-Ugly hashes as fingerprints. As we will see in Section 4 this algorithm is *local* and is therefore correct. Note that we have chosen the parameters so that the guarantee t is the same as that of winnowing. All that remains is to compute the optimal expected density.

Fix a document and consider a position i. Let G_i and B_i denote the events that the hash at i is good or bad respectively. (Our notation for an event also denotes the appropriate indicator random variable (1 = true and 0 = false) depending on context.) Let $P = 1/p$. (Note that to compete with winnowing we would need P to be rather small: $P \leq 2/(w + 1)$; however even a slightly larger P will allow for the $1 + x \approx e^x$ approximation we use below.) We have

$$\Pr[G_i] = P$$

and (except for the very first $w - 1$ hashes) for small P

$$\Pr[B_i] = (1 - P)^w \approx e^{-wP}.$$

Again, the expected value of a sum is the sum of the expected values. Ignoring the error introduced by the first $w - 1$ hashes, we have that the expected value of the non-ugliness of a position is

$$\begin{aligned} \text{Ex}\left[\sum G_i + B_i\right] &= \sum \text{Ex}\left[G_i\right] + \sum \text{Ex}\left[B_i\right] \\ &= NP + N(1-P)^w \\ &\approx N(P + e^{-wP}). \end{aligned}$$

The next step is to minimize the density. Let $f(P) = P+e^{-wP}$. Setting $f'(P_0) = 0$ and solving we have $e^{wP_0} = w$, or

$$P_0 = \frac{\ln w}{w}.$$

We check that $f''(P) = w^2 e^{-wP} > 0$ so we have found the global minimum. If we use this optimal value, P_0, the Safe *0 mod p* algorithm has density at least

$$f\left(P_0\right) = \frac{\ln w}{w} + e^{-\frac{w \ln w}{w}} = \frac{1 + \ln w}{w},$$

which is considerably more than that of winnowing: $2/(w + 1)$.

3.2 Queries

This section is primarily about how to choose hashes well, but we digress a bit here to discuss how hashes can be used once selected. In a typical application, one first builds a database of fingerprints and then queries the fingerprints of individual documents against this database (see Section 5). Winnowing gives us some flexibility

to treat the two fingerprinting times (database-build time and query time) differently.

Consider a database of fingerprints (obtained from k-grams) generated by winnowing documents with window size w. Now, query documents can be fingerprinted using a different window size. Let F_w be the set of fingerprints chosen for a document by winnowing with window size w. The advantage of winnowing query documents with a window size $w' \geq w$ is that $F_{w'} \subseteq F_w$, which means fewer memory or disk accesses to look up fingerprints. This may be useful if, for example, the system is heavily loaded and we wish to reduce the work per query, or if we are just interested in obtaining a faster but coarser estimate of the matching in a document.

We can extend this idea one step further. Fingerprint a query document with the same window w used to generate the database, and then sort all of the selected fingerprints in ascending order. Next look up some number of the fingerprints in the database, starting with the smallest. If we stop after a few, fixed number of hashes, we have realized Broder's and Heintze's approach for testing document resemblance [8, 5]. If we use all of the hashes as fingerprints, we realize the standard notion of testing for document containment. There is also a spectrum where we stop anywhere in between these two extremes. Broder's paper on resemblance and containment gives distinct algorithms to compute these two properties [5]; winnowing naturally realizes both.

4. LOCAL ALGORITHMS

In this section we consider whether there are fingerprinting algorithms that perform better than winnowing. We introduce the notion of *local* fingerprinting algorithms. We prove a lower bound for the density of a local algorithm given uniform identically distributed random input. This lower bound does not meet the upper bound for winnowing. We suspect the lower bound can be improved.

Winnowing selects the minimum value in a window of hashes, but it is clearly just one of a family of algorithms that choose elements from a local window. Not every method for selecting hashes from a local window maintains the guarantee, however. Assume, for example, that the window size is 50 and our approach is to select every 50^{th} hash as the set of fingerprints. While this method does select a hash from every window, it depends on the global position of the hash in a document, and, as discussed in Section 2, any such approach fails in the presence of insertions or deletions. The key property of winnowing is that the choice of hash depends only on the contents of the window—it does not depend on any external information about the position of the window in the file or its relationship to other windows. This motivates the following definition.

