CHAPTER 13

Algorithms for Delta Compression and Remote File Synchronization

TORSTEN SUEL
NASIR MEMON

OVERVIEW

Delta compression and remote file synchronization techniques are concerned with efficient file transfer over a slow communication link in the case where the receiving party already has a similar file (or files). This problem arises naturally, e.g., when distributing updated versions of software over a network or synchronizing personal files between different accounts and devices. More generally, the problem is becoming increasingly common in many network-based applications where files and content are widely replicated, frequently modified, and cut and reassembled in different contexts and packagings.

In this chapter, we survey techniques, software tools, and applications for delta compression, remote file synchronization, and closely related problems. We first focus on delta compression, where the sender knows all the similar files that are held by the receiver. In the second part, we survey work on the related, but in many ways quite different, problem of remote file synchronization, where the sender does not have a copy of the files held by the receiver.

13.1 INTRODUCTION

Compression techniques are widely used in computer networks and data storage systems to increase the efficiency of data transfers and reduce space requirements on the receiving device. Most techniques focus on the problem of compressing individual files or data streams of a certain type (text, images, audio). However, in today's network-based environment it is often the case that files and content are widely replicated, frequently modified, and cut and reassembled in different contexts and packagings.

Thus, there are many scenarios where the receiver in a data transfer already has an earlier version of the transmitted file or some other file that is similar, or where several similar files are transmitted together. Examples are the distribution of software packages when the receiver already has an earlier version, the transmission of a set of related documents that share structure or content (e.g., pages from the same web site), or the remote synchronization of a database. In these cases, we should be able to achieve better compression than that obtained by individually compressing each file. This is the goal of the delta compression and remote file synchronization techniques described in this chapter.

Consider the case of a server distributing a software package. If the client already has an older version of the software, then an efficient distribution scheme would send only a patch to the client that describes the differences between the old and the new versions. In particular, the client would send a request to the server that specifies the version number of the outdated version at the client. The server then looks at the new version of the software, and at the outdated version that we assume is available to the server, and computes and sends out a "patch" that the client can use to update its version. The process of computing such a patch of minimal size between two files is called *delta compression*, or sometimes also *delta encoding* or *differential compression*.

Of course, in the case of software updates these patches are usually computed off-line using well-known tools such as *bdiff*, and the client can then choose the right patch from a list of files. However, *bdiff* is not a very good delta compressor, and there are other techniques that can result in significantly smaller patch size.

When distributing popular software that is updated only periodically, it seems realistic to assume that the server has copies of the previous versions of the software which it can use to compute a delta of minimal size. However, in other scenarios, the server may have only the new version, due to the overhead of maintaining all outdated versions or due to client-side or third-party changes to the file. The *remote file synchronization* problem is the problem of designing a protocol between the two parties for this case that allows the client to update its version to the current one while minimizing communication between the two parties.

13.1.1 Problem Definition

More formally, we have two strings (files) f_{new}, $f_{old} \in \Sigma^*$ over some alphabet Σ (most methods are character/byte oriented) and two computers, C (the client) and S (the server), connected by a communication link.

- In the *delta compression* problem, C has a copy of f_{old} and S has copies of both f_{new} and f_{old}, and the goal for S is to compute a file f_δ of minimum size, such that C can reconstruct f_{new} from f_{old} and f_δ. We also refer to f_δ as a *delta* of f_{new} and f_{old}.
- In the *remote file synchronization* problem, C has a copy of f_{old} and S has only a copy of f_{new}, and the goal is to design a protocol between the two parties that results in C holding a copy of f_{new}, while minimizing the communication cost.

We also refer to f_{old} as a *reference file* and to f_{new} as the *current file*. For a file f, we use $f[i]$ to denote the ith symbol of f, $0 \leq i < |f|$, and $f[i, j]$ to denote the block of symbols from i until (and including) j. We note that while we introduce the delta compression problem here in a networking context, another important application area is in the space-efficient storage of similar files, e.g., multiple versions of a document or a software source—in fact, delta compression techniques were first introduced in the context of software revision control systems. We discuss such applications in Subsection 13.2.1, and it should be obvious how to adapt the definitions to such a scenario. Also, while our definition assumes a single reference file, f_{old}, there could be

several similar files that might be helpful in communicating the contents of f_{new} to the client, as discussed later.

In the case of the file synchronization problem, many currently known protocols [16, 35, 49] consist of a single round of communication, where the client first sends a request with a limited amount of information about f_{old} to the server, and the server then sends an encoding of the current file to the client. In the case of a multiround protocol, a standard model for communication costs based on latency and bandwidth can be employed to measure the cost of the protocol. A simple model commonly used in distributed computing defines the cost (time) for sending a message of length m as $L + m/B$, where L is the latency (delay) and B is the bandwidth of the connection.

There are several other interesting algorithmic problems arising in the context of delta compression and remote file synchronization that we also address. For example, in some cases there is no obvious similar file, and we may have to select the most appropriate reference file(s) from a collection of files. In the case of remote file synchronization, we would often like to estimate file similarity efficiently over a network. Finally, the data to be synchronized may consist of a large number of small records, rather than a few large files, necessitating a somewhat different approach.

13.1.2 Content of This Chapter

In this chapter, we survey techniques, software tools, and applications for delta compression and remote file synchronization. We consider scenarios in networking as well as storage. For simplicity, most of the time, we consider the case of a single reference file, though the case of more than one file is also discussed. We also discuss related problems such as how to select appropriate reference files for delta compression, how to estimate the similarity of two files, and how to reconcile large collections of record-based data.

In Section 13.2, we focus on delta compression, where the sender knows all the similar files that are held by the receiver. In Section 13.3, we survey work on the related, but in many ways quite different, problem of remote file synchronization, where the sender does not have a copy of the files held by the receiver. Finally, Section 13.4 offers some concluding remarks.

13.2 DELTA COMPRESSION

We now focus on the delta compression problem. We first describe some important application scenarios that benefit from delta compression. In Subsection 13.2.2 we give an overview of delta compression approaches, and Subsection 13.2.3 describes in more detail a delta compressor based on the Lempel–Ziv (LZ) compression algorithm. Experimental results for a few delta compressors are given in Subsection 13.2.4. Finally, we discuss the problems of space-constrained delta compression and of choosing good reference files in Subsections 13.2.5 and 13.2.6, respectively.

13.2.1 Applications

As mentioned above, most of the applications of delta compression are aimed at reducing networking or storage costs. We now describe a few of them in more detail.

13.2.1.1 Software Revision Control Systems

As mentioned in the Introduction, delta compression techniques were pioneered in the context of systems used for maintaining the revision history of software projects and other documents [8, 40, 45]. In these systems, multiple, often almost identical, versions of each object must be

stored in order to allow the users to retrieve past versions. For example, the RCS (Revision Control System) package [45] uses the *diff* delta compressor to reduce storage requirements. For more discussion on delta compression in the context of such systems, and an evaluation of different compressors, see the work of Hunt *et al.* [25].

13.2.1.2 Delta Compression at the File System Level

The Xdelta File System (XDFS) of MacDonald [29] aims to provide efficient support for delta compression at the file system level using a delta compressor called *xdelta*. This allows the efficient implementation of revision control systems, as well as some other applications listed here, on top of XDFS.

13.2.1.3 Software Distribution

As described in the example in the Introduction, delta compression techniques are used to generate software patches that can be efficiently transmitted over a network in order to update installed software packages.

13.2.1.4 Exploring File Differences

Techniques from delta compression can be used to visualize differences between different documents. For example, the well-known *diff* utility displays the differences between two files as a set of edit commands, while the *HtmlDiff* and *topblend* tools of Chen *et al.* [14] visualize the difference between two HTML documents.

13.2.1.5 Improving HTTP Performance

Several approaches have been proposed that employ delta compression to improve the latency for web accesses, by exploiting the similarity between current and outdated versions of a web page, and between different pages on the same web site. In particular, Banga *et al.* [6] and Mogul *et al.* [34] propose a scheme called *optimistic delta* in which a caching proxy attempts to hide the latency of server replies by first sending a potentially outdated cached version of a page to the client and then if necessary a small corrective patch once the server replies. In another approach, a client that already has an old version of a page in his cache sends a tag identifying this version to a proxy (or server) as part of the HTTP request; the proxy then sends the delta between the old and the current version to the client [18, 24]. This can significantly decrease the amount of data sent to the client and is thus more appropriate for clients connected via low-bandwidth links such as cellular modems.

