Finding Similar Files in a Large File System

Udi Manber

TR 93-33

October 1993

DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF
ARIZONA
TUCSON ARIZONA

To appear in the
1994 Winter USENIX Technical Conference

FINDING SIMILAR FILES IN A LARGE FILE SYSTEM

Udi Manber[1]

Department of Computer Science
University of Arizona
Tucson, AZ 85721
udi@cs.arizona.edu

ABSTRACT

We present a tool, called *sif*, for finding all similar files in a large file system. Files are considered similar if they have significant number of common pieces, even if they are very different otherwise. For example, one file may be contained, possibly with some changes, in another file, or a file may be a reorganization of another file. The running time for finding all groups of similar files, even for as little as 25% similarity, is on the order of 500MB to 1GB an hour. The amount of similarity and several other customized parameters can be determined by the user at a post-processing stage, which is very fast. *Sif* can also be used to very quickly identify all similar files to a query file using a preprocessed index. Application of *sif* can be found in file management, information collecting (to remove duplicates), program reuse, file synchronization, data compression, and maybe even plagiarism detection.

1. Introduction

Our goal is to identify files that came from the same source or contain parts that came from the same source. We say that two files are similar if they contain a significant number of common substrings that are not too small. We would like to find enough common substrings to rule out chance, without requiring too many so that we can detect similarity even if significant parts of the files are different. The two files need not even be similar in size; one file may be contained, possibly with some changes, in the other. The user should be able to indicate the amount of similarity that is sought and also the type of similarity (e.g., files of very different sizes may be ruled out). Similar files may be different versions of the same program, different programs containing a similar procedure, different drafts of an article, etc.

[1] Supported in part by an NSF Presidential Young Investigator Award (grant DCR-8451397), with matching funds from AT&T, by NSF grants CCR-9002351 and CCR-9301129, and by the Advanced Research Projects Agency under contract number DABT63-93-C-0052. Part of this work was done while the author was visiting the University of Washington.

The information contained in this paper does not necessarily reflect the position or the policy of the U.S. Government or other sponsors of this research. No official endorsement should be inferred.

The problem of computing the similarity between two files has been studied extensively and many programs, such as UNIX *diff*, have been developed to solve it. But using *diff* for all pairwise comparisons among, say, 5000 files would require more than 12 million comparisons taking about 5 *months* of CPU time, assuming 1 second per comparison. Even an order of magnitude improvement in comparison time will still make this approach much too slow. We present a new approach for this problem based on what we call *approximate fingerprints*. Approximate fingerprints provide a compact representation of a file such that, with high probability, the fingerprints of two similar files are similar (but not necessarily equal), and the fingerprints of two non-similar files are different. *Sif* works in two different modes: all-against-all and one-against-all. The first mode finds all groups of similar files in a large file system and gives a rough indication of the similarity. The running time is essentially linear in the total size of all files[2] and thus *sif* can be used for large file systems. The second mode compares a given file to a preprocessed *approximate index* of all other files, and determines very quickly (e.g., in 3 seconds for 4000 files of 60MB) all files that are similar to the given file. In both cases, similarity can be detected even if the similar portions constitute as little as 25% of the size of the smaller file.

We foresee several applications for *sif*. The most obvious one is to help in file management. We tested personal file systems and found groups of similar files of many different kinds. The most common ones were different versions of articles and programs (including ‘‘temporary’’ files that became permanent), some of which were generated by the owner of the file system, but some were obtained through the network (in which case, it is much harder to discover that they contain similar information). This information gave us a very interesting view of the file system (e.g., similarity between seemingly unrelated directories) that could not have been obtained otherwise. System administrators can find many uses for *sif*, from saving space to determining whether a version of a given program is already stored somewhere to detecting unauthorized copying. We plan to use *sif* in our work on developing general Internet resource discovery tools [BDMS93]. Identifying similar files (which abound in the Internet FTP space) can improve searching facilities by keeping less to search and giving the users less to browse through. *Sif* can be used as part of a global compression scheme to group similar files together before they are compressed. Yet another important application is in file synchronization for users who keep files on several machines (e.g., work, home, and portable). *Sif* can also be used by professors to detect cheating in homework assignments (although it would be relatively easy to beat it if one wants to put an effort into it), by publishers to detect plagiarism, by politicians to detect many copies of essentially the same form letter they receive from constituents, and so on.

