In-Place Reconstruction of Delta Compressed Files

Randal C. Burns
IBM Almaden Research Center
650 Harry Rd., San Jose, CA 95120
randal@almaden.ibm.com

Darrell D. E. Long[†]
Department of Computer Science
University of California, Santa Cruz, CA 95064
darrell@cs.ucsc.edu

Abstract

We present an algorithm for modifying delta compressed files so that the compressed versions may be reconstructed without scratch space. This allows network clients with limited resources to efficiently update software by retrieving delta compressed versions over a network.

Delta compression for binary files, compactly encoding a version of data with only the changed bytes from a previous version, may be used to efficiently distribute software over low bandwidth channels, such as the Internet. Traditional methods for rebuilding these delta files require memory or storage space on the target machine for both the old and new version of the file to be reconstructed. With the advent of network computing and Internet-enabled devices, many of these network attached target machines have limited additional scratch space. We present an algorithm for modifying a delta compressed version file so that it may rebuild the new file version in the space that the current version occupies.

1 Introduction

Recent developments in portable computing and computing appliances have resulted in a proliferation of small network attached computing devices. These include personal digital assistants (PDAs), Internet set-top boxes, network computers, control devices with analog sensors, and cellular devices. The software and operating systems of these devices may be updated by transmitting the new version of a program over a network. However, low bandwidth channels to network devices often makes the time to perform software update prohibitive. In particular, heavy Internet traffic results in high latency and low bandwidth to web-enabled clients and prevents the timely delivery of software.

Differential or delta compression [5, 1], compactly encoding a new version of a file using only the changed bytes from a previous version, can be used to reduce the size of the file to be transmitted and consequently the time to perform software update. Currently, decompressing delta encoded files requires scratch space, additional disk or memory storage, used to hold a required second copy of the file. Two copies of the compressed file must be concurrently available, as the delta file contains directives to read data from the old file version while the new file version is being materialized in another region of storage. This presents a problem. Network attached devices often have limited memory resources and no disks and therefore are not capable of storing two file versions at the same time. Furthermore, adding storage to network attached devices is not viable, as keeping these devices simple limits their production costs.

We address this problem by post-processing delta encoded files so that they are suitable for reconstructing the new version of the file *in-place*, materializing the new version in the same memory or storage space that the previous version occupies. A delta file can be considered a set of instructions to a computer to materialize a new file version in the presence of a *reference* version, the old version of the file. When rebuilding a version encoded by a delta file, data are both copied from the reference file to the new version and added explicitly when portions of the new version do not appear in the reference version.

If we attempt to reconstruct an arbitrary delta file in-place, the resulting output can often be corrupt. This occurs when the delta encoding instructs the computer to copy data from a file region where new file data has already been written. The data the command attempts to read have already been altered and the rebuilt file is not correct. By detecting and avoiding such conflicts, our method allows us to rebuild versions with no scratch space.

We present a graph-theoretic algorithm for post-processing delta files that detects situations where a delta file would attempt to read from an already written region and permutes the order that the commands in a delta file are applied to reduce the occurrence of these conflicts. The algorithm elim-

[†]The work of this author was performed while a Visiting Scientist at the IBM Almaden Research Center.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PODC 98 Puerto Vallarta Mexico
Copyright ACM 1998 0-89791-977-7/98/ 6...$5.00

inates the remaining data conflicts by removing commands that copy data and explicitly adding these data to the delta file. Eliminating data copied between versions increases the size of the delta encoding but allows the algorithm to always output an in-place reconstructible delta file.

Experimental results verify that modifying delta files for in-place reconstruction is a viable and efficient technology. Our findings indicate that a small fraction of compression is lost in exchange for in-place reconstructibility. Also, in-place reconstructible files can be generated efficiently; creating a delta file takes less time than modifying it to be in-place reconstructible.