It has also been observed that web pages on the same server often have a high degree of similarity (due to common layout and menu structure) that could be exploited with delta compression techniques. In particular, Chan and Woo [13] propose to identify candidate pages that are likely to be good reference files for delta compression by looking for URLs that share a long common prefix with the requested one. Other work [19] proposes a similar idea for dynamic pages, e.g., different stock quotes from a financial site, that share a lot of content.

13.2.1.6 Efficient Web Page Storage

The similarities between different versions of the same page or different pages on the same web site could also be used for increased storage efficiency in large-scale web repositories such as the Internet Archive[1] or the Stanford WebBase [23]. In particular, the Internet Archive aims to preserve multiple versions of each page, but these pages are currently stored without the use of delta compression techniques. A delta-compressed archive could be implemented on top of XDFS [29],

[1] `http://www.archive.org`.

but a specialized implementation with a more optimized delta compressor might be preferable in this case. If we also plan to exploit similarities between different pages, then the problem of selecting appropriate reference pages arises. While the general formulation of this problem, discussed further below, is quite challenging, specialized techniques based on URL prefix matching [13] plus separate detection of mirrors [9] and replicated collections [15] may suffice in practice.

13.2.2 Fundamentals

Recall that in the delta compression problem, we have two files, f_{old} and f_{new}, and the goal is to compute a file f_δ of minimum size, such that one can reconstruct f_{new} from f_{old} and f_δ. Early work on this problem was done within the framework of the *string-to-string correction problem*, defined in [50] as the problem of finding the best sequence of insert, delete, and update operations that transform one string to another. Approaches for solving this problem were based on finding the largest common subsequence of the two strings using dynamic programming and adding all remaining characters to f_{new} explicitly. However, the string-to-string correction problem does not capture the full generality of the delta compression problem as illustrated in the examples given in the previous subsection. For example, in the string-to-string correction problem, it is implicitly assumed that the data common to f_{new} and f_{old} appear in the same order in the two files. Furthermore, the string-to-string correction approach does not account for substrings in f_{old} appearing in f_{new} several times.

To resolve these limitations, Tichy [44] defined the *string-to-string correction problem with block moves*. A *block move* is a triple (p, q, l) such that $f_{old}[p, \ldots, p+l-1] = f_{new}[q, \ldots, q+l-1]$. It represents a non-empty common substring of f_{old} and f_{new} which is of length l. Given f_{old} and f_{new}, the file f_δ can then be constructed as a minimal *covering set* of block moves such that every element $f_{new}[i]$ that also appears in f_{old} is included in exactly one block move. It can be further argued that an f_δ constructed from the longest common subsequence approach mentioned earlier is just a special case of a covering set of block moves. The minimality condition then ensures the superiority of the block-moves approach to the longest common subsequence approach.

The question then arises—how does one construct an optimal f_δ given f_{old} and f_{new}? Tichy [44] also showed that a greedy algorithm results in a minimal cover set and that an f_δ based on a minimal cover set of block moves can be constructed in linear space and time using suffix trees. Unfortunately, the multiplicative constant in the space complexity makes the approach impractical. A more practical approach uses hash tables with linear space but quadratic time worst case complexity [44].

The block-moves framework described above represented a fundamental shift in the development of delta compression algorithms. While earlier approaches used an edit-based approach—i.e., construct an optimal sequence of edit operations that transform f_{old} into f_{new}, the block-moves algorithms use a *copy-based* approach—i.e., express f_{new} as an optimal sequence of copy operations from f_{old}.

The Lempel–Ziv string compression algorithms [52, 53] popularized in the 1980s yield another natural framework for realizing delta compression techniques based on the copy-based approach. In particular, the LZ77 algorithm can be viewed as a sequence of operations that involve replacing a prefix of the string being encoded with a reference to an identical previously encoded substring. In most practical implementations of LZ77, a greedy approach is used whereby the longest matching prefix found in the previously encoded text is replaced by a copy operation.

Thus, delta compression can be viewed as simply performing LZ77 compression with the file f_{old} representing "previously encoded" text. In fact, nothing prevents us from also including the part of f_{new} which has already been encoded in the search for a longest matching prefix. A few additional changes are required to get a really practical implementation of a LZ77-based delta compression technique. Several such implementations have been designed over the past decade but

the basic framework is the same. The difference lies mostly in the encoding and updating mechanisms employed by each. In the next subsection we describe one such technique in more detail.

13.2.3 LZ77-Based Delta Compressors

The best general-purpose delta compression tools are currently copy-based algorithms based on the Lempel–Ziv [52] approach. Example of such tools are *vdelta* and its newer variant *vcdiff* [25], the *xdelta* compressor used in *XDFS* [29], and the *zdelta* tool [47].

We now describe the implementation of such a compressor in more detail, using the example of *zdelta*. The *zdelta* tool is based on a modification of the *zlib* compression library of Gailly [21], with some additional ideas inspired by *vdelta*, and anyone familiar with *zlib*, *gzip*, and other Lempel–Ziv-based algorithms should be able to easily follow the description. Essentially, the idea in *zdelta*, also taken in the *vcdiff (vdelta)* and *xdelta* algorithms, is to encode the current file by pointing to substrings in the reference file as well as in the already encoded part of the current file.

To identify suitable matches during coding, we maintain two hash tables, one for the reference file, T_{old}, and one for the already coded part of the current file, T_{new}. The table T_{new} is essentially handled the same way as the hash table in *gzip*, where we insert new entries as we traverse and encode f_{new}. The table T_{old} is built beforehand by scanning f_{old}, assuming f_{old} is not too large. When looking for matches, we search in both tables to find the best one. Hashing of a substring is done based on its first three characters, with chaining inside each hash bucket.

Let us assume for the moment that both reference and current files fit into the main memory. Both hash tables are initially empty. The basic steps during encoding are as follows. (Decoding is fairly straightforward given the encoding.)

1. Preprocessing the Reference File:

For $i = 0$ to $len(f_{old}) - 3$:

(a) Compute $h_i = h(f_{old}[i, i+2])$, the hash value of the first three characters starting from position i in f_{old}.

(b) Insert a pointer to position i into hash bucket h_i of T_{old}.

2. Encoding the Current File:

Initialize pointers $p_1, \ldots, p_k$ to zero, say, for $k = 2$.

Set $j = 0$.

While $j \leq len(f_{new})$:

(a) Compute $h_j = h(f_{new}[j, j+2])$, the hash value of the first three characters starting from position j in f_{new}.

(b) Search hash bucket h_j in both T_{old} and T_{new} to find a "good match," i.e., a substring in f_{old} or the already encoded part of f_{new} that has a common prefix of maximum length with the string starting at position j of f_{new}.

(c) Insert a pointer to position j into hash bucket h_j of T_{new}.

(d) If the match is of length at least 3, encode the position of the match relative to j if the match is in f_{new} and relative to one of the pointers p_i if the match is in f_{old}. If several such matches of the same length were found in (b), choose the one that has the smallest relative distance to position j in f_{new} or to one of the pointers into f_{old}. Also encode the length of the match and which pointer was used as reference. Increase j by the length of the match, and possibly update some of the pointers p_i.

(e) If there is no match of length at least 3, write out character $f_{new}[j]$ and increase j by 1.

There are a number of additional details to the implementation. First, we can choose a variety of policies for updating the pointers p_i. The motivation for these pointers, as used in *vdelta*, is that in many cases the location of the next match from f_{old} is a short distance from the previous one,

especially when the files are very similar. Thus, by updating one of the pointers to point to the end of the previous match, we hope to very succinctly encode the location of the next match. In general, smart pointer movement policies might lead to additional moderate improvements over the existing tools.

Another important detail concerns the method used to encode the distances, match lengths, pointer information, and characters. Here, *zdelta* uses the Huffman coding facilities provided by *zlib*, while *vdelta* uses a byte-based encoding that is faster but less compact. In contrast, *xdelta* does not try any clever encoding at all, but leaves it up to the user to apply a separate compression tool on the output.