Our notion of similarity throughout this paper is completely syntactic. We make no effort to *understand* the contents of the files. Files containing similar information but using different words will not be considered similar. This approach is therefore very different from the approach taken in the information retrieval literature, and cannot be applied to discover semantic similarities. In a sense, this paper extends the work on approximate string matching (see, for example, our work on *agrep* [WM92a, WM92b]), except that instead of matching strings to large texts, we match parts of large texts to other parts of large texts on a very large scale. Another major difference is that we also solve the all-against-all version of the problem.

A different approach to identifying similarity of source code was taken by Baker [Ba93] who defined two source codes to be similar if one can be obtained from the other by changing parameter names. Baker called this similarity checking *parameterized patern matching* and presented several algorithms to identify similar source

[2] A sort, which is not a linear-time routine, is required, but we do not expect it to dominate the running time unless we compare more than 1-2GB of files.

codes. No other differences in the codes were allowed, however. It would be interesting to combine the two approaches.

2. Approximate Fingerprints

The idea of computing checksums to detect equal files has been used in many contexts. The addition of duplicate detection to DIALOG was hailed as a ‘‘a searcher’s dream come true’’ [Mi90]. The UNIX *sum* program outputs a 16-bit checksum and the approximate size of a given file. This information is commonly used to ensure that files are received undamaged and untouched. A similar notion of ‘‘fingerprinting’’ a file has been suggested by Rabin [Ra81] as a way to protect the file from unauthorized modifications. The idea is essentially to use a function that maps any size string to a number in a reasonably random way (not unlike hashing), with the use of a secret key. Any change to the file will produce a different fingerprint with high probability. Rabin suggested using 63-bit numbers which lead to extremely low probabilities of false positives. (He also designed a special function that has provable security properties.)

But fingerprints and checksums are good only for *exact* equality testing. Our goal is to identify *similar* files. We want to be able to detect that two files are similar even if their similarity covers as little as 25% of their content. Of course, we would like the user to be able to indicate how much similarity is sought. The basic idea is to use fingerprints on several small parts of the file and have several fingerprints rather than just one. But we cannot use fixed parts of a file (e.g., the middle 10%), because any insertion or deletion from that file will make those parts completely different. We need to be able to ‘‘synchronize’’ the equal parts in two different files and to do that without knowing apriori which files or which parts are involved. We will present techniques that are very effective for natural language texts, source codes, and other types of texts that appear in practice.

We achieve the kind of synchronization described above with the use of what we call *anchors*. An anchor is simply a string of characters, and we will use a fixed set of anchors. The idea is to achieve synchronization by extracting from the text strings that start with anchors. If two files contain an identical piece, and if the piece contains an anchor, then the string around the anchor is identical in the two files. For example, suppose that the string *acte* is an anchor. We search the file for all occurrences of *acte*. We may find the word *character* in which *acte* appears. We then scan the text for a fixed number of characters, say 50, starting from *acte*, and compute a checksum of these 50 characters. We call this checksum a *fingerprint*. The same fingerprint will be generated from all files that contain the same 50 characters, no matter where they are located. Of course, *acte* may not appear in the file at all, or it may appear only in places that have been modified, in which case no common fingerprints will be found. The trick is to use several anchors and to choose them such that they span the files in a reasonably uniform fashion. We devised two different ways to use anchors. The first is by analyzing text from many different files and selecting a fixed set of representative strings, which are quite common but not too common. The string *acte* is an example. Once we have a set of anchors, we scan the files we want to compare and search for all occurrences of all anchors. Fortunately, we can do it reasonably quickly using a our multiple-pattern matching algorithm (which is part of agrep [WM92a]). We will not elaborate too much here on this method of anchor selection, because the second method is much simpler.

The second method computes fingerprints of essentially all possible substrings of a certain length and chooses a subset of these fingerprints based on their values. Again, since two equal substrings will generate the same fingerprints, no matter where they are in the text, we have the synchronization that we need. Note that we cannot simply divide the text into groups of 50 bytes and use their fingerprints, because a single insertion at the beginning of the file will shift everything by 1 and cause all groups, and therefore all fingerprints, to be different.