DEFINITION 2 (LOCAL ALGORITHMS). *Let S be a* selection *function taking a w-tuple of hashes and returning an integer between zero and $w-1$, inclusive. A fingerprinting algorithm is* local *with selection function S, if, for every window $h_i, \ldots, h_{i+w-1}$, the hash at position $i+S(h_i, \ldots, h_{i+w-1})$ is selected as a fingerprint.*

It can be beneficial to weaken locality slightly to provide flexibility in choosing among equal fingerprints—see Section 5.1. We now show that any local algorithm is correct, in the sense that it meets the guarantee threshold t.

LEMMA 1 (CORRECTNESS OF LOCAL ALGORITHMS). *Any matching pair of substrings of length at least t is found by any local algorithm.*

PROOF. The sequence of hashes of k-grams representing the each substring spans at least one window, W, of length w. Because the selection function is only a function of the contents of W, the same fingerprint is selected from W in both copies. □

We now consider whether there is any local algorithm that is better than winnowing. We do not have a matching lower bound for winnowing, but we can show the following:

THEOREM 1 (LOWER BOUND). *Any local algorithm with noise threshold k and guarantee $t = w + k - 1$ has density*

$$d \geq \frac{1.5}{w+1}.$$

Note that winnowing algorithm, with a density of $2/(w+1)$, is within 33% of this lower bound.

PROOF. Assume the hashes are independent and uniformly distributed. Consider the behavior of the algorithm on every $w+1^{\text{st}}$ window. Such windows are separated by a single position that is not part of either window. Because the windows are disjoint, their hashes and selected fingerprints are independent of each other, and each window selects a separate fingerprint.

Now consider all of the windows between the i^{th} and $(i+w+1)^{\text{st}}$ windows W_i and W_{i+w+1}; these are the w windows that overlap the disjoint windows at either end. Let Z be the random variable such that $Z = 0$ iff among these windows, no additional fingerprint is selected, and $Z = 1$ otherwise. We compute a lower-bound on the expected value of Z.

Let X and Y denote the random variables $S(W_i)$ and $S(W_{i+w+1})$ respectively. Again, because the windows do not overlap, X and Y are independent.

Now, if $Y \geq X$ then $Z = 1$, because the algorithm is required to select at least one additional fingerprint from a window in between W_i and W_{i+w+1}. Otherwise $Z \geq 0$. Since X and Y are identically distributed we have $\Pr[Y > X] = \Pr[X > Y]$. Let Θ denote this quantity. Let $\Delta = \Pr[Y = X]$. We have $1 = 2\Theta + \Delta$. Thus $\Theta + \Delta = (1+\Delta)/2 > 1/2$ and

$$\text{Ex}[Z] \geq \Pr[Y \geq X] = \Theta + \Delta > 1/2.$$

We thus see that in every sequence of $w+1$ windows, in addition to the fingerprint selected in the first window we expect to select an additional distinct fingerprint at least half the time for one of the subsequent windows. The density of selected points is therefore

$$d \geq \frac{1.5}{w+1}.$$

□

OBSERVATION 1. *This result can be improved slightly: As a bit of notation let $x_i = Pr[X = i]$ for $i = 0, 1, \ldots, w-1$. Of course $\sum x_i = 1$. Then $\Delta = Pr[Y = X] = \sum x_i^2$. Apply the Cauchy-Schwartz inequality to show $\Delta \geq 1/w$. The proof above then gives density*

$$d \geq \frac{1.5 + \frac{1}{2w}}{w+1}.$$

Our lower bound proof relies only on information derived from two windows that are separated sufficiently to be disjoint. We conjecture therefore that $2/(w+1)$ is a lower bound on the density of any local fingerprinting algorithm.

total bytes	$7,182,692,852$
text bytes	$1,940,576,448$
hashes computed	$1,940,576,399$
winnowing fingerprints	$38,530,846$
measured density	0.019855
expected density	0.019802
fingerprints for $0 \bmod 50$	$38,761,128$
measured density	0.019974
expected density	0.020000
longest run with no fingerprint	29983

Figure 3: Results on 500,000 HTML pages

5. EXPERIMENTS

In this section we report on our experience with two different implementations of winnowing. In Section 5.1 we report on a series of experiments on text data taken from the World Wide Web, and in Section 5.2 we give a more qualitative report on experience over several years with a widely-used plagiarism detection service, MOSS.

