Polytechnic
UNIVERSITY
Brooklyn · Long Island · Westchester

Cluster-Based Delta Compression of a Collection of Files

Zan Ouyang   Nasir Memon   Torsten Suel   Dimitre Trendafilov

Department of Computer and Information Science

Technical Report
TR-CIS-2002-05
12/27/2002

Cluster-Based Delta Compression of a Collection of Files [*]

Zan Ouyang *Nasir Memon* *Torsten Suel* *Dimitre Trendafilov*

CIS Department
Polytechnic University
Brooklyn, NY 11201

Abstract

Delta compression techniques are commonly used to succinctly represent an updated version of a file with respect to an earlier one. In this paper, we study the use of delta compression in a somewhat different scenario, where we wish to compress a large collection of (more or less) related files by performing a sequence of pairwise delta compressions. The problem of finding an optimal delta encoding for a collection of files by taking pairwise deltas can be reduced to the problem of computing a branching of maximum weight in a weighted directed graph, but this solution is inefficient and thus does not scale to larger file collections. This motivates us to propose a framework for cluster-based delta compression that uses text clustering techniques to prune the graph of possible pairwise delta encodings. To demonstrate the efficacy of our approach, we present experimental results on collections of web pages. Our experiments show that cluster-based delta compression of collections provides significant improvements in compression ratio as compared to individually compressing each file or using `tar+gzip`, at a moderate cost in efficiency.

A shorter version of this paper appears in the Proceedings of the 3rd International Conference on Web Information Systems Engineering (WISE), December 2002.

---

[*] This project was supported by a grant from Intel Corporation, and by the Wireless Internet Center for Advanced Technology (WICAT) at Polytechnic University. Torsten Suel was also supported by NSF CAREER Award NSF CCR-0093400.

1 Introduction

Delta compressors are software tools for compactly encoding the differences between two files or strings in order to reduce communication or storage costs. Examples of such tools are the `diff` and `bdiff` utilities for computing edit sequences between two files, and the more recent `xdelta` [16], `vdelta` [12], `vcdiff` [15], and `zdelta` [26] tools that compute highly compressed representations of file differences. These tools have a number of applications in various networking and storage scenarios; see [21] for a more detailed discussion. In a communication scenario, they typically exploit the fact that the sender and receiver both possess a reference file that is similar to the transmitted file; thus transmitting only the difference (or delta) between the two files requires a significantly smaller number of bits. In storage applications such as version control systems, deltas are often orders of magnitude smaller than the compressed target file.

Delta compression techniques have also been studied in detail in the context of the World Wide Web, where consecutive versions of a web page often differ only slightly [8, 19] and pages on the same site share a lot of common HTML structure [5]. In particular, work in [2, 5, 7, 11, 18] considers possible improvements to HTTP caching based on sending a delta with respect to a previous version of the page, or another similar page, that is already located in a client or proxy cache.

In this paper, we study the use of delta compression in a slightly different scenario. While in most other applications, delta compression is performed with respect to a previous version of the same file, or some other easy to identify reference file, we are interested in using delta compression to better compress large collections of files where it is not obvious at all how to efficiently identify appropriate reference and target files. Our approach is based on a reduction to the optimum branching problem in graph theory and the use of recently proposed clustering techniques for finding similar files.

We focus on collections of web pages from several sites. Applications that we have in mind are efficient downloading and storage of collection of web pages for off-line browsing, and improved archiving of massive terabyte web collections such as the Internet Archive (see `http://archive.org`). However, the techniques we study are applicable to other scenarios as well, and might lead to new general-purpose tools for exchanging collections of files that improve over the currently used `zip` and `tar/gzip` tools.

1.1 Contributions of this Paper

In this paper, we study the problem of compressing collections of files, with focus on collections of web pages, with varying degrees of similarity among the files. Our approach is based on using an efficient delta compressor, in particular the `zdelta` compressor [26], to achieve significantly better compression than that obtained by compressing each file individually or by using tools such as `tar` and `gzip` on the collection. Our main contributions are:

- The problem of obtaining optimal compression of a collection of n files, given a specific delta compressor, can be solved by finding an optimal branching on a directed graph with n nodes and n^2 edges. We implement this algorithm and show that it can achieve significantly better compression than current tools. On the other hand, the algorithm quickly becomes inefficient as the collection size grows beyond a few hundred files, due to its quadratic complexity.

- We present a general framework, called *cluster-based delta compression*, for efficiently computing near-optimal delta encoding schemes on large collections of files. The framework combines the branching approach with two recently proposed hash-based techniques for clustering files by similarity [3, 10, 14, 17].

- Within this framework, we evaluate a number of different algorithms and heuristics in terms of compression and running time. Our results show that compression very close to that achieved by the optimal branching algorithm can be achieved in time that is within a small multiplicative factor of the time needed by tools such as `gzip`.

We also note three limitations of our study: First, our results are still preliminary and we expect additional improvements in running time and compression over the results in this paper. In particular, we believe we can narrow the gap between the speed of `gzip` and our best algorithms. Secondly, we restrict ourselves to the case where each target file is compressed with respect to a single reference file. Additional significant improvements in compression might be achievable by using more than one reference file, at the cost of additional algorithmic complexity. Finally, we only consider the problem of compressing and uncompressing an entire collection, and do not allow individual files to be added to or retrieved from the collection.

The rest of this paper is organized as follows. The next subsection lists related work. In Section 2 we discuss the problem of compressing a collection of files using delta compression, and describe an optimal algorithm based on computing a maximum weight branching in a directed graph. Section 3 provides our framework called *cluster-based delta compression* and outlines several approaches under this framework. In Section 4, we present our experimental results. Finally, Section 5 provides some open questions and concluding remarks.