A very important issue is what to do if the hash tables develop very long chains. For T_{new} this can be handled by simply evicting the oldest entries whenever a bucket grows beyond a certain size (as done in *gzip*), but for T_{old} things are more complicated. Note that full buckets can happen for two reasons. First, if f_{old} is very large, then all buckets of T_{old} may become large. This is handled in *vdelta* and *zdelta* by maintaining a "window" of fixed size (say, 32 or 64 kB) into f_{old}. Initially, the window is at the start of the file, and as the encoding proceeds we slide this window through f_{old} according to the positions where the best matches were recently found. A simple heuristic, employed by *zdelta*, uses a weighted average of the most recently used matches to determine when it is time to slide the window by half the window size. In that case, we remove all entries that are outside the new window from T_{old}, and add the new entries that are now inside the window. A more sophisticated scheme, used in *vcdiff*, computes fingerprints on blocks of a certain size (e.g., 1024 bytes) and then chooses a window position in f_{old} that maximizes the similarity with the currently considered area of f_{new}. In general, the problem of how to best move the window though f_{old} is difficult and not really resolved, and the best movement may not be sequential from start to end.

Second, even if f_{old} is small, or during one window position, some buckets can become very large due to a frequently occurring substring. In this case, it is not clear which entries should be evicted: those at the beginning or at the end of the file or current window. This issue is again related to the problem of finding and exploiting the pattern of matches in the reference file, which depends on the relation between the reference file and the current file, and a good solution is not yet apparent.

13.2.4 Some Experimental Results

We now show a few experimental results to give the user a feel for the compression performance of the available tools. In these results, we compare *xdelta*, *vcdiff*, and *zdelta* on two different sets of files:

1. The *gcc* and *emacs* data sets used in the performance study in [25], consisting of versions 2.7.0 and 2.7.1 of *gcc* and 19.28 and 19.29 of *emacs*, The newer versions of *gcc* and *emacs* consist of 1002 and 1286 files, respectively.
2. A set of artificially created files that model the degree of similarity between two files. In particular, we created two completely random files f_0 and f_1 of fixed length, and then performed delta compression between f_0 and another file f_m created by a "blending" procedure that copies text from either f_0 or f_1 according to a simple Markov process. By varying the parameters of the process, we can create a sequence of files f_m with similarity ranging from 0 ($f_m = f_1$) to 1 ($f_m = f_0$) on a non-linear scale.[2]

[2] More precisely our process has two states, s_0, where we copy a character from f_0, and s_1, where we copy a character from f_1, and two parameters, p, the probability of staying in s_0, and q, the probability of staying in s_1. In the experiments, we set $q = 0.5$ and vary p from 0 to 1. Clearly, a complete evaluation would have to look at several settings of q to capture different granularities of file changes.

Table 13.1 Compression Results for *gcc* and *emacs* Data Sets (sizes in kilobytes and times in seconds)

	gcc size	gcc time	emacs size	emacs time
Uncompressed	27,288	—	27,326	—
gzip	7,479	24/30	8,191	26/35
xdelta	461	20	2,131	29
vcdiff	289	33	1,821	36
zdelta	250	26/32	1,465	35/42

All runs were performed on a Sun E450 server with two 400-MHz UltraSparc IIe processors and 4 GB of main memory, with the data stored on a 10,000-rpm SCSI disk. We note that only one CPU was used during the runs and that memory consumption was not significant. (We also did multiple runs for each file in the collections and discarded the first one—the main result of this setup is to minimize disk access costs, thus focusing on the CPU costs of the different methods.)

For the *gcc* and *emacs* data sets, the *uncompressed* and *gzip* numbers are with respect to the newer releases. We see from the results that delta compression achieves significant improvements over *gzip* on these files, especially for the very similar *gcc* files (see Table 13.1). Among the delta compressors, *zdelta* gets the best compression ratio, mainly due to the use of Huffman coding instead of byte-based coding. The *xdelta* compressor performs worst in these experiments. As described in [29], *xdelta* aims to separate differencing and compression, and thus a standard compressor such as *gzip* can be applied to the output of *xdelta*. However, in our experiments, subsequent application of *gzip* did not result in any significant improvement on these data sets.

Concerning running times, all three delta compressors are only slightly slower than *gzip*, with *xdelta* coming closest (see Table 13.1). Note that for *gzip* and *zdelta* we report two different numbers that reflect the impact of the input/output method on performance. The first, lower number gives the performace using direct access to files, while the second number is measured using Standard I/O. The number for *vcdiff* is measured using Standard I/O, while *xdelta* uses direct file access. Taking these differences into account, all delta compressors are within at most 20% of the time for *gzip*, even though they must rocess two sets of files instead of just one as in *gzip*.

Looking at the results for different file similarities, we see the same ordering (see Fig. 13.1). Not surprisingly, when files are very different, delta compression does not help at all, while for almost identical files, all methods do quite well. However, we see that *vcdiff* and *zdelta* give benefits even for only slightly similar files for which *xdelta* does not improve over *gzip*. (Note that *gzip* itself does not provide any benefits here due to the incompressibility of the files.) We also see that the running times for the delta compressors decrease as file similarity increases (see Fig. 13.2); this is due to the increasing lengths of the matches found in the referene files (which decrease the number of searches in the hash tables). This effect largely explains why the delta compressors are almost as fast as *gzip* on collections with large similarity such as *gcc* and *emacs*; for files with low degrees of similarity, the three delta compressors take about 60 to 100% longer than *gzip*.

13.2.5 Space-Constrained Delta Compression

As described in Subsection 13.2.2, the greedy algorithm of Tichy [44] results in an optimal set of block moves. To show how these block moves are determined and encoded in a practical implementation, in Subsection 13.2.3 we discussed an implementation based on an LZ77-like

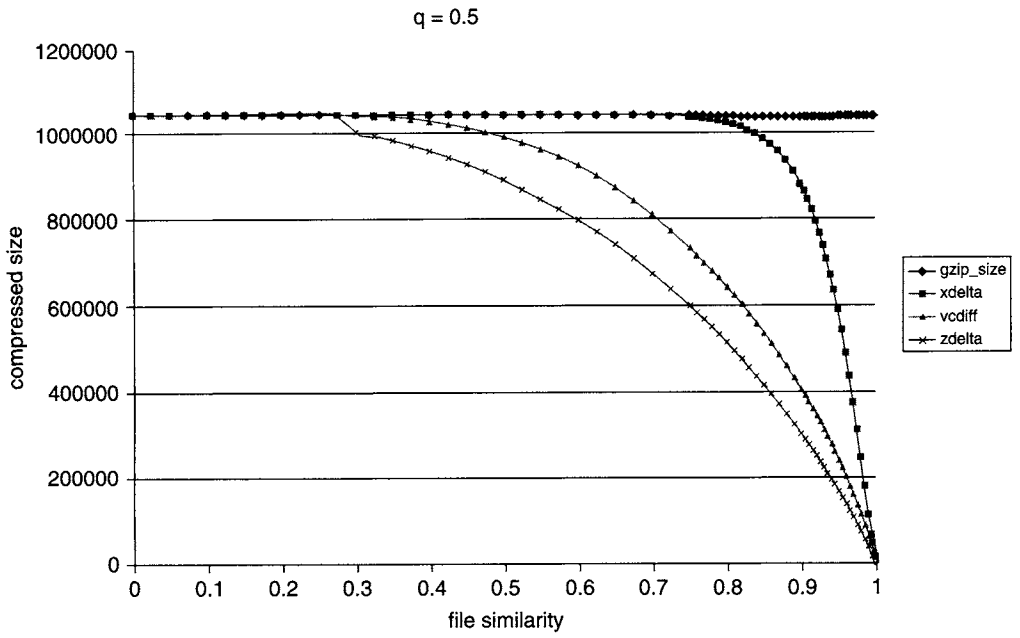

FIGURE 13.1
Compression versus file similarity (in kilobytes).