5.1 Experiments with Web Data

Because of its size and the degree of copying, the World-Wide Web provides a readily accessible and interesting data set for document fingerprinting algorithms. For these experiments, we used 500,000 pages downloaded from the Stanford WebBase [1]. We use the rolling hash function described in Section 2. Because fingerprinting a half-million Web pages generates nearly two billion hashes and 32-bits can represent only about four billion distinct hash values, we use 64-bit hashes to avoid accidental collisions. As an aside, we have found using a rolling (or incremental) hash function is important for performance with realistic k-gram sizes (say $k = 50$) when using 64-bit arithmetic. Recomputing a 64-bit hash from scratch for each k-gram reduces the throughput of the fingerprinting algorithm by more than a factor of four.

In our first experiment we simply fingerprinted 8MB of randomly generated text. This experiment serves solely to check that our hash function is reasonably good, so that we can trust the number of matches found in experiments on real data. Strings of 50 characters were hashed and the winnowing window was set at 100. Winnowing selected 0.019902 of the hashes computed, which very closely matches the expected density of $2/(100 + 1) = 0.019802$. Selecting hashes equal to *0 mod 50* results in a measured density of 0.020005, which is also very close to the predicted value of 0.02. We also observed a uniform distribution of hash values; taken all together, the hash function implementation appears to be sufficient for our fingerprinting algorithm.

Our second experiment calculated the hashes for 500,000 HTML documents and measured various statistics. We again measured the density and compared it with the expected density for both winnowing and selecting fingerprints equal to *0 mod p*. Again the winnowing window size is 100 and the noise threshold is 50. The results are shown in Figure 3.

There were interesting things to note in the data. Both algorithms come close to the expected density in each case. However, the gross averages cover up some local aberrations. For example, there is a run of over 29,900 non-whitespace, non-tag characters that has no hash that is *0 mod 50*. It is easily checked that the odds of this happening on uniformly random inputs are extremely small. (The chances that a string of 29,900 characters has no hash of a substring that is *0 mod 50* is $(1 - 1/50)^{29,851}$, which is less than 10^{-260}. Even in a terabyte, or 2^{40} bytes, of data, the chances that every substring of length 29,900 has no k-gram hash that is *0 mod 50*, is less than 10^{-220}.) Clearly the data on the Web is not uniformly random.

As discussed briefly in Section 1, there are long passages on the Web of repetitive, low-entropy strings. For example, in one experiment we did (not reported here) we stumbled across a collection of pages that appear to be raw data taken from sensors in a research experiment. This data consists mostly of strings of 0's with the occasional odd character thrown in. Both winnowing as defined so far and selecting hashes equal to *0 mod p* perform poorly on such data. For the latter, if a long string has few k-grams, then it is very likely that none of them is *0 mod p*, and no fingerprints are selected for the region at all. This is what leads to the large gaps in fingerprints for this strategy on real data.

Winnowing, however, has a different problem. In low-entropy strings there are many equal hash values, and thus many ties for the minimum hash in a given window. To be truly local and independent of global position, it is necessary to take, say, the rightmost such hash in the winnowing window. But in the extreme case, say a long string of 0's with only one k-gram, nearly every single hash is selected, because there is only a single k-gram filling the entire winnowing window and at each step of the algorithm we must choose the rightmost copy—which is a new copy in every window.

There is, however, an easy fix for this problem. We refine winnowing as follows:

DEFINITION 3 (ROBUST WINNOWING). *In each window select the minimum hash value. If possible break ties by selecting the same hash as the window one position to the left. If not, select the rightmost minimal hash. Save all selected hashes as the fingerprints of the document.*

Robust winnowing attempts to break ties by preferring a hash that has already been chosen by a previous window. This is no longer a local algorithm, but one easily observes that for any two matching substrings of length $t = w + k - 1$ we guarantee to select the same hash value and so the match is still found; we simply no longer guarantee that these fingerprints are in the same relative position in the substrings. However, the two fingerprints are close, within distance $w - 1$. This technique reduces the density on a string such as "0000 ..." from asymptotically 1 to just $1/w$, one fingerprint selected per window-length. We reran the experiment in Figure 3 and found that the density of winnowing dropped from 0.019855 to 0.019829. One can imagine non-text document sets where the difference could be greater.