1.2 Related Work

For an overview of delta compression techniques and applications, see [21]. Delta compression techniques were originally introduced in the context of version control systems; see [12, 25] for a discussion. Among the main delta compression algorithms in use today are `diff` and `vdelta` [12]. Using `diff` to find the difference between two files and then applying `gzip` to compress the difference is a simple and widely used way to perform delta compression, but it does not provide good compression on files that are only slightly similar. `vdelta`, on the other hand, is a relatively new technique that integrates both data compression and data differencing. It is a refinement of Tichy's block-move algorithm [24] that generalizes the well known Lempel-Ziv technique [27] to delta compression. In our work, we use the `zdelta` compressor, which was shown to achieve good compression and running time in [26].

The issue of appropriate distance measures between files and strings has been studied extensively, and many different measures have been proposed. We note that `diff` is related to the symmetric *edit distance* measure, while `vdelta` and other recent Lempel-Ziv type delta compressors such as `xdelta` [16], `vcdiff` [15], and `zdelta` [26] are related to the *copy distance* between two files. Recent work in [6] studies a measure called *LZ distance* that is closely related to the performance of Lempel-Ziv type compressing schemes. We also refer to [6] and the references therein for work on protocols for estimating file similarities over a communication link.

Fast algorithms for the optimum branching problem are described in [4, 22]. While we are not aware of previous work that uses optimum branchings to compress collections of files, there are two previous applications that are quite similar. In particular, Tate [23] uses optimum branchings to find an optimal scheme for compressing multispectral images, while Adler and Mitzenmacher [1] use it to compress the graph structure of the World Wide Web. Adler and Mitzenmacher [1] also show that a natural extension of the branching problem to hypergraphs that can be used to model delta compression with two or more reference files is NP Complete, indicating that an efficient optimal solution is unlikely.

We use two types of hash-based clustering techniques in our work, a technique with quadratic complexity called *min-wise independent hashing* proposed by Broder in [3] (see also Manber and Wu [17] for a similar technique), and a very recent nearly linear time technique called *locality-sensitive hashing* proposed by Indyk and Motwani in [14] and applied to web documents in [10].

Finally, Chan and Woo [5] observe that in the case of web pages, similarities in the URL provide a powerful heuristic for identifying good reference files for delta compression. Thus, another web page from a close-by subdirectory on the same web server often shares a lot of content and structure with the given page; in principle this should also provide a useful heuristic for other file collections in a hierarchical file system. In this paper, we do not follow this approach, as it is highly dependent on the nature of the collection. However, in practice it might be useful to combine the two approaches in some way.

2 Delta Compression Based on Optimum Branchings

Delta compressors such as `vcdiff` or `zdelta` provide an efficient way to encode the difference between two similar files. However, given a collection of files, we are faced with the problem of succinctly representing the entire collection through appropriate delta encodings between target and reference files. We observe that the problem of finding an optimal encoding scheme for a collection of files through pairwise deltas can be reduced to that of computing an optimum branching B of an appropriately constructed weighted directed graph G.

2.1 Problem Reduction

Formally, a branching B of a directed graph G is defined as a set of edges such that (1) B contains at most one incoming edge for each node, and (2) B does not contain a cycle. Given a weighted directed graph, a maximum branching is a branching of maximum edge weight. Given a collection of n files we construct a complete directed graph $G = (V, E)$ where each node corresponds to a file and each directed edge (i, j) has a corresponding weight $w_{i,j}$ that represents the reduction (in bytes) obtained by delta-compressing file j with respect to file i. In addition to these n nodes, the graph G includes an extra *null node* corresponding to the empty file that is used to model the compression savings if a file is compressed by itself (using, e.g., `zlib`, or `zdelta` with an empty reference file).

Given the above formulation it is not difficult to see that a maximum branching of the graph G gives us an optimal delta encoding scheme for a collection of files. Condition (1) in the definition of a branching expresses the constraint that each file is compressed with respect to only one other file. The second condition ensures that there are no cyclical dependencies that would prevent us from decompressing the collection. Finally, given the manner in which the weights have been assigned, a maximum branching results in a compression scheme with optimal benefit over the uncompressed case.

Figure 1 shows the weighted directed graph formed by a collection of four files. In the example, node 0 is the null node, while nodes 1, 2, 3, and 4 represent the four files. The weights on the edges from node 0 to nodes 1, 2, 3, and 4 are the compression savings obtained when the target files are

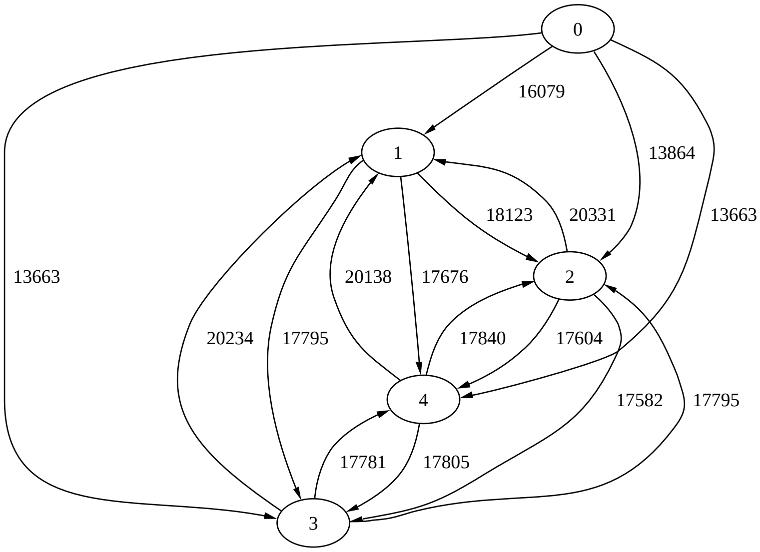

Figure 1: Example of a directed and weighted complete graph. The optimal branching for the graph consists of the edges $(0,1)$, $(1,2)$, $(1,3)$, and $(3,4)$