In §2, we summarize the preceding work in the field of delta compression. We describe how delta files are encoded in §3. In §4, we present an algorithm that modifies delta encoded files to be in-place reconstructible. In §5, we further examine the exchange of run-time and compression performance. In §6, we present limits on the size of the digraphs our algorithm generates. Section 7 presents experimental results for the execution time and compression performance of our algorithm and we presents our conclusions in §8.

2 Related Work

Encoding versions of data compactly by detecting altered regions of data is a well known problem. The first applications of delta compression found changed lines in text data for analyzing the recent modifications to files [6]. Considering data as lines of text fails to encode minimum sized delta files, as it does not examine data at a minimum *granularity* and only finds matching data that are *aligned* at the beginning of a new line.

The problem of compactly representing the changes between version of data was formalized as string-to-string correction with block move [14] – detecting maximally matching regions of a file at an arbitrarily fine granularity without alignment. Even though the general problem of detecting and encoding version to version modifications was well defined, delta compression applications continued to rely on the alignment of data, as in database records [13], and the grouping of data into block or line granularity, as in source code control systems [12, 15], to simplify the combinatorial task of finding the common and different strings between files.

Efforts to generalize delta compression to data that are not aligned and to minimize the granularity of the smallest change resulted in algorithms for compressing data at the granularity of a byte. Early algorithms were based upon either dynamic programming [9] or the greedy method [11] and performed this task using time quadratic in the length of the input files. Recent advances in differencing algorithms have produced efficient algorithms that detect matching strings between versions at an arbitrarily fine granularity without alignment restrictions [1, 5]. These differencing algorithms trade an experimentally verified small amount of compression in order to run using time linear in the length of the input files. The improved algorithms allow large files without known structure to be efficiently differenced and permits the application of delta compression to backup and restore [4], file system replication, and software distribution.

Recently, applications distributing HTTP objects using delta files have emerged [10, 2]. This permits web servers to both reduce the amount of data to be transmitted to a client and reduce the latency associated with loading web pages. Efforts to standardize delta files as part of the HTTP protocol and the trend towards making small network devices, for example hand-held organizers, HTTP compliant indicate the need to efficiently distribute data to network devices.

3 Encoding Delta Files

Differencing algorithms compactly encode the changes between file versions by finding strings in the new file that may be copied from the prior version of the same file. Differencing algorithms perform this task by partitioning the data in the file into strings that may be encoded using copies and strings that do not appear in the prior version and must be explicitly added to the new file. Having partitioned the file to be compressed, the algorithm outputs a delta file that encodes this version compactly. This delta file consists of an ordered sequence of ***copy*** commands and ***add*** commands. An add command is an ordered pair, $\langle t, l\rangle$, where t (to) encodes the string offset in the file version and l (length) encodes the length of the string. This pair is followed by the l bytes of data to be added (Figure 1).

The encoding of a copy command is an ordered triple, $\langle f, t, l\rangle$ where f (from) encodes the offset in the reference file from which data are copied, t encodes the offset in the new file where the data are to be written, and l encodes that length of the data to be copied. The copy command is a directive that copies the string data in the interval $[f, f+l-1]$ in the reference file to the interval $[t, t+l-1]$ in the version file.

In the presence of the reference file, a delta file rebuilds the version file with add and copy directives. The intervals in the version file encoded by these directives are disjoint. Therefore, any permutation of the order these commands are applied to the reference file materializes the same output version file.

4 An In-Place Reconstruction Algorithm

This algorithm modifies an existing delta file so that it can correctly reconstruct a new file version in the space the current version occupies. At a high level, our technique examines the input delta file to find copy commands that read from the write interval of other copy commands. These po-