We need to consider *all* 50-byte substrings, including all overlaps. We now present an efficient method to compute all these fingerprints. Denote the text string by $t_1 t_2 \cdots t_n$. The fingerprint for the first 50-byte substring will be

$$F_1 = (t_1 \cdot p^{49} + t_2 \cdot p^{48} + \cdots + t_{50}) \; mod \; M, \quad \text{where } p \text{ and } M \text{ are constants.}$$

The best way to evaluate a polynomial given its coefficients is by Horner's rule:

$$F_1 = (p \cdot ((\; \cdots \; (p \cdot (p \cdot t_1 + t_2) + t_3) \; \cdots \;)) + t_{50}) \; mod \; M.$$

If we now want to compute F_2, then we need only to add the last coefficient and remove the first one:

$$F_2 = (p \cdot F_1 + t_{51} - t_1 \cdot p^{49}) \; mod \; M.$$

We compute a table of all possible values of $(t_i \cdot p^{49}) \; mod \; M$ for all 256 byte values and use it throughout. Overall, computing all fingerprints is proportional to the number of characters but not to the size of the fingerprint. Deciding which fingerprints to select can be done in many ways, the simplest of them is by taking those with the last k bits equal to 0. Approximately one fingerprint out of 2^k characters will be selected. We use a prime number for p, 2^{30} for M, and $k = 8$. Since all selected fingerprints have 8 least significant bits equal to 0, their values should be shifted by 8 before storing them to save space. If the number of files is very large, we may need to use larger fingerprints (i.e., select 2^{31} or 2^{32}) to minimize the number of equal fingerprints by chance.

The second method is easier to use, because the anchors are in a sense universal. They are selected truly at random. It relieves the user from the task of adjusting the anchors to the text. With the first method, anchors that are optimized for Wall Street Journal articles may not be as good for medical articles or computer programs. Anchors for one language may not be good for another language. On the other hand, some users may want to have the ability to fine tune the anchors. For example, with hashing, there is a 1 in 2^k chance (256 in our case) that a string of 50 blanks is selected. If it is, the corresponding fingerprint may appear many times in the file, and it will hardly be representative of the file. The same holds for many other non-representative strings. (We actually encountered that problem; a string of 50 underline symbols turned out to be selected.) One can change the hash function (e.g., by changing p), but there is very little control over the results. One precaution that we take is forbidding overlaps of fingerprints. In other words, once a fingerprint is identified, the text is shifted to the end of it. This way, if, for example, 50 underline symbols form a fingerprint, and the text contains 70 underline symbols, we will not generate 21 duplicate fingerprints.

Both methods are susceptible to bad fingerprints, even for strings that seem representative. The worst example we encountered are fingerprints that are contained in the headers of Postscript files. These headers are large, similar, and ubiquitous; they make many unrelated Postscript files, especially small ones, look very similar. The best solution in this case is to identify the Postscript file when the file is opened and disregard the headers. We discuss handling special files in Section 6.

Although both methods are not perfect, both are good. Having spurious fingerprints is not a major problem as long as there are enough representative fingerprints. Typically, the probability of the same string of 50 bytes appearing in two unrelated files is quite low. And since we require several shared fingerprints we are quite assured of filtering noise. If sufficient number of fingerprints are common to two files then this is a good enough evidence that the two files are similar in some way.

3. Finding Similar Files to a Given File

In this mode, a given file, let's call it the *query file*, is compared to a large set of files that have already been ‘‘fingerprinted.’’ The collection of all fingerprints, which we will denote by *All_Fingers*, is maintained in one large file. With each fingerprint we must associate the file it came from. We do that by maintaining the names of all files that were fingerprinted, and associating with each fingerprint the *number* of the corresponding file. The first thing we do is generate the set *Query_Fingers* of all fingerprints for the query file. We now have to look in *All_Fingers* and compare all fingerprints there to those of *Query_Fingers*. Searching a set is one of the most basic data structure problems and there are many ways to handle it; the most common techniques use hashing or tree structures. In this case, we can also sort both sets and intersect them. But we found that a simple solution using multi-pattern matching was just as effective. We store the fingerprints in *All_Fingers* as we obtained them without providing any other structure, putting one fingerprint together with its file number per line. Then we use *agrep* to search the file *All_Fingers* using *Query_Fingers* as the set of patterns. The output of the search is the list of all common fingerprints to *Query_Fingers* and *All_Fingers*. As long as the set *All_Fingers* is no more than a few megabytes, this search is very effective. (We plan to provide other options for very large indexes.)