data set	pages	average size	total size	cat+gzip ratio	cat+gzip time	opt branch ratio	opt branch time
CBC	530	19.5 KB	10.6 MB	5.83	3.8 s	10.01	4133 s
CBSNews	218	37.0 KB	8.3 MB	5.06	3.2 s	15.42	993 s
USAToday	344	47.5 KB	16.7 MB	6.30	5.98 s	18.64	2218 s
CSmonitor	388	41.7 KB	16.6 MB	5.06	6.7 s	17.31	4373 s
Ebay	100	21.5 KB	2.2 MB	6.78	0.8 s	10.90	168 s
Thomas-dist	105	26.5 KB	2.8 MB	6.39	1.0 s	9.73	361 s
all sites	1685	33.9 KB	57.2 MB	5.53	18.5 s	12.36	-

Table 1: Compression ratios for some collections of files.

compressed by themselves. The weights for all other edges (i,j) represent compression savings when file j is compressed using i as a reference file. The optimal sequence for compression is $(0,1)$, $(1,2)$, $(1,3)$, and $(3,4)$, i.e., file 1 is compressed by itself, files 2 and 3 are compressed by computing a delta with respect to file 1, and file 4 is compressed by computing a delta with respect to file 3.

2.2 Experimental Results

We implemented delta compression based on the optimal branching algorithm described in [4, 22], which for dense graphs takes time proportional to the number of edges. Table 1 shows compression results and times on several collections of web pages that we collected by crawling a limited number of pages from each site using a breadth-first crawler.

The results indicate that the optimum branching approach can give significant improvements in compression over using `cat` or `tar` followed by `gzip`, outperforming them by a factor of 2 to 3. However, the major problem with the optimum branching approach is that it becomes very inefficient as soon as the number of files grows beyond a few dozens. Thus, for the `cbc.ca` data set with 530 pages, it took more than an hour ($4133s$) to perform the computation, while multiple hours were needed for the set with all sites.

Figure 2 plots the running time in seconds of the optimal branching algorithm for different numbers of files, using a set of files from the `gcc` software distribution also used in [12, 26]. Time is plotted on a logarithmic scale to accomodate two curves: the time spent on computing the edge weights (upper curve), and the time spent on the actual branching computation after the weights of the graph have been determined using calls to `zdelta` (lower curve). While both curves grow quadratically, the vast majority of the time is spent on computing appropriate edge weights for the graph G, and only a tiny amount is spent on the actual branching computation afterwards.

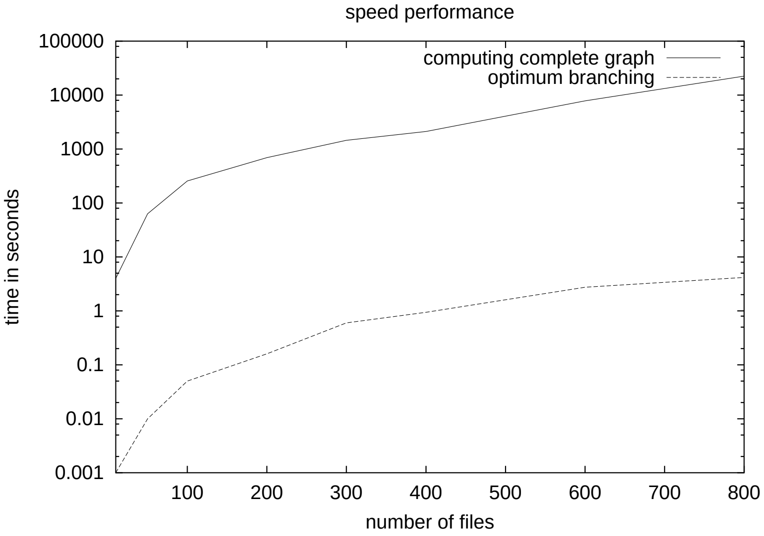

Figure 2: Running time of the optimal branching algorithm

Thus, in order to compress larger collections of pages, we need to find techniques that avoid computing the exact weights of all edges in the complete graph G. In the next sections, we study such techniques based on clustering of pages and pruning and approximation of edges. We note that another limitation of the branching approach is that it does not support the efficient retrieval of individual files from a compressed collection, or the addition of new files to the collection. This is a problem in some applications that require interactive access, and we do not address it in this paper.

3 Cluster-Based Delta Compression

As shown in the previous section, delta compression techniques have the potential for significantly improved compression of collections of files. However, the optimal algorithm based on maximum branching quickly becomes a bottleneck as we increase the collection size n, mainly due to the quadratic number of pairwise delta compression computations that have to be performed. In this section, we describe a basic framework, called *Cluster-Based Delta Compression*, for efficiently computing near-optimal delta compression schemes on larger collections of files.

3.1 Basic Framework

We first describe the general approach, which leads to several different algorithms that we implemented. In a nutshell, the basic idea is to prune the complete graph G into a sparse subgraph G', and then find the best delta encoding scheme within this subgraph. More precisely, we have the following general steps:

(1) **Collection Analysis:** Perform a clustering computation that identifies pairs of files that are very similar and thus good candidates for delta compression. Build a sparse directed subgraph G' containing only edges between these similar pairs.