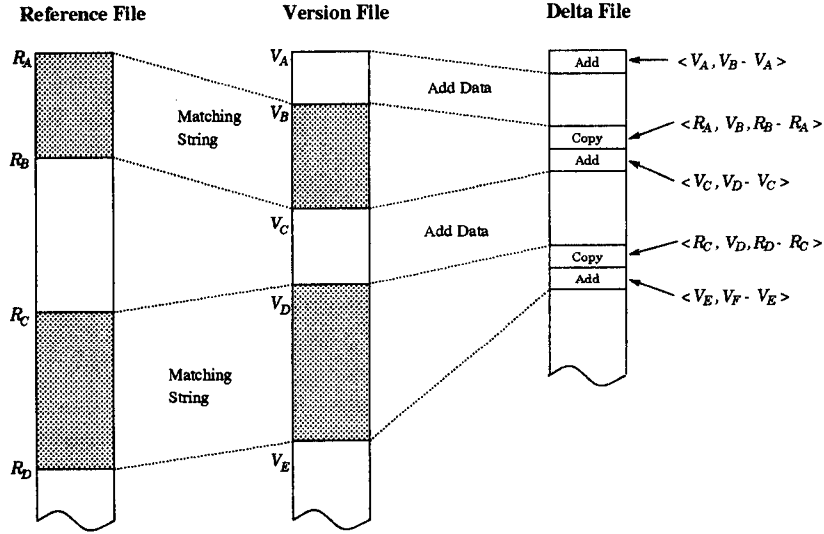

Figure 1: Encoding delta files. Common strings are encoded as copy commands $\langle f, t, l \rangle$ and new strings in the new file are encoded as add commands $\langle t, l \rangle$ followed by the string of length l that the command adds.

tential data conflicts are encoded into a digraph. This digraph is then topologically sorted to produce an ordering on these copy commands that reduces data conflicts. We eliminate the remaining conflicts by converting copy commands to add commands and output the permuted and converted commands as an in-place reconstructible delta file.

While our algorithm can most easily be described as a post-processing step on an existing delta file, as done in this work, it also integrates easily into a compression algorithm so that an in-place reconstructible file may be output directly.

4.1 Conflict Detection

Since we are reconstructing files in-place, we need concern ourselves with the order that the copy commands are applied. Assume that we attempt to read from an offset that has already been written. This will result in an incorrect reconstruction since the reference file data we are attempting to read have been overwritten. This is termed a *write before read* (*WR*) conflict [3].

For copy commands $\langle f_i, t_i, l_i \rangle$ and $\langle f_j, t_j, l_j \rangle$, with $i < j$, there is a *WR* conflict when

$$[t_i, t_i + l_i - 1] \cap [f_j, f_j + l_j - 1] \neq \emptyset. \quad (1)$$

In other words, copy command i and copy command j conflict if i writes to the interval from which copy command j reads data. By convention, we consider copy commands that are numbered and ordered so that, if $i < j$, copy command i is read and written before copy command j. By enforcing this constraint on copy commands i and j, we limit ourselves to applying the delta file commands serially, which is appropriate for limited capability network devices.

This definition only considers *WR* conflicts between copy commands and neglects add commands. Add commands only write data to the new file; they do not read data from the reference file. Consequently, all potential *WR* conflicts associated with adding data may be avoided by placing add commands at the end of a delta file. This way, all reads associated with copy commands are completed before the first add command is processed.

Additionally, we define *WR* conflicts so that a copy command cannot conflict with itself. Yet, a single copy command's read and write intervals may intersect and would seem to cause a conflict. Copy commands whose read and write intervals are overlapping can be dealt with by performing the copy in either a *left-to-right* or *right-to-left* manner. For copy command $\langle f, t, l \rangle$, if $f \geq t$, then when the file is being reconstructed, copy the string byte by byte starting at the left-hand side. Since, the f (from) offset in the reference file is always greater than the t (to) offset in the new file, a *left-to-right* copy never reads a byte that has been overwritten by a previous byte in the string. When $f < t$, a symmetric argument shows that we should start our copy at the right hand edge of the string and work backwards. For this example, we performed the copies in a byte-wise fashion. However, the notion of a left-to-right or right-to-left copy applies to moving a read/write buffer of any size.