Once we have the list of all files containing fingerprints common to the query file, we output those that have more than a given percentage common fingerprints (with default of 50%). This fingerprint percentage number gives a rough estimate for the similarity of the two files. More precisely, it gives an indication of how much of the query file is contained in the file we found. It is interesting to note that this ratio can be greater than 1. That is, the number of common fingerprints to *All_Fingers* and *Query_Fingers* can actually be higher than the total number of fingerprints in *Query_Fingers*. The reason for that is that some fingerprints may appear more than once, and will thus be counted more than once.[3]

We also provide 32-bit checksums for all files to allow exact comparisons. We compute such checksums together with the fingerprints and store them (along with the file sizes in bytes for extra safety and for more information in the output) with the list of files. We also compute the checksum for the query file and determine whether some files are *exactly* equal to it. This whole process normally takes 2-3 seconds.

4. Comparing All Against All

Comparing all files against all other files is more complicated. It turns out that providing a good way to view the output is one of the major difficulties. The output is a set of sets, or a *hypergraph*, with some similarity relationships among the elements. Hypergraphs are very hard to view (see Harel [Ha88]). We discuss here one approach that is an extension of the one-against-all paradigm, and therefore quite intuitive to view. We experimented with other more complicated approaches and we mention one of them briefly at the end.

The input to the problem is now a directory (or several directories), which we will traverse recursively checking all the files below it, and a threshold number *T* indicating how similar we want the files to be. The first stage is identical to the preprocessing described in the previous section. All files are scanned, all fingerprints are recorded in the fingerprint file *All_Fingers*, and the names, checksums, and sizes of all files are recorded in another file. It will be useful in this stage to separate files according to some types (e.g., text, binary, compressed),

[3] In fact, this happened to us the very first time we tested *sif*. The query file was an edited version of a call for papers and it matched with 160% "similarity" another file that happened to contain (unintentionally) *two* slightly different versions of the same call for papers (saved at different times from two mail messages into the same file). Of course, this kind of information is also very useful.

using a program such as Essence [HS93], such that files of different types will not be compared. In the current implementation we use only two types, text files and non-text files.

Given *All_Fingers*, we now sort all fingerprints and collate, for each fingerprint associated with more than one file, the list of all file numbers sharing it. The value of the fingerprint itself is not important any more and it is discarded. An example is shown in Figure 1. Removing the fingerprints that appear in no more than one file reduces the size of *All_Fingers* significantly (for my file system, it was by a factor of 30), so it is much easier to work with. We now have a list of sets and we sort it again, lexicographically, and for sets that appear more than once (meaning the same files share more than one fingerprint) we replace the many copies by one copy and a counter. Figure 2 shows the output of this step, which we denote by *Sorted_Fingers*. Sets with large counters — that is, sets that share significant number of fingerprints — should definitely be part of the output, but how exactly

```
10 1174 129 196 647
10 129 196 647
12 213 30 3023 40 4207 44 649 733 942 962
12 213 30 3023 40 4207 44 649 733 942 962
10 129 196 647
11 212 3021 4203 648 732 76 942 961
12 213 30 3023 40 4207 44 649 733 942 962
11 212 3021 4203 648 732 942 961
10 129 196 647
11 212 3021 4203 648 732 942 961
12 213 30 3023 40 4207 44 649 733 77 942 962
11 212 3021 4203 648 732 76 942 961
```

Figure 1: Typical (partial) output of sets of file numbers that share common fingerprints.

```
1 10 1174 129 196 647
3 10 129 196 647
3 12 213 30 3023 40 4207 44 649 733 942 962
2 11 212 3021 4203 648 732 76 942 961
2 11 212 3021 4203 648 732 942 961
1 12 213 30 3023 40 4207 44 649 733 77 942 962
```

Figure 2: Lines of file numbers that share common fingerprints after sorting and collating.
The number in bold is the counter (the number of fingerprints shared by this group of file numbers).

to organize the output turns out to be a very difficult problem.

We are dealing with sets of sets and as a result many complicated scenarios are possible. The similarity relation as we defined it is not transitive. It is possible that file A is similar to file B, which is similar to file C, but A and C have no similarity. You can also have A similar to B, B similar to C, and A similar to C, but the three of them together, A, B, and C, share no common fingerprints (recall that we allow similarity to correspond to as little as 25% of the files). Or, A, B, C, and D can share 20 fingerprints in common (which is significant), A and C share 40 fingerprints, B and D share 65 fingerprints, A, B, and D share 38 fingerprints, and so on. Does the user really want to see all combinations?