(2) **Assigning Weights:** Compute or estimate appropriate edge weights for G'.

(3) **Maximum Branching:** Perform a maximum branching computation on G' to determine a good delta encoding.

The assignment of weights in the second step can be done either precisely, by performing a delta compression across each remaining edge, or approximately, e.g., by using estimates for file similarity produced during the document analysis in the first step. Note that if the weights are computed precisely by a delta compressor and the resulting compressed files are saved, then the actual delta compression after the last step consists of simply removing files corresponding to unused edges (assuming sufficient disk space).

The primary challenge is Step (1), where we need to efficiently identify a small subset of file pairs that give good delta compression. We will solve this problem by using two sets of known techniques for document clustering, one set proposed by Broder [3] (see also [17] for a similar idea), and one set proposed by Indyk and Motwani [14] and applied to document clustering by Haveliwala, Gionis, and Indyk [10]. These techniques were developed in the context of identifying near-duplicate web pages and finding closely related pages on the web. While these problems are clearly closely related to our scenario, there are also a number of differences that make it nontrivial to apply the techniques to delta compression, and in the following we discuss these issues.

3.2 File Similarity Measures

The compression performance of a delta compressor on a pair of files depends on many details, such as the precise locations and lengths of the matches, the internal compressibility of the target file, the windowing mechanism, and the performance of the internal Huffman coder. A number of formal measures of file similarity, such as edit distance (with or without block moves), copy distance, or LZ distance [6] have been proposed that provide reasonable approximations; see [6, 21] for a discussion. However, even these simplified measures are not easy to compute with, and thus the clustering techniques in [3, 17, 10] that we use are based on two even simpler similarity measures, which we refer to as *shingle intersection* and *shingle containment*.

Formally, for a file f and an integer q, we define the *shingle set* (or q-gram set) $S(f)$ of f as the multiset of substrings of length q (called shingles) that occur in f. Given two files f and f', we define the shingle intersection of f and f' as $I(f, f') = \frac{|S(f) \cap S(f')|}{|S(f) \cup S(f')|}$. We define the *shingle* *containment* of f with respect to f' as $C(f, f') = \frac{|S(f) \cap S(f')|}{|S(f)|}$. (Note that shingle containment is not symmetric.)

Thus, both of these measures assign higher similarity scores to files that share a lot of short substrings, and intuitively we should expect a correlation between the delta compressibility of two files and these similarity measures. In fact, the following relationship between shingle intersection and the edit distance measure can be easily derived:

- Given two files f and f' within edit distance d, and a shingle size q, we have $|S(f) \cap S(f')| \geq \max(|f|, |f'|) + q - 1 - d \cdot q$.

We refer to [9] for a proof and a similar result for the case of edit distance with block moves. A similar relationship can also be derived between shingle containment and copy distances. Thus, shingle intersection and shingle containment are related to the edit distance and copy distance measures, which have been used as models for the corresponding classes of edit-based and copy-based delta compression schemes.

While the above discussion supports the use of the shingle-based similarity measures in our scenario, in practice the relationship between these measures and the achieved delta compression ratio is quite noisy. Moreover, for efficiency reasons we will only approximate these measures, introducing additional potential for error.

3.3 Clustering Using Min-Wise Independent Hashing

We now describe the first set of techniques, called *min-wise independent hashing*, that was proposed by Broder in [3]. (A similar technique is described by Manber and Wu in [17].) The simple idea in this technique is to approximate the shingle similarity measures by sampling a small subset of shingles from each file. However, in order to obtain a good estimate, the samples are not drawn independently from each file, but they are obtained in a coordinated fashion using a common set of random hash functions that map shingles of length q to integer values. We then select in each file the smallest hash values obtained this way.

We refer the reader to [3] for a detailed analysis. Note that there are a number of different choices that can be made in implementing these schemes:

- **Choice of hash functions:** We used a class of simple linear hash functions analyzed by Indyk in [13] and also used in [10].

- **Sample Sizes:** One option is to use a fixed number of samples, say 100 or 1000, from each file, independent of file size. Alternatively, we could sample at a constant rate, say $1/64$ or $1/128$, resulting in sample sizes that are proportional to file sizes.

- **One or several hash functions:** One way to select s samples from a file is to use s hash functions, and include the minimum value under each hash function in the sample. Alternatively, we could select one random hash function, and select the s smallest values under this hash function. We selected the second method as it is significantly more efficient, requiring only one hash function computation for each shingle.

- **Shingle size:** We used a shingle size of $q = 4$ bytes in the results reported here. (We also experimented with $q = 8$ but achieved slightly worse results.)

After selecting the sample, we estimate the shingle intersection or shingle containment measures by intersecting the samples of every pair of files. Thus, this phase takes time quadratic in the number of files. Finally, we decide which edges to include in the sparse graph G'. There are two independent choices to be made here:

- **Similarity measure:** We can use either intersection or containment as our measure.
- **Threshold versus k best neighbors:** We could keep all edges above a certain similarity threshold, say 50%, in the graph. Or, for each file, we could keep the k most promising incoming edges, for some constant k, i.e., the edges coming from the k nearest neighbors w.r.t. the estimated similarity measure.

A detailed discussion of the various implementation choices outlined here and their impact on running time and compression is given in the experimental section.

The total running time for the clustering step using min-wise independent hashing is thus roughly[1] $O(nm + n^2 \cdot s)$ where n is the number of files, m the (average) size of each file, and s the (average) size of each sample. The main advantage over the optimal algorithm is that for each edge, instead of performing a delta compression step between two files of size m (several kilobyte), we perform a simpler computation between two samples of some small size s (say, $s = 50$). This results in a significant speedup over the optimal algorithm in practice, although the algorithm will eventually become inefficient due to the quadratic complexity.