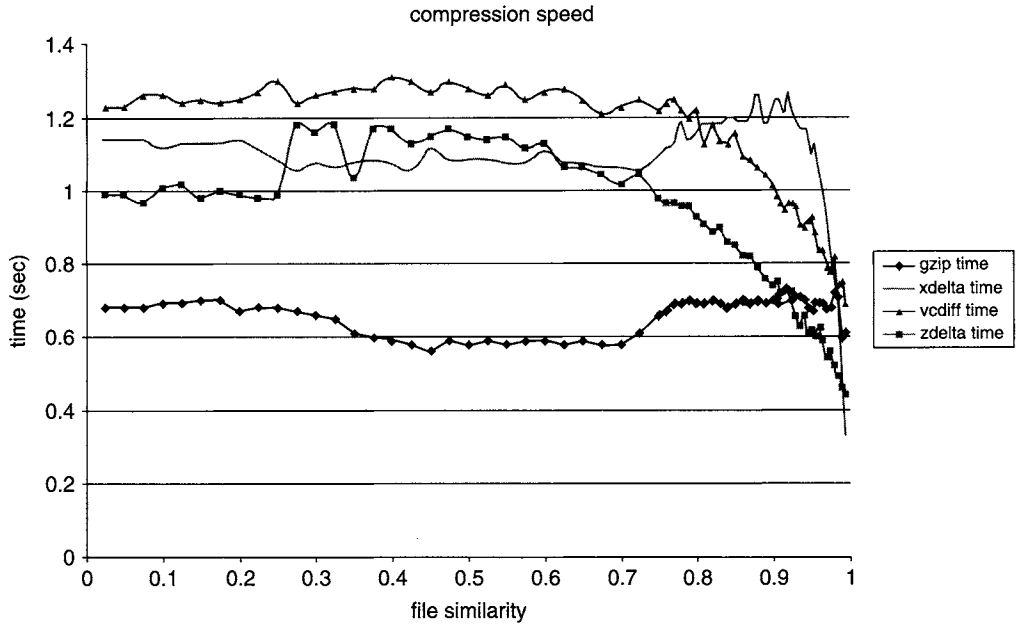

FIGURE 13.2
Running time versus file similarity (in seconds).

framework. However, these algorithms can give very poor performance when limited memory resources are available to the compressor or decompressor. In [3] Ajtai *et al.* looked at the problem of delta compression under various resource constraints. They first examined delta compression in linear time and constant space, which is relevant when the files f_{old} and f_{new} are too large to fit in memory. A simple solution in this case would be to restrict the search for the longest prefix in f_{old} to the forward direction only. That is, when looking for a match we ignore the part of f_{old} that precedes the end of the substring that was just encoded. However, this results in significantly suboptimal compression when substrings occur in different order in f_{old} and f_{new}.

To alleviate this problem Ajtai *et al.* [3] propose a *correcting one-pass* algorithm which utilizes a buffer that holds all copy commands and performing corrections on these commands later when better matches are found. Two types of corrections are made. *Tail corrections* occur when, after a copy command from a previously unencoded part of f_{old} is inserted, the algorithm attempts to extend the matching string *backward* in both f_{old} and f_{new}. If such matches are found going in a backward direction, there is potential for replacing the previous copy commands by integrating them into the current copy command. The second type of correction, called *general correction*, occurs if a matching substring M is found that is already encoded in f_{new}. In this case, the algorithm tries to determine whether the earlier encoding of M can be further compacted now that M can be encoded by a single copy command. Further, to limit the space consumed by the hash tables that store substring locations, they use a technique called *checkpointing* that restricts the locations of a substring that are inserted into the hash table to a small but carefully selected number. The result of these extensions is a delta compression technique that is practical in terms of time and space complexity, works with arbitrary size files, and yields good compression.

13.2.6 Choosing Reference Files

In some applications, delta compression performance depends heavily on the choice of appropriate reference files. For example, to compress a set of related files, we need to choose for each file one or several reference files that have significant similarity to it; each reference file itself can also be compressed this way, provided that no cycles are created. In the case of a single reference file per compressed file, this problem is equivalent to finding an *optimum branching* in a corresponding directed graph where each edge (i, j) has a weight equal to size of the delta of i with respect to reference file j. This problem can be solved in time quadratic in the number of documents [12, 42], but the approach suffers from two drawbacks: First, the solution may contain very long chains of documents that must be accessed in order to uncompress a particular file. Second, for large collections the quadratic time becomes unacceptable, particularly the cost of computing the appropriate weights of the edges of the directed graph.

If we impose an upper bound on the length of the reference chains, then finding the optimum solution becomes NP Complete [43]. If we allow each file to be compressed using more than one reference file, then this problem can be reduced to a generalization of optimum branching to hypergraphs and has been shown NP Complete even with no bound on the length of chains [2].

Some experiments on small web page collections using minimum branching and several faster heuristics are given in [38], which show significant differences in compression performance between different approaches. For very large collections, general document clustering techniques such as [10, 22, 30], or specialized heuristics such as [9, 13, 15] for the case of web documents, could be applied. In particular, Chan and Woo [13] demonstrate that there is significant benefit in choosing more than one reference page to compress a web page.

One example of long reference chains arises when dealing with many different versions of the same file, such as in a revision control system. In this case, the choice of the reference file that minimizes the delta is usually obvious, but this choice would make retrieval of very old versions

quite expensive.[3] Several techniques have been proposed for dealing with this problem [11, 29, 45], by creating a limited number of additional "shortcuts" to older versions.

13.3 REMOTE FILE SYNCHRONIZATION

In this section, we focus on the remote file synchronization problem, i.e., the case where the server does not have access to the reference file. This obviously changes the problem significantly, and the known algorithms for this problem are quite different from those for delta compression. We discuss the two main known approaches for file synchronization: (i) a practical approach based on string fingerprints implemented by the *rsync* algorithm that does not achieve any provable near-optimal bounds and (ii) an approach based on colorings of hypergraphs that achieves provable performance bounds under certain models for file distance, but that seems unsuitable in practice. We also discuss the closely related problems of how to estimate file similarity and how to reconcile sets of records in a database.

We first describe a few application scenarios for file synchronization. In Subsection 13.3.2 we describe the *rsync* algorithms, and Subsection 13.3.3 gives some experimental results comparing *rsync* and delta compressor performance. Subsection 13.3.4 presents some general theoretical bounds, and Subsection 13.3.5 discusses provable results for specific distance measures. Finally, Subsection 13.3.6 discusses how to estimate the distance between two files, while Subsection 13.3.7 looks at the problem of reconciling large sets of data records.

13.3.1 Applications

The applications for file synchronization are similar to those for delta compression. Synchronization is more general in that it does not require knowledge of the reference file; on the other hand, delta compression tends to significantly outperform synchronization in terms of compression ratio. There are several reasons that the server may not have the reference file, such as the space, disk access, or software overhead of maintaining and retrieving old versions of files as references, changes to the file at the client or a third party, or later deletion (eviction) of old versions at the server. Some typical scenarios are shown in the following.

13.3.1.1 Synchronization of User Files

There are a number of software packages such as *rsync* [48, 49] Microsoft's *ActiveSync*, Puma Technologies' *IntelliSync*, or Palm's *HotSync* that allow "synchronization" between desktops, mobile devices, or web-accessible user accounts. In this scenario, files or records can be updated by several different parties, and time stamps may be used to determine which version on which device is the most recent.

We note that there are several challenges for these tools. For data in the form of files, we have the remote file synchronization problem already defined in the Introduction, where we would like to avoid transferring the entire file. For data consisting of large sets of small records, e.g., addresses or appointments on a handheld device, the problem is how to identify those records that have changed without sending an individual fingerprint or time stamp for each record. This problem, modelled as a *set reconciliation* problem in [33], is discussed in Subsection 13.3.7. Many existing packages transfer the entire item if any change has occurred, which is reasonable for small record-based data, but not for larger files. In addition, there is also the general and nontrivial problem of defining the proper semantics for file system synchronization; see [5] for a detailed discussion.

[3] Note that these systems often compress older versions with respect to newer ones.

13.3.1.2 Remote Backup of Massive Data Sets

Synchronization can be used for remote backup of data sets that may have changed only slightly between backups [48]. In this case, the cost of keeping the old version at the server is usually prohibitive, making delta compression techniques inefficient. (See also [11] for an approach that adapts delta compression to this case.)