To avoid *WR* conflicts and achieve the in-place reconstruction of delta files, we employ the following three techniques.

1. Place all add commands at the end of the delta file to avoid data conflicts with copy commands.

2. Permute the order of application of the copy commands

to reduce the number of write before read conflicts.

3. For remaining *WR* conflicts, remove the conflicting operation by converting a copy command to an add command and place it at the end of the delta file.

Any delta file may be post-processed using these methods to create a delta file that can be reconstructed in-place. For many delta files, a permutation that eliminates all *WR* conflicts is unattainable. Consequently, we require the conversion of copy commands to add commands to create correct in-place reconstructible files for all inputs.

Having post-processed a delta file for in-place reconstruction, the permuted and modified delta file obeys the property

$$(\forall j)\left[[f_j, f_j + l_j - 1] \cap \left(\bigcup_{i=1}^{j-1}[t_i, t_i + l - 1]\right) = \emptyset\right], \quad (2)$$

indicating the absence of any *WR* conflict. Equivalently, it guarantees that the data a copy command reads and transfers are always data from the original file.

4.2 Generating Conflict Free Permutations

In order to find a permutation that minimizes *WR* conflicts, we encode potential conflicts between the copy commands in a digraph and topologically sort this digraph. A topological sort on digraph $G = \langle V, E\rangle$ produces a linear order on all vertices so that if G contains edge $\vec{uv}$ then vertex u precedes vertex v in topological order.

Our technique constructs a digraph so that each copy command in the delta file has a corresponding vertex in the digraph. On this set of vertices, we construct an edge relation with a directed edge $\vec{uv}$ from node u to node v when copy command u's read interval intersects copy command v's write interval. Edge $\vec{uv}$ indicates that by performing command u before command v, the delta file avoids a WR conflict. For digraphs with this edge relation, a topologically sorted version of a digraph adheres to the requirement for in-place reconstruction (Equation 2).

As a total topological ordering is only possible on acyclic digraphs and the digraphs we construct on delta files may contain cycles, we enhance a standard topological sort to break cycles and output a total topological order on a subgraph. Common implementations of topological sort can detect cycles [8]. Upon detecting a cycle, our modified topological sort breaks the cycle by removing a vertex. When completing this enhanced sort, the result consists of a digraph containing a subset of all vertices in topological order and a set of vertices that were removed. This algorithm re-encodes the data contained in the copy commands of the removed vertices as add commands in the output in-place reconstructible delta file.

As these converted add commands are followed by the string that contains the encoded data, this replacement reduces compression in the delta file. An in-place conversion algorithm can minimize the amount of compression lost by selecting the optimal set of copy commands for converting the input digraph into a acyclic digraph. However, this natural optimization problem is NP-hard. For our implementation of in-place conversion, we examine two policies for breaking cycles. The *constant time* policy picks the easiest vertex to remove, based on the execution order of the topological sort, and deletes this node. This policy performs no extra work when breaking cycles. The *locally minimum* policy detects a cycle and loops through all nodes in the cycle to determine and then delete the minimum cost node, *i.e.* the node that encodes the smallest copied string. The locally minimum policy may perform as much additional work as the total length of cycles found by the algorithm. We further examine the issues of efficiently breaking cycles in §5.

Our algorithm for converting delta files into in-place reconstructible delta files takes the following steps to find and eliminate *WR* conflicts between a reference file and the new version to be materialized.

Algorithm

1. Given an input delta file, the first step is to partition the commands in the file into a set C of copy commands and a set A of add commands.

2. Sort the copy commands by increasing write offset, $C_{\text{sorted}} = \{c_1, c_2, ..., c_n\}$. For c_i and c_j, this set obeys: $i < j \longleftrightarrow t_i < t_j$.