3.4 Clustering Using Locality-Sensitive Hashing

The second set of techniques, proposed by Indyk and Motwani [14] and applied to document clustering by Haveliwala, Gionis, and Indyk [10], is an extension of the first set that results in an almost linear running time. In particular, these techniques avoid the pairwise comparison between all n files by performing a number of sorting steps on specially designed hash signatures that can directly identify similar files.

The first step of the technique is identical to that of the min-wise independent hashing technique for fixed sample size. That is, we select from each file a fixed number of min-wise independent hash values, using s different random hash functions. For a file f, let $h_i(f)$ be the value selected by the ith hash function. The main idea, called *locality-sensitive hashing*, is to now use these hash values to construct *file signatures* that consist of the concatenation of w hash values (e.g., for $w = 4$ we concatenate four 32-bit hash values into one 128-bit signature). If two files agree on their signature, then this is strong evidence that their intersection is above some threshold. It can be formally shown that by repeating this process a number of times that depends on w and the chosen threshold, we will find most pairs of files with shingle intersection above the threshold, while avoiding most of the pairs below the threshold. For a more formal description of this technique we refer to [10].

The resulting algorithm consists of the following steps:

(1) *Sampling:* Extract a fixed number s of hash values $h_i(f)$ from each file f in the collection, using s different hash functions.

(2) *Locality-sensitive hashing:* Repeat the following l times:

---

[1] If s different hash functions are used, then an additional factor of s has to be added to the first term.

(a) Randomly select w indexes i_0 to i_{w-1} from $\{0, \ldots, s-1\}$.

(b) For each file f construct a signature by concatenating hash values $h_{i_0}(f)$ to $h_{i_{w-1}}(f)$.

(c) Sort all resulting signatures, and scan the sorted list to find all pairs of files whose signature is identical.

(d) For each such pair, add edges in both directions to G'.

Thus, the running time of this method is given by $O(smn + lwn \cdot \lg(wn))$, where s, l, and w are constants in the range from 2 to at most 100 depending on the choice of parameters. We discuss parameter settings and their consequences in detail in the experimental section.

We note two limitations. First, the above implementation only identifies the pairs that are above a given fixed similarity threshold. Thus, it does not allow us to determine the k best neighbors for each node, and it does not provide a good estimate of the precise similarity of a pair (i.e., whether it is significantly or only slightly above the threshold). Second, the method is based on shingle intersection, and not shingle containment. Addressing these limitations is an issue for future work.

4 Experimental Evaluation

In this section, we perform an experimental evaluation of several cluster-based compression schemes that we implemented based on the framework from the previous section. We first introduce the algorithm and the experimental setup. In Subsection 4.2 we show that naive methods based on thresholds to do not give good results. The next three subsections look at different techniques that resolve this problem, and finally Subsection 4.7 presents results for our best two algorithms on a larger data set. Due to space constraints and the large number of options, we can only give a selection of our results. We refer the reader to [20] for a more complete evaluation.

4.1 Algorithms

We implemented a number of different algorithms and variants. In particular, we have the following options:

- Basic scheme: MH vs. LSH.
- Number of hash function: single hash vs. multiple hash.
- Sample size: fixed size vs. fixed rate.
- Similarity measure: intersection vs. containment.
- Edge pruning rule: threshold vs. best neighbors vs. heuristics.
- Edge weight: exact vs. estimated.

We note that not every combination of these choices make sense. For example, our LSH implementations do not support containment or best neighbors, and require a fixed sample size. On the other hand, we did not observe any benefit in using multiple hash functions in the MH scheme, and thus assume a single hash function for this case. We note that in our implementations, all samples were treated as sets, rather than multi-sets, so a frequently occurring string is presented at most once.[2]

---

[2] Intuitively, this seems appropriate given our goal of modeling delta compression performance.

algorithm	sample size	threshold	remaining edges	branching size (edges)	benefit over zlib	total size
zlib					0	10.92 MB
tar+gzip					1893402	9.03 MB
optimal			2,782,224	1667	6980935	3.94 MB
MH intersect	100	20%	357,961	1616	6953569	3.97 MB
		40%	154,533	1434	6601218	4.32 MB
		60%	43,289	988	5326760	5.60 MB
		80%	2,629	265	1372123	9.55 MB
MH intersect	$\frac{1}{128}$	20%	391,682	1641	6961645	3.96 MB
		40%	165,563	1481	6665907	4.25 MB
		60%	42,474	1060	5450312	5.46 MB
		80%	4,022	368	1621910	9.30 MB
MH contain	$\frac{1}{128}$	20%	1,258,272	1658	6977748	3.94 MB
		40%	463,213	1638	6943999	3.98 MB
		60%	225,675	1550	6724167	4.17 MB
		80%	79,404	1074	5016699	5.91 MB

Table 2: Number of remaining edges, number of edges in the final branching, and compression benefit for threshold-based clustering schemes for different sampling techniques and threshold values.

All algorithms were implemented in C and compiled using `gcc 2.95.2` under Solaris 7. Experiments were run on a E450 Sun Enterprise server, with two UltraSparc IIe CPUs at 400MHz and 4 GB of RAM. Only one CPU was used in the experiments, and data was read from a single $10,000$ rpm SCSI disk. We note that the large amount of memory and fast disk minimize the impact of I/O on the running times. We used two data sets:

- The *medium data set* consists of the union of the six web page collections from Section 2, with 1668 files and a total size of $48.7MB$.
- The *large data set* consists of 20180 HTML pages crawled from the `poly.edu` domain, with a total size of $257.8MB$. The pages were crawled in a breadth-first crawl that attempted to fetch all pages reachable from the `www.poly.edu` homepage, subject to certain pruning rules to avoid dynamically generated content and cgi scripts.