One may wonder why we bother worrying about low-entropy strings at all, as they are in a technical sense inherently uninteresting. But just because data is low-entropy does not mean that people are not interested in it—take the example of the sensor data given above. Such strings do exist and people may want to fingerprint a large corpus of low entropy data so that copies of it can be tracked just as they may want to fingerprint any other sort of document.

Our final experiment examines in more detail the structure of copying in 20,000 Web pages taken from our corpus of 500,000 pages. Interestingly, even though the theoretical predictions based upon an assumption that the input is uniformly random work very well, the distribution of real data is hardly uniform. We need two definitions:

- Let the *frequency* of a k-gram (or its hash) be the number of times it occurs.
- Sort the frequencies in monotonically decreasing order. The *rank* of a k-gram (or its hash) is the position of its frequency

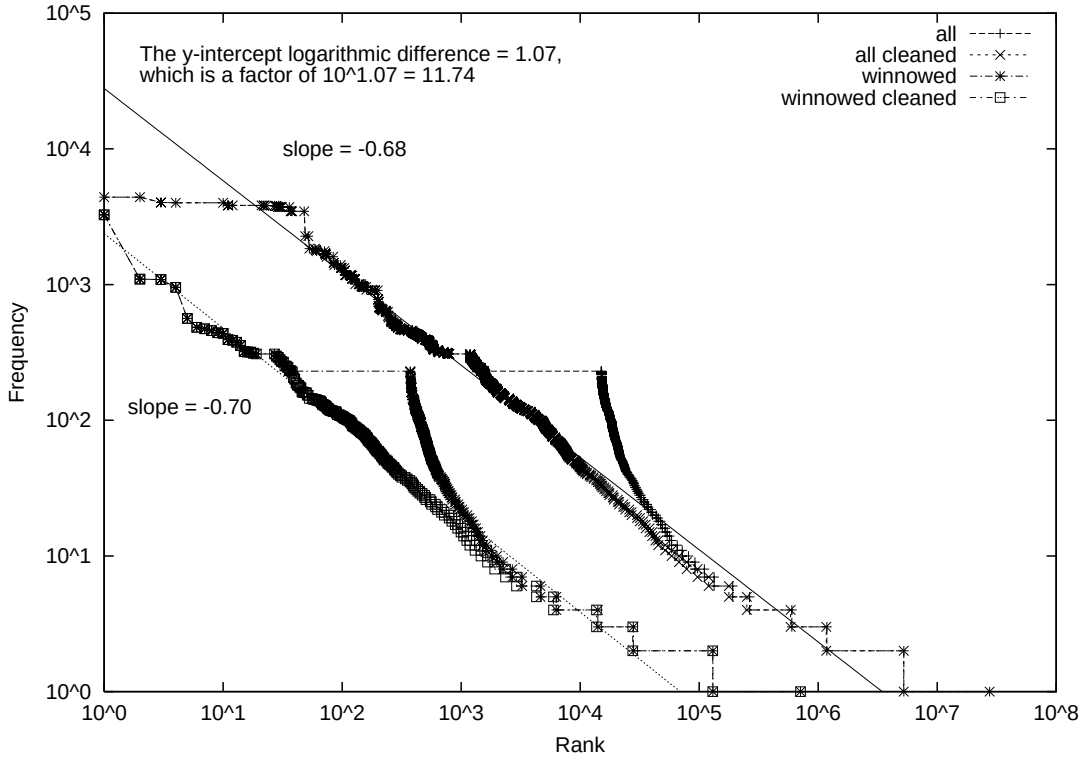

Figure 4: Log-log plot of Frequency by Rank for all hashes (upper line) and for fingerprints (lower line) on 20,000 Web pages.

on this list, starting with 1 for the most frequently occurring k-gram.