3. Construct a digraph from the copy commands. For the copy commands $c_1, c_2, ..., c_n$, we create a vertex set $V = \{v_1, v_2, ..., v_n\}$. The edge set E is built by adding an edge from node v_i to node v_j when copy command c_i reads from the interval to which c_j writes:

$$\vec{v_i v_j} \longleftrightarrow [f_i, f_i + l_i - 1] \cap [t_j, t_j + l_j - 1] \neq \emptyset.$$

4. Perform a topological sort on the nodes of the digraph. This sort also detects cycles in the digraph and breaks them. When breaking a cycle, one node is selected, using either the locally minimum or constant time cycle breaking policy, and removed. The data encoded in its copy command are replaced by an equivalent command and put into set A. The output of the topological sort is an ordering of the remaining copy commands that obeys the property in Equation 2.

5. Traverse the sorted digraph, writing to the delta file, in topological order, the copy command encoded by each node.

6. Output all add commands in the set A to the delta file.

The resulting delta file reconstructs the new version *out of order*, both out of write order in the version file and out of

the order that the commands appeared in the original delta file.

4.3 Algorithmic Performance

Given the set of copy commands C with $|C|$ members, the presented algorithm uses time $O(|C| \log |C|)$ for both sorting the copy commands by write order and for finding conflicting commands, using binary search on the sorted write intervals for the $|V|$ nodes in V – recall that $|V| = |C|$. Additionally, the algorithm separates and outputs add commands using time $O(|A|)$ and builds the edge relation using time $O(|E|)$. The total execution time is $O(|C| \log |C| + |E| + |A|)$. This result relies upon a topological sort algorithm that runs in time $O(|V| + |E|)$ [8]. Additionally, we have assumed that cycles found in the digraph will be broken with the constant time policy ($O(1)$). The algorithm uses space $O(|E| + |C| + |A|)$ to store the data associated with each set.

The algorithm accepts as input a delta file of length n. It generates as many nodes as copy commands and the number of copy commands can grow linearly in the size of the input delta file, $|V| = |C| = O(n)$. The same is true of add commands, $|A| = O(n)$. However, we have no bound for the number of edges, excepting the theoretical bound on general digraphs $O(|V|^2)$. In §6, we demonstrate by example that the subclass of digraphs generated by our algorithm can realize this bound. On the other hand, we also show that the number of edges in the class of digraphs our algorithm generates is linear in the length of the version file V that the delta file encodes (Lemma 1). We denote the length of V by L_V.

Substituting these bounds into performance expression, for an input delta file on length n encoding a version file of length L_V, our algorithm runs in time $O(n \log n + L_V)$ and space $O(n + L_V)$.

5 Strategies for Breaking Cycles

The topological sort algorithm described above generates an ordering on the nodes of a digraph and detects cycles that are in conflict with the ordering. Having found the cycles in a digraph, we are faced with the problem of choosing a method to break all of these cycles. We previously mentioned a *constant time* policy that breaks individual cycles using time $O(1)$. This policy can easily be implemented in a topological sort by selecting the last node in sort order before the cycle was found. However, this method does not guarantee to break cycles with a minimum compression cost. Yet, other techniques for breaking cycles turn out to be either intractable or offer no compression advantage in the worst case. In fact, determining the minimum cost, in terms of lost compression, set of nodes in a digraph to be removed turns out to be NP-hard.

We note that the digraphs the algorithm generates are a subclass of general digraphs. We denote this subclass *conflicting read write interval (CRWI)* digraphs. To the best of our knowledge, this class has not previously been studied. While we know little about the its structure, it is clear that the class of CRWI digraphs is smaller than that of general digraphs. For example, the CRWI class does not include any complete digraphs with more than two vertices.

We define the amount of compression lost upon deleting a node the *cost* of deletion. Based on this cost function, we formulate the optimization problem of finding the minimum cost set of nodes to delete to make a digraph acyclic. A copy command is an ordered triple of constant size, $c = \langle f, t, l \rangle$. An add command is an ordered double $a = \langle t, l \rangle$ followed by the l bytes of data to be added to the new version of the file. Replacing a copy command with an add commands increases the delta file size by exactly $l - |f|$, where $|f|$ is the size of the encoding of f.