4.2 Threshold-Based Methods

The first experiments that we present look at the performance of MH and LSH techniques that try to identify and retain all edges that are above a certain similarity threshold.

In Table 2 we look at the optimum branching method and at three different algorithms that use a fixed threshold to select edges that are considered similar, for different thresholds. For each method, we show the number of similar edges, the number of edges included in the final branching, and the total improvement obtained by the method as compared to compressing each file individually using `zlib`. The results demonstrate a fundamental problem that arises in these threshold-based methods: for high thresholds, the vast majority of edges is eliminated, but the resulting branching is of poor quality compared to the optimal one. For low thresholds, we obtain compression close to the optimal, but the number of similar edges is very high; this is a problem since the number of edges included in G' determines the cost of the subsequent computation.[3] Unfortunately, these numbers indicate that there is no real "sweet spot" for the threshold that gives both a small number of remaining edges and good compression on this data set.

We note that this result is not due to the precision of the sampling-based methods, and it also holds for threshold-based LSH algorithms. A simplified explanation for this is that data sets contain different clusters of various similarity, and a low threshold will keep these clusters intact as dense graphs with many edges, while a high threshold will disconnect too many of these clusters, resulting in inferior compression. This leads us to study several techniques for overcoming this problem:

- **Best neighbors:** By retaining only the best k incoming edges for each node according to the MH algorithm, we can keep the number of edges in G' bounded by kn.
- **Estimating weights:** Another way to improve the efficiency of threshold-based MH algorithms is to directly use the similarity estimate provided by the MH schemes as the edge weight in the subsequent branching.
- **Pruning heuristics:** We have also experimented with heuristics for decreasing the number of edges in LSH algorithms, described further below.

In summary, using a fixed threshold followed by an optimal branching on the remaining edges does not result in a very good trade-off between compression and running time.

4.3 Distance Measures

Next, we investigate the assumption underlying our algorithms that the shingle intersection and shingle inclusion measures correlate with delta-compressibility.

Figure 3 plots this correlation as follows: on each x-axis, we have the 2782224 edges in the complete directed graph on the 1668 files of the data set, sorted from left to right by intersection and containment measure, respectively, and split into groups of 1000 files. (Thus, the leftmost coordinate 0 on the x-axis corresponds to the 1000 directed edges in the graph with smallest intersection or containment similarity measure.) For each group of files, we plot on the y-axis the minimum, maximum, and average multiplicative benefit beyond `zlib` compression provided by `zdelta` (i.e., 2 means that `zdelta` compressed twice as well as `zlib` in terms of the compression ratio). These three measures are clearly visible in the plots as three different bands of points. We note from the maximum and minimum values at the top and bottom that the relationship is quite noisy. Considering the middle band showing the averages, we see that containment provides a somewhat better model for delta compressibility. This is not surprising given the relationship between copy distance and containment and the fact that `zdelta`, as most modern delta compressors, is copy-based. For this reason, we decided to limit our study of MH algorithms to the containment measure, which uses a fixed sampling rate.

---

[3] For example, to compute the exact weight of each edge above a 20% threshold (in the case of fixed sample size) we have to perform $357,961$ calls to `zdelta` at a cost of about $10ms$ each.

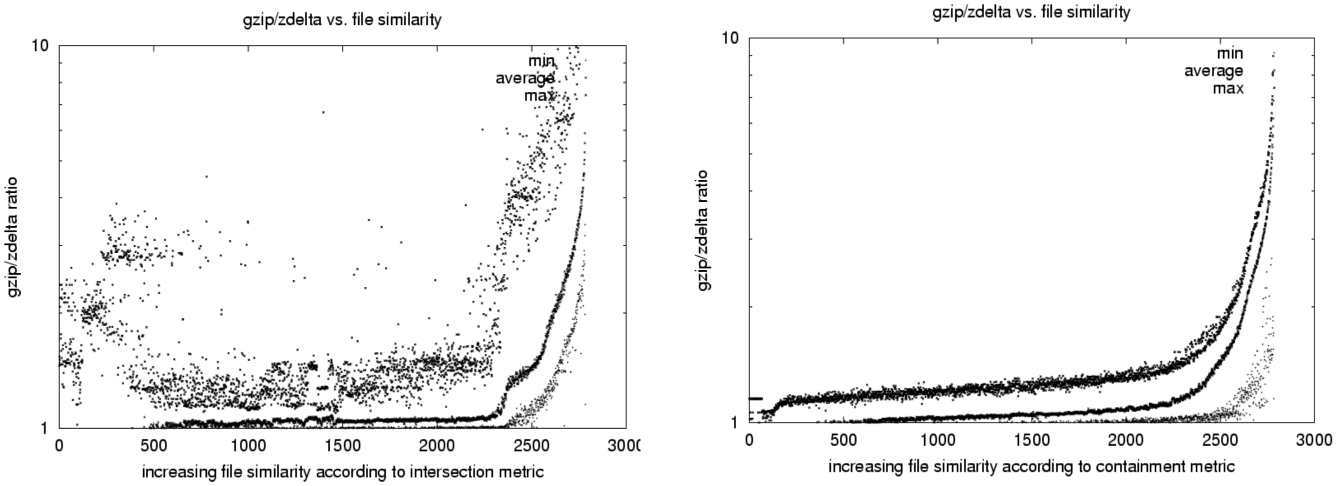

Figure 3: Relationship between similarity measures and delta compression benefits.