13.3.1.3 Web Access

File synchronization has also been considered for efficient HTTP transfer between clients and a server or proxy.[4] The advantage is that the server does not have to keep track of the old versions held by the clients and does not need to fetch such versions from disk upon a request. However, as shown in Subsection 13.3.3, file synchronization techniques achieve significantly worse compression ratios than delta compressors and thus typically provide benefits only for files that are "very similar." (We are not aware of any study quantifying these advantages and disadvantages for HTTP transfer.)

13.3.1.4 Distributed and Peer-to-Peer Systems

Synchronization can be used to update highly distributed data structures, such as routing tables, name services, indexes, or replication tables. A significant amount of recent work has looked at highly distributed and peer-to-peer systems and services. We expect interesting applications of file synchronization and set reconciliation to arise in this context, particularly in peer-to-peer systems where nodes are often unavailable for significant periods of time and thus must update their data structures upon rejoining the system.

13.3.2 The *rsync* Algorithm

We now describe the algorithm employed by the widely used *rsync* file synchronization tool[5] of Tridgell and MacKerras (see [48, 49]). A similar approach was also proposed by Pyne in a U.S. Patent [39]. For intuition, consider the following simple problem that captures some of the challenges.

Assume that two parties communicate via telephone, with each party holding a copy of a book. Now suppose that the two copies could differ in a few places. How can the two parties find out if the two books are identical or not, and if not where and how they exactly differ, without reading an entire book over the phone? The answer to the first question is simple: by computing a checksum (e.g., MD5) for each book and comparing the two checksums, it can be decided if the books are identical or not. The answer to the second question, however, is more difficult. A first naive approach would partition the book into two blocks, the first and second halves of the book, and then recurse on those blocks where the checksums differ, until the precise locations of the differences are found. However, this approach fails in the simple case where one book contains an additional word at the beginning, thus destroying the alignments of all the block boundaries between the two books. Thus, a more careful approach is needed, although the basic idea of using checksums on blocks is still relevant.[6]

We now describe the refined *rsync* algorithm. Essentially, the idea is to solve the alignment problem by computing blockwise checksums for the reference file and comparing these checksums

[4] See, e.g., the *rproxy* project at `http://rproxy.samba.org/`.

[5] Available at `http://rsync.samba.org/`.

[6] Note that if the books are divided into natural components such as chapters, sections, and subsections, then the alignment problem does not arise. This is similar to the setup described in Subsection 13.3.7 where both files consist of individual records with boundaries known to both parties.

not just with the "corresponding" blockwise checksums of the current file, but with the checksums of all possible positions of blocks in the current file. As a result, the server knows which parts of the current file already exist in the reference file and which new parts need to be communicated to the client. For efficiency reasons, two different checksums are communicated to the server, a fast but unreliable one, and a very reliable one that is more expensive to compute.

1. **At the client:**
   (a) Partition f_{old} into blocks $B_i = f_{old}[ib, (i+1)b - 1]$ of some size b to be determined later.
   (b) For each block B_i, compute two checksums, $u_i = h_u(B_i)$ and $r_i = h_r(B_i)$, and communicate them to the server. Here h_u is the unreliable but fast checksum function, and h_r is the reliable but expensive checksum function.
2. **At the server:**
   (a) For each pair of received checksums (u_i, r_i), insert an entry (u_i, r_i, i) into a dictionary data structure, using u_i as key value.
   (b) Perform a pass through f_{new}, starting at position $j = 0$ and involving the following steps:
      (i) Compute the unreliable checksum $h_u(f_{new}[j, j + b - 1])$ on the block starting at j.
      (ii) Check the dictionary for any block with a matching unreliable checksum.
      (iii) If found, and if the reliable checksums also match, transmit a pointer to the index of the matching block in f_{old} to the client, advance j by b positions, and continue.
      (iv) If none found, or if the reliable checksums did not match, transmit the symbol $f_{new}[j]$ to the client, advance j by one position, and continue.
3. **At the client:**
   (a) Use the incoming stream of data and pointers to blocks in f_{old} to reconstruct f_{new}.

Thus, the fast and unreliable checksum is used to find likely matches, and the reliable checksum is then used to verify the validity of the match.[7] The reliable checksum is implemented using MD4 (128 bits). The unreliable checksum is implemented as a 32-bit "rolling checksum" that allows efficient sliding of the block boundaries by one character; i.e., the checksum for $f[j+1, j+b]$ can be computed in constant time from $f[j, j+b-1]$.

Clearly, the choice of a good block size is critical to the performance of the algorithm. Unfortunately, the best choice is highly dependent on the degree of similarity between the two files—the more similar the files are, the larger the block size we can choose. Moreover, the location of changes in the file is also important. If a single character is changed in each block of f_{old}, then no match will be found by the server and *rsync* will be completely ineffective; on the other hand, if all changes are clustered in a few areas of the file, *rsync* will do very well even with a large block size. Given these observations, some basic performance bounds based on block size and number and size of file modifications can be shown. However, *rsync* does not have any good performance bounds with respect to common file distance metrics such as edit distance [37].

Of course, in general the optimal block size changes even within a file. In practice, *rsync* starts out with a block size of several hundred bytes and uses heuristics to adapt the block size later. Another optimization in *rsync* allows the server to compress all transmitted symbols for unmatched parts of the file using the LZ compression algorithm; this gives significant additional benefits in many situations as shown in the following.

[7] In addition, a checksum on the entire file is used to detect the (extremely unlikely) failure of the reliable checksum, in which case the entire procedure is repeated with a different choice of hash functions.

13.3.3 Some Experimental Results for *rsync*

We now provide some experimental results to give the reader an idea about the performance of *rsync* in comparison to delta compression techniques. The results use the *gcc* and *emacs* data sets from Subsection 13.2.4. We report five different numbers for *rsync*: the amount of data sent from client to server (request), the amount of data sent from server to client (reply), the amount sent from server to client with compression option switched on (reply compressed), and the total for both directions in uncompressed (total) and compressed (total compressed) form.

We observe that without compression option, *rsync* does worse than *gzip* on the *emacs* set (see Table 13.2). However, once we add compression for the reply message, *rsync* does significantly better than *gzip*, although it is still a factor of 3 to 4 from the best delta compressor. We also compared *rsync* on the artificial data sets from Subsection 13.2.4; due to the fine distribution of file changes for $q = 0.5$, *rsync* only achieves any benefits at all for p very close to 1. We note that *rsync* is typically applied in situations where the two files are very similar, and hence these numbers may look overly pessimistic. Clearly, *rsync* provides benefits to many people who use it on a daily basis.

In Table 13.3, we see how the performance of *rsync* varies as we decrease the block size used. The size of the server reply decreases as a smaller block size is used, since this allows a finer granularity of matches. Of course, the size of the request message increases, since more hash values need to be transmitted, and eventually this overcomes the savings for the reply (especially in the compressed case since the hash values in the request are incompressible).

In summary, there still exists a gap between delta compression and remote file synchronization techniques in terms of performance; we believe that this indicates room for significant

Table 13.2 Compression Results for *gcc* and *emacs* Data Sets (in kilobytes)

	gcc	emacs
Uncompressed	27,288	27,326
gzip	7,479	8,191
xdelta	461	2,131
vcdiff	289	1,821
zdelta	250	1,465
rsync request	180	227
rsync reply	2,445	12,528
rsync reply compressed	695	4,200
rsync total	2,626	12,756
rsync total compressed	876	4,428

Table 13.3 Compression Results for *emacs* with Different Block Sizes (in kilobytes)

	700	500	300	200	100	80
rsync request	227	301	472	686	1328	1649
rsync reply	12,528	11,673	10,504	9,603	8433	8161
rsync reply compressed	4,200	3,939	3,580	3,283	2842	2711
rsync total	12,756	11,974	10,976	10,290	9762	9810
rsync total compressed	4,428	4,241	4,053	3,970	4170	4360

improvements. One promising approach uses multiple roundtrips between client and sever, e.g., to determine the best block size or to recursively split blocks; see [16, 20, 37] for such methods. Recent experimental results by Orlitsky and Viswanathan [37] show improvements over *rsync* using such a multiround protocol.[8]

13.3.4 Theoretical Results

In addition to the heuristic solution given by *rsync*, a number of theoretical approaches have been studied that achieve provable bounds with respect to certain formal measures of file similarity. Many definitions of file similarity have been studied, e.g., *Hamming distance, edit distance*, or measures related to the compression performance of the Lempel–Ziv compression algorithm [16]. Results also depend on the number of messages exchanged between the two parties (e.g., in *rsync*, two messages are exchanged). In this subsection, we describe the basic ideas underlying these approaches and give an overview of known results, with emphasis on a few fundamental bounds presented by Orlitsky [35]. See [17, 28] and the references in [35] for some earlier results on this problem. The results in [35] are stated for a very general framework of pairs of random variables; in the following we give a slightly simplified presentation for the case of correlated (similar) files.