Given cost function labeling each node v_i with cost $l_i - |f_i|$, we can express the elimination of cycles in an optimization problem:

> **Input:** CRWI digraph $G = \langle V, E \rangle$ with function $\text{Cost}(v_i)$ assigning a positive integer to every node $v_i \in V$.
>
> **Optimization Problem:** Minimize $\sum_{s_i \in S} \text{Cost}(s_i)$ in $S \subset V$ so that the digraph resulting from G by eliminating all the vertices in S and their adjacent edges is acyclic.

This problem is NP-hard. The related decision problem can be shown NP-complete by reduction from Karp's well known problem [7] of determining if there exists a set of vertices and their adjacent edges to remove in a general digraph that make that digraph acyclic and the set has fewer members than some fixed target. The fundamental concept of the reduction is a construction that encodes the input general digraph for Karp's problem into a digraph with membership in class CRWI. We will not present the reduction in this work.

Having established the intractability of the globally optimal solution, let us consider local solutions, minimizing the increase in file size when breaking each cycle individually. When encountering a cycle, a locally minimum solution loops through that cycle determining the minimum cost vertex and removes it.

An adversarial example establishes the incapability of the locally minimum solution to produce a reasonable global solution. In the digraph of Figure 2, with membership in CRWI, the locally minimum policy for breaking cycles looks at all cycles $(v_0, \ldots, v_i, v_0)$ for $i \in 1, 2, \ldots, k$. For each cycle, it chooses to delete the minimum cost node at cost $= C$. As a result, the algorithm deletes nodes $v_1, v_2, \ldots, v_k$. However, deleting node v_0 is the globally optimal solution.

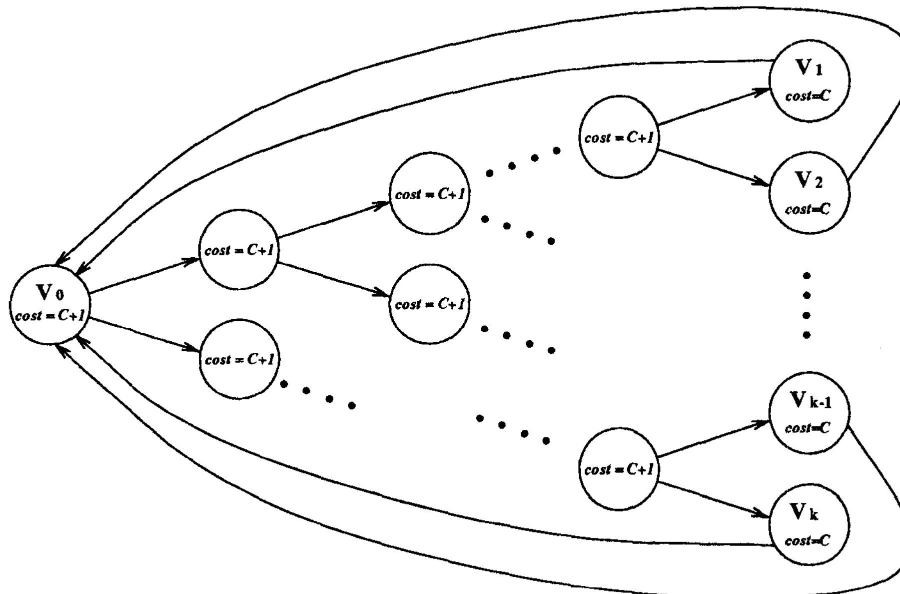

Figure 2: A CRWI digraph constructed from a binary tree by adding a directed edge from each leaf to the root node. The locally minimum cycle breaking policy performs poorly on this CRWI digraph, removing each leaf vertex, instead of the root vertex.

In this example, the size of the delta associated with the locally minimum solution grows arbitrarily larger than that of the globally optimal solution as n increases.