Plotting the resulting (*rank*, *frequency*) pairs on a log-log scale one obtains a line of slope about -0.7, demonstrating a power law relationship between frequency and rank

$$f \propto r^{-0.7}.$$

Two such plots of (*rank*, *frequency*) pairs are shown in Figure 4. The upper curve is all k-gram hashes computed from the entire set of 20,000 Web pages, while the lower curve is only those hashes selected as fingerprints by winnowing. (This is the reason we have limited the data set in this experiment to 20,000 pages. Even on 20,000 Web pages saving the hash of every k-gram requires quite a bit of storage.) Our data contains an aberrant "plateau", perhaps because of one document with a long repeating pattern of text, or one file that occurs many times in our sample. The peaked-looking curves labeled "all" show all of the data while the curves labeled "cleaned" have the aberrant plateau removed.

Since frequencies are integers, they form plateaus where a set of hashes all have the same frequency. For example, for the upper curve there are over 20 million fingerprints with one occurrence, which ties them all for the last rank. While more obvious in the lower right, these plateaus occur throughout the data. Additionally, due to the logarithmic scale, almost all points are in the lower right. Thus, if one were to actually plot all points, the line fit would be quite poor, as it would just go through the center of the last two plateaus (we tried this). It would also overwhelm the plotting program with points that do nothing other than thicken the horizontal lines drawn between points.

We plot only the left and right endpoints of each plateau (a plateau of length one is plotted twice so the weights are not biased during line fit). The lines are fitted to the cleaned plateau endpoints. The lines fit quite well, giving a slope of about -0.7, which corresponds to the exponent of the power law. Zipf seems to have first noticed the power law phenomenon, stating what has become known as "Zipf's Law" [16]: the frequencies of English words are proportional to the inverse of their rank when listed in decreasing order. That is, frequency and rank of English words exhibit a power law relationship with exponent -1.

In this experiment, 82% of the fingerprints selected by winnowing were chosen only once; 14% were selected twice; and only 2% occurred three times. At the other extreme, one k-gram appears 3,270 times across all the documents. The distribution of frequencies for the set of all hashes is nearly identical to the distribution for winnowed fingerprints; again the number of k-grams that occur once is 82%, while 14% occur twice and 2% occur three times.

We have looked at some of the most common strings and found that they are what one might expect: strings taken from menus (e.g., "English Spanish German French ..."), common legal boilerplate (e.g., disclaimers), and finally repetitive strings (for some reason the string "documentwritedocumentwritedocumentwrite" was very common in our sample). We suspect that the repetitive strings, in particular, are responsible for the most common k-grams. Because such strings have relatively few k-grams, they dramatically increase the frequency of a few k-grams in the overall statistics.

5.2 Plagiarism Detection

One of the authors has run MOSS, a widely-used plagiarism detection service, over the Internet since 1997. MOSS, which stands for Measure Of Software Similarity, accepts batches of documents and returns a set of HTML pages showing where significant sections of a pair of documents are very similar. MOSS is primarily used for detecting plagiarism in programming assignments in computer science and other engineering courses, though several text

formats are supported as well. The service currently uses robust winnowing, which is more efficient and scalable (in the sense that it selects fewer fingerprints for the same quality of results) than previous algorithms we have tried. There are a few issues involved in making such a system work well in practice.

For this application, positional information (document and line number) is stored with each selected fingerprint. The first step builds an index mapping fingerprints to locations for all documents, much like the inverted index built by search engines mapping words to positions in documents. In the second step, each document is fingerprinted a second time and the selected fingerprints are looked up in the index; this gives the list of all matching fingerprints for each document.

Now the list of matching fingerprints for a document d may contain fingerprints from many different documents $d_1, d_2, \ldots$. In the next step, the list of matching fingerprints for each document d is sorted by document and the matches for each pair of documents $(d, d_1), (d, d_2), \ldots$ is formed. Matches between documents are rank-ordered by size (number of fingerprints) and the largest matches are reported to the user. Note that up until this last step, no explicit consideration of pairs of documents is required. This is very important, as we could not hope to carry out copy detection by comparing each pair of documents in a large corpus. By postponing the quadratic computation to the last step, we can optimize it by never materializing the matching for a pair of documents if it falls below some user-specified threshold.