13.3.4.1 Distance Measures

We first discuss the issue of *distance measures*, which formalize the notion of file similarity. Note that Hamming distance, one of the most widely studied measures in coding theory, is not very appropriate in our scenario, since a single insertion of a character at the beginning of a file would result in a very large distance between the old and new files, while we expect a reasonable algorithm to be able to synchronize these two files with only a few bytes of communication. In the *edit distance* measure, we count the number of single-character change, insertion and deletion operations needed to transform one file into another, while more generalized notions of edit distance also allow for deletions and moves of blocks of data or may assign different weights to the operations. Finally, another family of distance measures is based on the number of operations needed to construct one file by copying blocks over from the other file and by inserting single characters; an example is the LZ measure proposed in [16]. As already discussed in Section 13.2, a realistic measure should allow for moves and copies of large blocks; however, from a theoretical point of view allowing such powerful operations makes things more complicated.

There are a few properties of distance functions that we need to discuss. A distance measure is called *symmetric* if the distance from f_0 to f_1 is the same as the distance from f_1 to f_0. This property is satisfied by Hamming and edit distance, but is not true for certain generalized forms of edit distance and copy-based measures. A distance measure d is a metric if (a) it is symmetric, (b) $d(x, y) \geq 0$ for all x, y, (c) $d(x, y) = 0$ iff $x = y$, and (d) it observes the Triangle Inequality.

13.3.4.2 Balanced Pairs

Now assume that as part of the input, we are given upper bounds $d_{new} \geq d(f_{old}, f_{new})$ and $d_{old} \geq d(f_{new}, f_{old})$ on the distances between the two files. We define $N_k(f)$, the k-neighborhood of a file f, as the set of all files f' such that $d(f', f) \leq k$. Thus, given the upper bounds on the distances, the server holding f_{new} knows that f_{old} is one of the files in $N_{d_{new}}(f_{new})$, while the client holding f_{old} knows that f_{new} is in $N_{d_{old}}(f_{old})$. We refer to the size of $N_{d_{new}}(f_{new})$ (resp. $N_{d_{old}}(f_{old})$) as the *ambiguity* of f_{new} (resp. f_{old}). We assume that both parties know *a priori* upper

[8] Note that these roundtrips are not incurred on a per-file basis, since we can handle many files at the same time. Thus, latency due to additional roundtrips is not a problem in many situations, although the amount of "state" that has to be maintained between roundtrips may be more of an issue.

bounds on both ambiguities, referred to as the *maximum ambiguity* of f_{new} (resp. f_{old}), written $mamb(f_{new})$ (resp. $mamb(f_{old})$). (In the presentation in [35], these bounds are implied by the given random distribution; in our context, we can assume that both parties compute estimates of their ambiguities beforehand based on file lengths and estimates of file distances.[9]

We say that a pair of files f_{new} and f_{old} is balanced if $mamb(f_{new}) = mamb(f_{old})$. Note that this property may depend on the choice of the two files as well as the distance measure (and possibly also the value of k). In the case of Hamming distance, all pairs of equal size are balanced, while for many edit and copy-based measures this is not true. In general, the more interesting measures are neither metrics nor generate balanced pairs. However, some results can be shown for distance measures that can be approximated by metrics or in cases where the ambiguities are not too different.

13.3.4.3 Results

We now describe a few fundamental results given by Orlitsky in [35, 36]. We assume that both parties *a priori* have some upper bounds on the distances between the two files. The goal is to limit the number of bits communicated in the worst case for a given number of roundtrips, with unbounded computational power available at the two parties (as commonly assumed in the formal study of communication complexity [27]). In the following, all logarithms are with base 2.

Theorem 13.1. *At least* $\lceil \log(mamb(f_{old})) \rceil$ *bits must be communicated from server to client by any protocol that works correctly on all pairs* f_{new} *and* f_{old} *with* $d(f_{new}, f_{old}) \leq d_{old}$*, independent of the number of messages exchanged and even if the server knows* f_{old} [36].

This first result follows directly from the fact that the client needs to be able to distinguish between all $mamb(f_{old})$ possible files f_{new} that satisfy $d(f_{new}, f_{old}) \leq d_{old}$. (Note that the result is independent of d_{new}.) Interestingly, if we send only a single message from server to client, we can match this result up to a factor of 2 in the case of a balanced pair, as shown in the following result.

Theorem 13.2. *There is a protocol that sends a single message of at most* $\log(mamb(f_{old})) + \log(mamb(f_{new})) + 1$ *bits from server to client and that works on all* f_{new} *and* f_{old} *with* $d(f_{new}, f_{old}) \leq d_{old}$ *and* $d(f_{old}, f_{new}) \leq d_{new}$ [35].

The result is obtained by considering the *characteristic hypergraph* for the problem, first defined by Witsenhausen [51] and obtained by adding a vertex for each file $f_{new} \in \Sigma^*$ and for each $f_{old} \in \Sigma^*$ a hyperedge $E(f_{old}) = \{f_{new} \mid d(f_{new}, f_{old}) \leq d_{old} \text{ and } d(f_{old}, f_{new}) \leq d_{new}\}$. Since each vertex is adjacent to at most $mamb(f_{new})$ edges and each edge contains at most $mamb(f_{old})$ vertices, the chromatic number of the hypergraph is at most $mamb(f_{old}) \cdot mamb(f_{new})$. If the server sends to the client the color of f_{new}, using $\lceil \log((mamb(f_{old}) \cdot mamb(f_{new})) \rceil \leq \log(mamb(f_{old})) + \log(mamb(f_{new})) + 1$ bits, then the client can reconstruct f_{new}.

As shown by Kahn and Orlitsky (see [35]), this result is almost tight for single-message protocols. However, if we allow more than a single message, much better results can be obtained, as briefly outlined in the following:

Theorem 13.3. *There exist protocols that work on all* f_{new} *and* f_{old} *with* $d(f_{new}, f_{old}) \leq d_{old}$ *and* $d(f_{old}, f_{new}) \leq d_{new}$ *and that achieve the following bounds* [35]*:*

(a) at most $2\log(mamb(f_{old})) + \log\log(max\{mamb(f_{new}), mamb(f_{old})\}) + 4$ *bits with two messages exchanged between client and server,*

[9] Note that explicitly transmitting the precise values of the ambiguities would require about the same amount of data transfer as the file reconciliation problem itself, as implied by the bounds below.

(b) at most $\log(mamb(f_{old})) + 3\log\log(max\{mamb(f_{new}), mamb(f_{old})\}) + 11$ *bits with three messages exchanged between client and server, and*

(c) at most $\log(mamb(f_{old})) + 4\log\log(mamb(f_{old}))$ *bits with four messages exchanged between client and server.*

The bound for two messages is based on a fairly simple and elegant construction of a family of perfect hash functions using the Lovasz Local Lemma [4]. The bound itself improves on the one-message bound above only for very unbalanced pairs, i.e., when $mamb(f_{old})$ is much smaller than $mamb(f_{new})$, but the result is the main building block for the three-message case. The three-message result is almost optimal for approximately balanced pairs, but not for very unbalanced pairs. This problem is resolved by the four-message protocol, which is in fact independent of $mamb(f_{new})$ and depends only on $mamb(f_{old})$. Thus, at most four messages suffice in principle to get a bit-optimal protocol, up to lower order terms.

While these results are very important in characterizing the fundamental complexity of the remote file synchronization problem, they suffer from three main limitations. First, the protocols do not seem to imply any efficiently implementable algorithms, since the client would have to check a large number of possible files f_{new} in order to find the one that has the color or hash value generated by the protocol. Second, many of the results rely on the existence of balanced pairs, and third, it is not clear what bounds are implied for interesting distance measures, which as discussed are rarely balanced or symmetric. This last issue is discussed in the next subsection.