We note however one potential drawback of the containment measure. Consider a large collection of small files that all have a significant containment w.r.t. a single very large file. In this case, we may decide to compress each of the small files using the large file as a reference file, resulting in many scans over the large file, even though there might be other smaller reference files that would achieve almost the same compression ratio.[4] In general, the intersection metric always assigns a low similarity value to pairs of very different size, while the containment metric often assigns a high similarity value to pairs where the reference file is quite large, potentially resulting in slow compression and decompression time. In our experiments, however, running time was not significantly impacted.

4.4 Using Best Neighbors

We now look at the case where we limit the number of remaining edges in the MH algorithm by keeping only the k most similar edges into each node, as proposed above. Clearly, this limits the total number of edges in G' to kn, thus reducing the cost of the subsequent computations.

Table 3 shows the running times of the various phases and the compression benefit as a function of the number of neighbors k and the sampling rate. The clustering time of course depends heavily on the sampling rate; thus one should use the smallest sampling rate that gives reasonable compression, and we do not observe any significant impact on compression up to a rate of $1/128$ for the file sizes we have. The time for computing the weights of the graph G' grows (approximately) linear with k. The compression rate grows with k, but even for very small k, such as $k = 2$, we get results that are within 5% of the maximum benefit. As in all our results, the time for the actual branching computation on G' is negligible.

---

[4] In fact, using the large file could potentially result in worse compression since the pointer and window movement mechanisms in the delta compressor may not be able to fully capture all the common substrings that are distributed over the large file.

sample size	k	cluster time	weighing time	branching time	benefit over zlib
1/2	1	1198.25	51.44	0.02	6137816
	2	1201.27	84.17	0.02	6693921
	4	1198.00	149.99	0.04	6879510
	8	1198.91	287.31	0.09	6937119
1/128	1	40.52	47.77	0.02	6124913
	2	40.65	82.88	0.03	6604095
	4	40.57	149.06	0.03	6774854
	8	40.82	283.57	0.09	6883487

Table 3: Running time and compression benefit for k-neighbor schemes.

k	cluster time	branching time	zdelta time	benefit over zlib
1	39.26	0.02	45.63	6115888
2	39.35	0.02	48.49	6408702
4	39.35	0.02	48.14	6464221
8	39.40	0.06	49.63	6503158

Table 4: Running time and compression benefit for k-neighbor schemes with sampling rate $1/128$ and estimated edge weights.

4.5 Estimated Weights

By using the containment measure values computed by the MH clustering as the weights of the remaining edges in G', we can further decrease the running time, as shown in Table 4. The time for building the weighted graph is now essentially reduced to zero. However, we have an extra step at the end where we perform the actual compression across edges, which is independent of k and has the same cost as computing the exact weights for $k = 1$. Looking at the achieved benefit we see that for $k = 8$ we are within about 7% of the optimum, at a total cost of less than 90 seconds (versus about 16 seconds for standard `zlib` and several hours for the optimum branching).

4.6 LSH Pruning Heuristic

For LSH algorithms, we experimented with a simple heuristic for reducing the number of remaining edges where, after the sorting of the file signatures, we only keep a subset of the edges in the case where more than 2 files have identical signatures. In particular, instead of building a complete graph on these files, we connect these files by a simple linear chain of directed edges. This somewhat arbitrary heuristic (which actually started out as a bug) results in significantly decreased running time at only a slight cost in compression, as shown in Table 5. We are currently looking at other more principled approaches to thinning out tightly connected clusters of edges.

threshold	edges	branching size	benefit over zlib
20%	28,866	1640	6689872
40%	8,421	1612	6242688
60%	6,316	1538	5426000
80%	2,527	1483	4945364

Table 5: Number of remaining edges and compression benefit for LSH scheme with pruning heuristic.

algorithm	running time	size
uncompressed		257.8 MB
zlib	73.9	42.3 MB
cat+gzip	79.5	30.5 MB
fast MH	996.3	23.7 MB
best MH	2311.1	19.2 MB
fast LSH	800.0	21.7 MB
best LSH	2175.1	19.5 MB

Table 6: Comparison of best MH and LSH schemes to `zlib` and `cat+gzip`.

4.7 Best Results for Large Data Set

Finally, we present the results of the best schemes identified above on the large data set of 20180 pages from the `poly.edu` domain. We note that our results are still very preliminary and can probably be significantly improved by some optimizations. We were unable to compute the optimum branching on this set due to its size, though we expect the result to be slightly lower than the $19.2MB$ achieved by the best method.

The MH algorithms used $k = 8$ neighbors and estimated and exact edge weights, respectively. The LSH algorithms used thresholds of 50% and 25%, respectively, and the pruning heuristic from the previous subsection. For fast MH, about 75% of the running time is spent on the clustering, which scales as $\Theta(n^2)$ and thus eventually becomes a bottleneck, and 25% on the final compression step. For the others, more than 75% is spent computing the exact weights of remaining edges. There are several optimizations that we are currently implementing and that should result in significant reductions in running time. In particular, we can combine the two methods by first running LSH with a low threshold, and then using MH on the remaining edges to choose the k neighbors and estimate their weights. We are also working on optimizations in the sampling and hashing phase of the algorithms.

5 Concluding Remarks

In this paper, we have investigated the problem of using delta compression to obtain a compact representation of a cluster of files. As described, the problem of optimally encoding a collection using delta compression based on a single file can be reduced to the problem of computing a maximum weight branching. However, while providing superior compression, this algorithm does not scale to larger collections, motivating us to propose a faster cluster-based delta compression framework. We studied several file clustering heuristics and performed extensive experimental comparisons. Our preliminary results show that significant compression improvements can be obtained over `tar+gzip` at moderate additional computational costs.