There are a number of issues in a full copy-detection system beyond how fingerprints are selected. To give the reader some sense of how winnowing fits into a complete system, we briefly discuss the most important problems.

MOSS has several thousand users who wish to do copy detection for many different kinds of data. As mentioned in Section 1, we use the following architecture. For each document format, a front-end specific to that format eliminates features that should not distinguish documents (e.g., we eliminate white space in text). As output each front-end produces a string of a standard form, which is the input to the fingerprinting engine. The fingerprinting code itself knows nothing about the different kinds of documents. This architecture has proven essential to maintaining support for a wide variety of document formats. While this benefit may seem obvious, we report it because it is very tempting to put some document semantics in the fingerprinting routines, but we have always found it to be better to keep the document-specific processing separate.

Efficiency is an important consideration for fingerprinting. In Figure 5 we give code for an efficient implementation of the main winnowing loop. This implementation takes advantage of the fact that by far the most common case is that the minimum value from the preceding window is still within the current window; in this case checking to see if there is a new minimum requires only a single comparison. The only instance in which it is necessary to recompute the minimum by traversing the entire window is the case where the minimum hash of the preceding window is just outside of the current window; note that the loop that does the scan of the array works from right-to-left to ensure that the rightmost minimal hash is selected. Thus, the choice of which of several equal hashes to select is not completely arbitrary. Note the `record` function must compute the global position using the relative position, `min`. Saving this position, together with the selected hash, creates a fingerprint. This loop implements winnowing—it always selects the rightmost minimal hash in a window. To implement robust winnowing the $\leq$ comparison on line marked $(*)$ should be replaced by $<$.

As a minor aside, because winnowing selects the minimum hash in each window, the distribution of hashes selected is skewed. If a uniform distribution is desired, the selected hashes can be hashed a second time (not shown in Figure 5).

```
void winnow(int w /*window size*/) {
  // circular buffer implementing window of size w
  hash_t h[w];
  for (int i=0; i<w; ++i) h[i] = INT_MAX;
  int r = 0;              // window right end
  int min = 0;            // index of minimum hash
  // At the end of each iteration, min holds the
  // position of the rightmost minimal hash in the
  // current window.  record(x) is called only the
  // first time an instance of x is selected as the
  // rightmost minimal hash of a window.
  while (true) {
    r = (r + 1) % w;      // shift the window by one
    h[r] = next_hash();   // and add one new hash
    if (min == r) {
      // The previous minimum is no longer in this
      // window.  Scan h leftward starting from r
      // for the rightmost minimal hash.  Note min
      // starts with the index of the rightmost
      // hash.
      for(int i=(r-1)%w; i!=r; i=(i-1+w)%w)
        if (h[i] < h[min]) min = i;
      record(h[min], global_pos(min, r, w));
    } else {
      // Otherwise, the previous minimum is still in
      // this window. Compare against the new value
      // and update min if necessary.
      if (h[r] <= h[min]) {  // (*)
        min = r;
        record(h[min], global_pos(min, r, w));
      }
    }
  }
}
```

Figure 5: Code for winnowing.

A very significant issue in a practical copy-detection system is the ability to ignore boiler-plate. For example, standard copyright notices, disclaimers, and other legalese would all come under the heading of material that we would not be interested in for many applications. In the case of plagiarism detection, boilerplate is usually material supplied by a course instructor that is expected to be part of the final solution—i.e., it is sanctioned copying. Excluding boilerplate is easily done by fingerprinting the boilerplate with a special document ID that indicates any match with that fingerprint should be discarded.

Presentation of the results is another important issue for users. Statistics such as reporting the percentage of overlap between two documents are useful, but not nearly as useful as actually showing the matches marked-up in the original text. MOSS uses the fingerprints to determine where the longest matching sequences are; in particular, if a_1 in document 1 matches a_2 in document 2, and b_1 in document 1 matches b_2 in document 2, and furthermore a_1 and b_1 are consecutive in document 1 and a_2 and b_2 are consecutive in document 2, then we have discovered a longer match across documents consisting of a followed by b. While this merging of matches is easy to implement, k-grams are naturally coarse and some of the match is usually lost at the beginning and the end of the match. It is possible that once a pair of similar documents are detected using fingerprinting that it would be better to use a suffix-tree algorithm [15] to find maximal matches in just that pair of documents.