13.3.5 Results for Particular Distance Measures

We now discuss bounds on communication that can be achieved for particular distance measures, focusing on results from [16, 35, 37].

In general, protocols for a distance measure typically consist of a first phase that uses a distance estimation technique to get an upper bound on the distances between the two files (discussed in the next subsection) and a second phase based, e.g., on one of the protocols of Orlitsky [35], that synchronizes the two files. In order to show bounds for the second phase relative to the file distance under a particular metric, we need to (i) investigate whether or not the files can be assumed to be balanced under the distance measure and select the appropiate protocol, and (ii) bound the maximum ambiguities based on the properties of the distance measures, the upper bounds on the distances, and the lengths of the files.

This approach is taken in [16, 35] to show bounds for a few distance measures. In particular, [16] derives the following bounds based on file lengths and distances:

- $O(\log(|f_{new}|) \cdot d(f_{new}, f_{old}))$ bits under the Hamming distance measure, with two roundtrips;
- $O(\log^3(|f_{new}|) \cdot d(f_{new}, f_{old}))$ bits under the edit distance measure and the LZ measure introduced in [16], with two roundtrips; and
- $O(\log(|f_{new}|) \cdot d(f_{new}, f_{old}))$ bits under the edit distance measure and the LZ measure, with $\log^2(|f_{new}|) + 1$ roundtrips.

In these results, all except the last roundtrip are used for distance estimation; the last bound is based on a more precise distance estimation. With the exception of the Hamming distance, where efficient coding techniques are known [1], these results are not practical due to the above-mentioned problem of efficiently decoding at the recipient.

Some more practical protocols are also given in [16, 37, 41]. The protocols in [16, 37] use a hierarchical partitioning approach, resulting in a logarithmic number of roundtrips but avoiding the decoding problems of the other approaches. Some experimental results in [37] show significant improvements for some data sets as compared to *rsync*, with a protocol that gives provable bounds for a variant of edit distance.

13.3.6 Estimating File Distances

As mentioned, many protocols require *a priori* knowledge of upper bounds on the distances between the two files. We discuss protocols for estimating these distances. We note that we could apply the sampling techniques in [10, 30] to construct fingerprints of the files that could be efficiently transmitted (see also [22] for the problem of finding similar files in larger collections). While these techniques may work well in practice to decide how similar two files are, they are not designed with any of the common distance measures in mind, but are based on the idea of estimating the number of common substrings of a given length.

Good distance estimation techniques for some specific metrics are described in [16]. These techniques consist of a single roundtrip in which a total of $O(\log n)$ bits are sent to exchange fingerprints obtained by appropriate sampling techniques. In particular, [16] shows that the LZ measure, which for the purpose of compression captures a lot of the notion of file similarity, as well as the edit distance, can be approximated through a slightly modified LZ-like measure that is in fact a metric and that this metric itself can be converted into Hamming distance.

13.3.7 Reconciling Database Records and File Systems

In the remote file synchronization problem, we assume that we have already identified two corresponding files, f_{new} and f_{old}, that need to be synchronized, and we would like to avoid transmitting an entire file to do so. However, in many scenarios we have a large number of items (files in a file system or records in a database), only a few of which have been changed. In this case, we would like to identify those items that have been changed without transmitting a separate fingerprint or time stamp for each one. Once we have identified these items, we can then synchronize them using the remote file synchronization techniques presented earlier or by transmitting the entire item in the case of small database records. In the following, we discuss the problems arising in this scenario, with emphasis on a recent approach described in [26, 33, 46]. Some earlier work on reconciliation of record-based data appeared in [1, 7, 31, 32].

Consider the case, assumed in [46], of a handheld device using the Palm Hotsync program to synchronize its database of addresses or appointments with a desktop device. If the handheld was last synchronized with the same desktop, then the Palm Hotsync software can use auxiliary logging information to efficiently identify items that need to be synchronized. However, in the general case where the two parties have not recently synchronized and where both may have added or deleted data, simple logging information will not help, and the software transmits all records.

Assume that we compute a fingerprint of small, fixed size (e.g., MD4 with 128 bits) for each object and that S_{new} is the set of fingerprints (integers) held by the server and S_{old} is the set of fingerprints held by the client.[10] Then the *set reconciliation problem* is the problem of determining the differences $S_{old} - S_{new}$ and $S_{new} - S_{old}$ of the two sets at the client. This then allows the client to decide which records it needs to request from and upload to the server. (Of course, in other cases, each party might want to obtain one of the differences.)

The reader might already observe that this scenario is now quite similar to that encountered in the context of error correcting codes (erasure codes) and this observation is used by Minsky *et al.* [33] to apply techniques from coding theory to the problem. In particular, one solution based on Reed–Solomon codes comes within a factor 2 of the information-theoretic lower bound, while a solution based on the interpolation of characteristic polynomials comes very close to optimal. The protocols in [33, 46] assume that an upper bound on the number of differences between the

[10] Note that if the records are very short, e.g., a few bytes, then we can directly perform reconciliation of the data without using fingerprints.

two sets is known, which can be obtained either by guessing as described in [46] or possibly by using known techniques for estimating set intersection sizes in [10, 30].

Experiments in [46] on a Palm OS handheld device demonstrate significant benefits over the Palm Hotsync approach in many cases. One critical issue is still the amount of computation required, which depends heavily on the number of differences between the two sets.

We note that we could also place S_{new} into a file f_{new}, and S_{old} into a file f_{old}, in sorted order and then apply *rsync* to these files (or even to a concatenation of the original files or records). However, this would not achieve the best possible bounds since set reconciliation is really an easier problem than file synchronization due to the assumption of sets with known record boundaries.

13.4 CONCLUSIONS AND OPEN PROBLEMS

In this chapter, we have described techniques, tools, and applications for delta compression and remote file synchronization problems. We believe that the importance of these problems will increase as computing becomes more and more network-centric, with millions of applications distributing, sharing, and modifying files on a global basis.

In the case of delta compression, existing tools already achieve fairly good compression ratios, and it will be difficult to significantly improve upon these results, although more modest improvements in terms of speed and compression are still possible. Recall that the best delta algorithms are currently based on Lempel–Ziv-type algorithms. In the context of (non-delta) compression, such algorithms, while not providing the absolute best compression for particular types of data, are still considered competitive for general-purpose compression. We would expect a similar situation for delta compression, with major improvements possible only for special types of data. On the the other hand, a lot of work still remains to be done on how to best use delta compression techniques, e.g., how to cluster files and identify good reference files, and what additional applications might benefit from delta compression techniques.

For remote file synchronization techniques, on the other hand, there still seems to be a significant gap in compression performance between the currently available tools and the theoretical limits. It is an important question whether we can modify the theoretical approaches from communication complexity into efficiently implementable algorithms with provably good performance. A more realistic goal would be to engineer the approach in *rsync* in order to narrow the gap in performance between remote file synchronization and delta compression.

ACKNOWLEDGMENTS

Thanks are extended to Dimitre Trendafilov for his work on implementing the *zdelta* compressor and to Dimitre Trendafilov and Patrick Noel for providing the experimental data. This work was supported by NSF CAREER Award NSF CCR-0093400 and by Intel Corp.

13.5 REFERENCES

1. Abdel-Ghaffar, K., and A. El Abbadi, 1994. An optimal strategy for comparing file copies. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, No. 1, pp. 87–93, January 1994.
2. Adler, M., and M. Mitzenmacher, 2001. Towards compressing web graphs. In *Proceedings of the IEEE Data Compression Conference (DCC)*, March 2001.
3. Ajtai, M., *et al.* 2000. Compactly Encoding Unstructured Inputs with Differential Compression. IBM Research Report RJ 10187, September 2000.