In the first version of *sif* each file *file* that appears somewhere in *Sorted_Fingers* is considered separately. All sets (lines in Figure 2) containing *file* are collected. (This is done while the sets are constructed by associating the corresponding set numbers with each file.) Denote these sets by $S_1, S_2, ..., S_k$ and their counters by $c_1, c_2, ..., c_k$. For each other file that appears in any of the S_i's, the sum of the corresponding c_i's is computed. If that sum, as a percentage of the total number of fingerprints that were found for *file*, is more than the threshold, then the file is considered similar to *file*. The output consists of all similar files to *file*, the similarity for each file, and the size of each file. It is easy at this point to skip files whose size differ substantially from that of *file* (if that is what the user wants to see), or files that fall under some other rules specified by the user; for example, only files with the same suffix may be considered similar. One way to reduce the size of this output without losing significant information is to eliminate duplicate sets. If, for example, 7 files are similar, the list of these files will appear 7 times, one for each of them. The similarity percentages may be different in each instance because they are computed as percentages, but the differences are usually minor. So, to reduce the output, any set of files is output no more than once. An example of a partial output is shown in Figure 3.

```
The following groups of files are similar. Minimum similarity = 25%

R100 /u1/udi/xyz/abc/foo.c 10763
  79 /u1/udi/qwe/ewq/bar.c 10979
  75 /u1/udi/uuu/xxx/foobar.c 9560
```

R indicates the file with which all others are compared. The first number is the percentage of similarity with the R file. The numbers at the end indicate the file sizes. (Except for the file names, this is an actual output.)

Figure 3: An example of a group of similar files as output by *sif*.

The second version of *sif* will include as an option a list of *interesting* similar files, where a set is considered interesting if all members of the set have sufficient number of common fingerprints *and* the set is not a subset of a larger interesting set. The problem of generating interesting sets turns out to be NP-complete, but we devised an algorithm that we hope will work reasonably well in practice.

5. Experience and Performance

As expected, *sif* performs very well on random tests. For example, we took a file containing a C program of size 30K, and made 300 random substitutions (with repetition), each of size 50, thus changing about 50% of the file (but leaving significant chunks unchanged). We then ran *sif* (in the one-against-all mode) setting the threshold at a very low 5%. We ran this experiment 50 times. Each time *sif* found the right file (among 4000 other files of about 60MB) and only it. The similarity that *sif* reported ranged from 37% to 62%, averaging 52%. The average running time for one test (user + system time) was 3.1 seconds (not counting, of course, the time it took originally to build the index). (All experiments were run on a DEC 5000/240 workstation running Ultrix.)

The real question, however, is the performance on real data. We tried *sif* on several file systems. The running time for computing all fingerprints was 3-6 seconds per MB which is in the order of 500MB to 1GB per hour. It takes longer if there are many files and directories and they are small. The sorting of the fingerprints takes from a third to a half of this time (sorting is not a linear-time algorithm, so it takes longer for large number of fingerprints; we tried up to 200MB which generated 800,000 fingerprints). The rest of the algorithm takes much less time.

The first run was on a file system with 2750 (uncompressed) text files of about 40MB. It took 127.4 seconds (user + system time) to generate all fingerprints (and determine that 800 other files are non-text files), 74.3 seconds to sort the fingerprints, and 14.6 seconds to perform the rest of the computation (only 1 second of which depends on the similarity parameter, so changing it will take just one more second literally). Another test was performed on a large collection of ‘‘Frequently Asked Questions’’ (FAQ) files extracted automatically from many newsgroups. For example, two FAQs that were archived under different names (misc.quotes and misc.quotations.sources) turned out to be almost the same.

The next collection of files presented us with almost the worst case. We obtained, through the Alex system [Ca92], a large collection of 21249 README files taken from thousands of ftp sites across the Internet. Their total size was 73MB, which puts the average size at a very low 3K. We found 3620 groups of equal files and 2810 other groups of similar files (the similarity threshold was set at 50%). In many cases it would be impossible to tell that the files are similar by looking only at their names (e.g., draco.ccs.yorku.ca:pub/doc/tmp/read.me and ee.utah.edu:rfc/READ.ME are very similar but not the same). The most challenging experiment was the whole X11R5 distribution (380MB, ˜200MB of which were 21288 ascii files). There were 657 groups of equal files, 3445 groups of similar files with 25% similarity threshold, and 2915 groups with 50% threshold. For most of the experiments described here (with the exception of the FAQ files), we used the first method with a list of very frequent anchors to allow high precision. On the average, one fingerprint was generated for about every 200 bytes of text.