In Section 2 we mentioned that there appears to be a sharp thresh-

old between what people consider coincidental similarity (meaning reuse of idioms, common words, etc.) and copying. We have no formal experiments on this topic, but we have informally experimented with MOSS by simply examining the results of tests on sample data. Regardless of input data type, the result is always the same: There is some value of k (dependent on the document type) for which the reported matches are likely to be the result of copying, and for a slightly smaller value of k significant numbers of obvious false positives appear in the results. Along the same lines, early versions of MOSS incorporated a technique similar to Baker's parameterized matches (Section 2). However, we found that replacing all of the parameters with a single constant and increasing k by 1 worked just as well. This appears to be a general trick: sophisticated efforts to exploit document semantics can often be closely approximated by very simple exploits of document semantics together with a small increase in k.

We can report that after years of service, MOSS performs its function very well. False positives (hash collisions) have never been reported, and all the false negatives we have seen were quickly traced back to the source, which was either an implementation bug or a user misunderstanding. Furthermore, users report that copy detection does dramatically reduce the instances of plagiarism in their classes.

6. CONCLUSIONS

We have presented winnowing, a local document fingerprinting algorithm that is both efficient and guarantees that matches of a certain length are detected. We have also presented a non-trivial lower bound on the complexity of any local document fingerprinting algorithm. Finally, we have discussed a series of experiments that show the effectiveness of winnowing on real data, and we have reported on our experience with the use of winnowing in practice.

7. ACKNOWLEDGMENTS

The authors wish to thank Joel Auslander, Steve Fink, and Paul Tucker for many useful discussions and for helping make this work possible.

8. REFERENCES

[1] Arvind Arasu, Junghoo Cho, Hector Garcia-Molina, Andreas Paepcke, and Sriram Raghavan. Searching the web. *ACM Transactions on Internet Technology (TOIT)*, 1(1):2–43, 2001.

[2] Brenda S. Baker. On finding duplication and near-duplication in large software systems. In L. Wills, P. Newcomb, and E. Chikofsky, editors, *Second Working Conference on Reverse Engineering*, pages 86–95, Los Alamitos, California, 1995. IEEE Computer Society Press.

[3] Brenda S. Baker and Udi Manber. Deducing similarities in java sources from bytecodes. In *Proc. of Usenix Annual Technical Conf.*, pages 179–190, 1998.

[4] Sergey Brin, James Davis, and Héctor García-Molina. Copy detection mechanisms for digital documents. In *Proceedings of the ACM SIGMOD Conference*, pages 398–409, 1995.

[5] Andrei Broder. On the resemblance and containment of documents. In *SEQS: Sequences '91*, 1998.

[6] Andrei Broder, Steve Glassman, Mark Manasse, and Geoffrey Zweig. Syntactic clustering of the web. In *Proceedings of the Sixth International World Wide Web Conference*, pages 391–404, April 1997.

[7] The Crystals. Da do run run, 1963.

[8] Nevin Heintze. Scalable document fingerprinting. In *1996 USENIX Workshop on Electronic Commerce*, November 1996.

[9] James Joyce. *Finnegans wake [1st trade ed.]*. Faber and Faber (London), 1939.

[10] Richard M. Karp and Michael O. Rabin. Pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[11] Sergio Leone, Clint Eastwood, Eli Wallach, and Lee Van Cleef. *The Good, the Bad and the Ugly / Il Buono, Il Brutto, Il Cattivo (The Man with No Name)*. Produzioni Europee Associate (Italy) Production, Distributed by United Artists (USA), 1966.

[12] Udi Manber. Finding similar files in a large file system. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 1–10, San Fransisco, CA, USA, 17–21 1994.

[13] Peter Mork, Beitao Li, Edward Chang, Junghoo Cho, Chen Li, and James Wang. Indexing tamper resistant features for image copy detection, 1999. URL: citeseer.nj.nec.com/mork99indexing.html.

[14] 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.

[15] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995.

[16] George K. Zipf. *The Psychobiology of Language*. Houghton Mifltm Co., 1935.