4. Alon, N., and J. Spencer, 1992. *The Probabilistic Method.* 1992, John Wiley & Sons, NY.
5. Balasubramaniam, S., and B. Pierce, 1998. What is a file synchronizer? In *Proceedings of the ACM/IEEE MOBICOM'98 Conference*, pp. 98–108, October 1998.
6. Banga, G., F. Douglis, and M. Rabinovich, 1997. Optimistic deltas for WWW latency reduction. In *1997 USENIX Annual Technical Conference, Anaheim, CA*, pp. 289–303, January 1997.
7. Barbara, D., and R. Lipton, 1991. A class of randomized strategies for low-cost comparison of file copies. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2, No. 2, pp. 160–170, April 1991.
8. Berliner, B., 1990. CVS II: Parallelizing software development. In *Proceedings of the Winter 1990 USENIX Conference*, pp. 341–352, January 1990.
9. Bharat, K., and A. Broder, 1999. Mirror, mirror on the web: A study of host pairs with replicated content. In *Proceedings of the 8th International World Wide Web Conference*, May 1999.
10. Broder, A., 1997. On the resemblance and containment of documents. In *Compression and Complexity of Sequences (SEQUENCES'97)*, pp. 21–29. IEEE Comput. Soc., Los Alamitos, CA.
11. Burns, R., and D. Long, 1997. Efficient distributed backup with delta compression. In *Proceedings of the Fifth Workshop on I/O in Parallel and Distributed Systems (IOPADS).*
12. Camerini, P., L. Fratta, and F. Maffioli, 1979. A note on finding optimum branchings. *Networks*, Vol. 9, pp. 309–312.
13. Chan, M., and T. Woo, 1999. Cache-based compaction: A new technique for optimizing web transfer. In *Proceedings of INFOCOM'99*, March 1999.
14. Chen, Y., F. Douglis, H. Huang, and K. Vo, 2000. Topblend: An efficient implementation of HtmlDiff in Java. In *Proceedings of the WebNet2000 Conference*, October 2000.
15. Cho, J., N. Shivakumar, and H. Garcia-Molina, 2000. Finding replicated web collections. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 355–366, May 2000.
16. Cormode, G., M. Paterson, S. Sahinalp, and U. Vishkin, 2000. Communication complexity of document exchange. In *Proceedings of the ACM–SIAM Symposium on Discrete Algorithms*, January 2000.
17. Costa, M., 1983. Writing on dirty paper. *IEEE Transactions on Information Theory*, pp. 439–441, May 1983.
18. Delco, M., and M. Ionescu, 2000. xProxy: A transparent caching and delta transfer system for web objects. Unpublished manuscript. May 2000.
19. Douglis, F., A. Haro, and M. Rabinovich, 1997. HPP: HTML macro-preprocessing to support dynamic document caching. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, December 1997.
20. Evfimievski, A., 1998. A probabilistic algorithm for updating files over a communication link. In *Procedings of the Ninth Annual ACM–SIAM Symposium on Discrete Algorithms,* pp. 300–305, January 1998.
21. Gailly, J., zlib compression library. Available at: http://www.gzip.org/zlib/.
22. Haveliwala, T. H., A. Gionis, and P. Indyk, 2000. Scalable techniques for clustering the web. In *Proceedings of the WebDB Workshop*, Dallas, TX, May 2000.
23. Hirai, J., S. Raghavan, H. Garcia-Molina, and A. Paepcke, 2000. WebBase: A repository of web pages. In *Proceedings of the 9th International World Wide Web Conference*, May 2000.
24. Housel, B., and D. Lindquist, 1996. WebExpress: A system for optimizaing web browsing in a wireless environment. In *Proceedings of the 2nd ACM Conference on Mobile Computing and Networking*, pp. 108–116, November 1996.
25. Hunt, J., K. P. Vo, and W. Tichy, 1998. Delta algorithms: An empirical analysis. *ACM Transactions on Software Engineering and Methodology*, Vol. 7.
26. Karpovsky, M., L. Levitin, and A. Trachtenberg, 2001. Data verification and reconciliation with generalized error-control codes. In *39th Annual Allerton Conference on Communication, Control, and Computing.*
27. Kushilevitz, E., and N. Nisan, 1997. *Communication Complexity.* Cambridge Univ. Press, Cambridge, UK.
28. Levenshtein, V. I., 1966. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, Vol. 10, No. 8, pp. 707–710. February 1966.
29. MacDonald, J., 2000. *File System Support for Delta Compression*, M.S. thesis, University of California at Berkeley, May 2000.

30. Manber, U., and S. Wu, 1994. GLIMPSE: A tool to search through entire file systems. In *Proceedings of the 1994 Winter USENIX Conference*, pp. 23–32, January 1994.
31. Metzner, J., 1983. A parity structure for large remotely located replicated data files. *IEEE Transactions on Computers*, Vol. 32, No. 8, pp. 727–730, August 1983.
32. Metzner, J., 1991. Efficient replicated remote file comparison. *IEEE Transactions on Computers*, Vol. 40, No. 5, pp. 651–659, May 1991.
33. Minsky, Y., A. Trachtenberg, and R. Zippel, 2000. Set Reconciliation with Almost Optimal Communication Complexity, Technical Report TR2000-1813, Cornell University, Ithaca, NY.
34. Mogul, J. C., F. Douglis, A. Feldmann, and B. Krishnamurthy, 1997. Potential benefits of delta-encoding and data compression for HTTP. In *Proceedings of the ACM SIGCOMM Conference*, pp. 181–196.
35. Orlitsky, A., 1993. Interactive communication of balanced distributions and of correlated files. *SIAM Journal of Discrete Math*, Vol. 6, No. 4, pp. 548–564.
36. Orlitsky, A., 1991. Worst-case interactive communication. II. Two messages are not optimal. *IEEE Transactions on Information Theory*, Vol. 37, No. 4, pp. 995–1005, July 1991.
37. Orlitsky, A. and K. Viswanathan, 2001. Practical algorithms for interactive communication. In *IEEE Int. Symp. on Information Theory*, June 2001.
38. Ouyang, Z., N. Memon, T. Suel, and D. Trendafilor, 2002. Cluster-based delta compression for collections of web pages. In *Proceedings of the 3rd International Conference on Web Information Systems Engineering (WISE)*, December 2002.
39. Pyne, C., 1995. Remote File Transfer Method and Apparatus. U.S. Patent 5446888.
40. Rochkind, M., 1975. The source code control system. *IEEE Transactions on Software Engineering*, Vol. 1, pp. 364–370, December 1975.
41. Schwarz, T., R. Bowdidge, and W. Burkhard, 1990. Low cost comparison of file copies. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pp. 196–202.
42. Tarjan, R., 1977. Finding optimum branchings. *Networks*, Vol. 7, pp. 25–35.
43. Tate, S., 1997. Band ordering in lossless compression of multispectral images. *IEEE Transactions on Computers*, Vol. 46, No. 45, pp. 211–320.
44. Tichy, W., 1984. The string-to-string correction problem with block moves. *ACM Transactions on Computer Systems*, Vol. 2, No. 4, pp. 309–321, November 1984.
45. Tichy, W., 1985. RCS: A system for version control. *Software—Practice and Experience*, Vol. 15, pp. 637–654. July 1985.
46. Trachtenberg, A., D. Starobinski, and S. Agarwal, 2001. Fast PDA Synchronization Using Characteristic Polynomial Interpolation, Technical Report BU-2001-03, Department of Electrical and Computer Engineering. Boston University.
47. Trendafilov, D., N. Memon, and T. Suel, 2002. zdelta: A simple delta compression tool. Technical Report, CIS Department, Polytechnic University, Brooklyn, NY, June 2002.
48. Tridgell, A., 2000. *Efficient Algorithms for Sorting and Synchronization*, Ph.D. thesis, Australian National University, Canberra, Australia, April 2000.
49. Tridgell, A., and P. MacKerras, 1996. The rsync Algorithm, Technical Report TR-CS-96-05, Australian National University, Canberra, Australia, June 1996.
50. Wagner, R. A., and M. J. Fisher, 1973. The string-to-string correction problem. *Journal of the ACM*, Vol. 21, No. 1, pp. 168–173, January 1973.
51. Witsenhausen, H., 1976. The zero-error side information problem and chromatic numbers. *IEEE Transactions on Information Theory*, Vol. 22, No. 5, pp. 592–593, September 1976.
52. Ziv, J., and A. Lempel, 1977. A universal algorithm for data compression. *IEEE Transactions on Information Theory*, Vol. 23, No. 3, pp. 337–343.
53. Ziv, J., and A. Lempel, 1978. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, Vol. 24, No. 5, pp. 530–536.