The space required to hold the fingerprints for the one-against-all tool is currently about 5% of the total space. By a better encoding of the fingerprints this figure will be reduced to about 2%.

6. Future Work

We are extending *sif* in four areas. The first is adapting the fingerprint generation to file types. We already mentioned Postscript files, for which the headers should be excluded from generating fingerprints. This is relatively easy to do because Postscript files and their headers can be easily recognized. Another example is removing formatting statements from files (e.g., troff or TeX files). Other types present more difficult problems. The two most notable types we would like to handle are executables and compressed files. Both types are very sensitive to change. Adding one statement to a program can change all addresses in the executable. In compressed files that translate strings to dictionary indices (e.g., Lempel-Ziv compression) one change in the dictionary can change all indices. The challenge is to find invariants and generate fingerprints accordingly (e.g., ignoring addresses altogether, exploring the relationships between dictionary indices).

The second area is allowing different treatments of small and large files. Currently, we treat all files equally. A noble idea, but sometimes not effective. We figured that at least 5-10 shared fingerprints are needed for a strong evidence of similarity (the exact number depends on the file type). If we seek 50% similarity, then each file needs at least 10-20 fingerprints. In the current setting (which can be easily changed), *sif* generates 3-4 fingerprints per 1K, which makes *sif* only marginally effective for files of less than 5K. On the other hand, a file of 1MB can generate 4,000 fingerprints. If we adjust the number of fingerprints to the size of the file, we may lose the ability to determine whether a small file is contained in a large file, but this is not always needed. Adjusting the number of fingerprints is easy with both methods of anchor selection (by decreasing the number of anchors with the first method or increasing the value of *k* with the second method).

The third area is providing convenient facilities for comparing two directories. Current tools for comparing directories (such as the system V *dircpm* program) rely on file names and checksums. For files that do not match exactly, *dircpm* will only list them as not being equal. Comparing two directories based on content can be done with essentially the same tools we already have, but we need to allow the use of the filenames in the similarity measure.

The fourth area is customizing the output generation. The most difficult problem is how to provide users with flexible means to extract only the similarity they seek. We could attach a large number of options to *sif* (a popular UNIX solution) and provide hooks to external filter routines (such as ones using Essence [HS93]). We would like to have something more general. The problem of finding *interesting* similar files (as defined at the end of Section 4) is very intriguing from a practical and also a theoretical point of view.

Acknowledgements

We thank Vincent Cate for supplying us with the README files collected by Alex. Richard Schroeppel suggested the use of hashing to select anchors. Gregg Townsend wrote the filter that collects the FAQ files.

References

[Ba93]

Baker, B. S., ‘‘A theory of parameterized patern matching: Algorithms and applications,’’ *25th Annual ACM Symposium on Theory of Computing*, San Diego, CA (May 1993), pp. 71–80.

[BDMS93]

Bowman, C. M., P. B. Danzig, U. Manber, and M. F. Schwartz, ‘‘Scalable Internet Resource Discovery: Research Problems and Approaches,’’ University of Colorado Technical Report# CU-CS-679-93 (October 1993), submitted for publication.

[Ca92]

Cate, V., ‘‘Alex — a global filesystem,’’ *Proceedings of the Usenix File Systems Workshop*, pp. 1–11, May 1992.

[Ha88]

Harel D., ‘‘On Visual Formalisms,’’ *Communications of the ACM*, **31** (May 1988), pp. 514–530.

[HS93]

Hardy D. R., and M. F. Schwartz, ‘‘Essence: A resource discovery system based on semantic file indexing,’’ *USENIX Winter 1993 Technical Conference*, San Diego (January 1993), pp. 361–374.

[Mi90]

Miller C., ‘‘Detecting duplicates: a searcher’s dream come true’’ *Online*, **14**, 4 (July 1990), pp. 27–34.

[Ra81]

Rabin, M. O., ‘‘Fingerprinting by Random Polynomials,’’ Center for Research in Computing Technology, Harvard University, Report TR-15-81, 1981.

[WM92a]

Wu S. and U. Manber, ‘‘Agrep — A Fast Approximate Pattern-Matching Tool,’’ *Usenix Winter 1992 Technical Conference*, San Francisco (January 1992), pp. 153–162.

[WM92b]

Wu S., and U. Manber, ‘‘Fast Text Searching Allowing Errors,’’ *Communications of the ACM* **35** (October 1992), pp. 83–91.