Many open questions remain. First, some additional optimizations are possible that should lead to improvements in compression and running time, including faster sampling and better pruning heuristics for LSH methods. Second, the cluster-based framework we have proposed uses only pairwise deltas, that is, each file is compressed with respect to only a single reference file. It has been shown [5] that multiple reference files can result in significant improvements in compression, and in fact this is already partially exploited by `tar+gzip` with its $32\ KB$ window on small files. As discussed, a polynomial-time optimal solution for multiple reference files is unlikely, and even finding schemes that work well in practice is challenging. Our final goal is to create general purpose tools for distributing file collections that improve significantly over `tar+gzip`.

In related work, we are also studying how to apply delta compression techniques to a large web repository[5] that can store billions of pages on a network of workstations. Note that in this scenario, fast insertions and lookups are crucial, and significant changes in the approach are necessary. An early prototype of the system is currently being evaluated.

References

[1] M. Adler and M. Mitzenmacher. Towards compressing web graphs. In *Proc. of the IEEE Data Compression Conference (DCC)*, March 2001.

[2] G. Banga, F. Douglis, and M. Rabinovich. Optimistic deltas for WWW latency reduction. In *1997 USENIX Annual Technical Conference, Anaheim, CA*, pages 289–303, January 1997.

[3] A. Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences (SEQUENCES'97)*, pages 21–29. IEEE Computer Society, 1997.

[4] P. Camerini, L. Fratta, and F. Maffioli. A note on finding optimum branchings. *Networks*, 9:309–312, 1979.

[5] M. Chan and T. Woo. Cache-based compaction: A new technique for optimizing web transfer. In *Proc. of INFOCOM'99*, March 1999.

[6] G. Cormode, M. Paterson, S. Sahinalp, and U. Vishkin. Communication complexity of document exchange. In *Proc. of the ACM–SIAM Symp. on Discrete Algorithms*, January 2000.

[7] M. Delco and M. Ionescu. xProxy: A transparent caching and delta transfer system for web objects. May 2000. unpublished manuscript.

[8] F. Douglis, A. Feldmann, B. Krishnamurthy, and J. Mogul. Rate of change and other metrics: a live study of the World Wide Web. In *Proc. of the USENIX Symp. on Internet Technologies and Systems (ITS-97)*, pages 147–158, Berkeley, December 8–11 1997. USENIX Association.

---

[5] Similar to the Internet Archive at `http://www.archive.org`.

[9] L. Gravano, P. Ipeirotis, H. Jagadish, N. Koudas, S. Muthukrishnan, L. Pietarinen, and D. Srivastava. Using q-grams in a DBMS for approximate string processing. *IEEE Data Engineering Bulletin*, 24(4):28–34, December 2001.

[10] T.H. Haveliwala, A. Gionis, and P. Indyk. Scalable techniques for clustering the web. In *Proc. of the WebDB Workshop*, Dallas, TX, May 2000.

[11] B. Housel and D. Lindquist. WebExpress: A system for optimizaing web browsing in a wireless environment. In *Proc. of the 2nd ACM Conf. on Mobile Computing and Networking*, pages 108–116, November 1996.

[12] J. Hunt, K. P. Vo, and W. Tichy. Delta algorithms: An empirical analysis. *ACM Transactions on Software Engineering and Methodology*, 7, 1998.

[13] P. Indyk. A small approximately min-wise independent family of hash functions. In *Proc. of the 10th Symp. on Discrete Algorithms*, January 1999.

[14] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proc. of the 30th ACM Symp. on Theory of Computing*, pages 604–612, May 1998.

[15] D. Korn and K.-P. Vo. Engineering a differencing and compression data format. In *Proceedings of the Usenix Annual Technical Conference*, pages 219–228, June 2002.

[16] J. MacDonald. File system support for delta compression. MS Thesis, UC Berkeley, May 2000.

[17] U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. In *Proc. of the 1994 Winter USENIX Conference*, pages 23–32, January 1994.

[18] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta-encoding and data compression for HTTP. In *Proc. of the ACM SIGCOMM Conference*, pages 181–196, 1997.

[19] Z. Ouyang, N. Memon, and T. Suel. Delta encoding of related web pages. In *Proc. of the IEEE Data Compression Conference (DCC)*, March 2001.

[20] Z. Ouyang, N. Memon, T. Suel, and D. Trendafilov. Cluster-based delta compression of a collection of files. Technical Report TR-CIS-2002-05, Polytechnic University, CIS Department, October 2002.

[21] T. Suel and N. Memon. Algorithms for delta compression and remote file synchronization. In Khalid Sayood, editor, *Lossless Compression Handbook*. Academic Press, 2002. to appear.

[22] R. Tarjan. Finding optimum branchings. *Networks*, 7:25–35, 1977.

[23] S. Tate. Band ordering in lossless compression of multispectral images. *IEEE Transactions on Computers*, 46(45):211–320, 1997.

[24] W. Tichy. The string-to-string correction problem with block moves. *ACM Transactions on Computer Systems*, 2(4):309–321, November 1984.

[25] W. Tichy. RCS: A system for version control. *Software - Practice and Experience*, 15, July 1985.

[26] D. Trendafilov, N. Memon, and T. Suel. zdelta: a simple delta compression tool. Technical Report TR-CIS-2002-02, Polytechnic University, CIS Department, June 2002.

[27] J. Ziv and A. Lempel. A universal algorithm for data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.