The locally minimum solution loops through the cycles it finds and uses total execution time proportional to the length of these cycles. For the example in Figure 2, the algorithm does $\log |V|$ work to break each cycle and takes total time $O(|V| \log |V|)$. The locally minimum solution on this example does not degrade the asymptotic performance of the in-place algorithm. However, a worst case bound on the length of the cycles examined by the locally minimum solution on CRWI digraphs remains an open problem and could be as large of $O(|V|^2)$.

Breaking cycles in constant time, arbitrarily selecting a node in the cycle to delete, also fails to approximate the globally optimal solution to within a constant. However, by choosing a node in the cycle in constant time, the asymptotic run-time of the algorithm is preserved.

The merit of the locally minimum solution, as compared to breaking cycles in constant time, is difficult to determine. On delta files whose digraphs have sparse edge relations, cycles are infrequent and looping through cycles saves compression at little cost. However, worst case analysis indicates no preference for the locally minimum solution when compared to the constant time policy, even though our intuition may indicate otherwise. This motivates a performance investigation of the run-time and compression associated with these two policies (§7).

6 Bounding the Size of the Digraph

The performance of digraph construction, topological sorting and cycle breaking depends upon the number of edges in the digraphs our algorithm constructs. We previously asserted (§4.3) that the number of edges in a CRWI digraph constructed can grow quadratically with the number of copy commands and is also bounded by the length of the version file. We present analysis to verify these assertions.

No digraph may have more than $O(|V|^2)$ edges. We now show an example of two file versions whose CRWI digraph realizes this bound to establish that it is tight. Consider a file of length L that is broken up into blocks of size $\sqrt{L}$ (Figure 3). There are $\sqrt{L}$ such blocks, $b_1, b_2, ..., b_{\sqrt{L}}$. Assume that all blocks excluding the first block in the new file, $b_2, b_2, ..., b_{\sqrt{L}}$, are all copies of the first block in the reference file. Also, the first block in the new file consists of $\sqrt{L}$ copies of length 1 from any location in the reference file.

Since each length 1 command writes into each length $\sqrt{L}$ command's read interval, there is a edge between every $\sqrt{L}$ length node and every length 1 node. This digraph has $\sqrt{L} - 1$ nodes each with out-degree $\sqrt{L}$ for total edges in $\Omega(L) = \Omega(|C|^2)$. While the length 1 copies of seem improbable, more reasonable variations of this example with larger constant size copies and more sparse edge relations may be constructed in a similar fashion.

The $\Omega(L)$ bound also turns out to be the maximum number of possible edges.

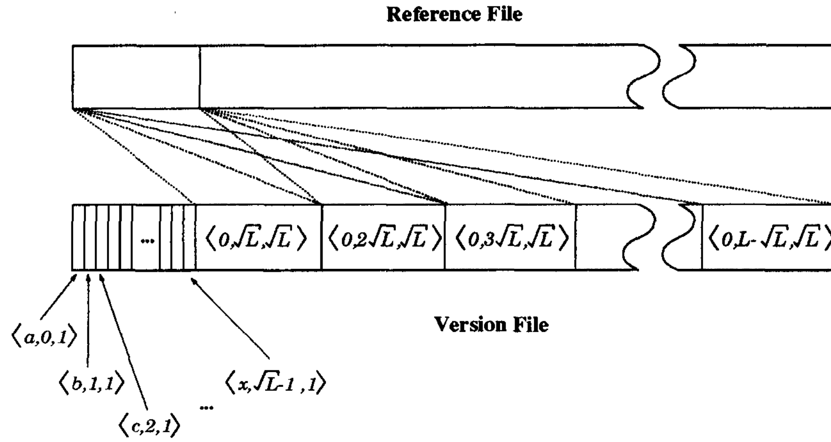

Figure 3: Reference and version file that have $O(|C|^2)$ conflicts.

Lemma 1 *For an input delta file encoding a version V of length L_V, the number of edges in the digraph generated to encode potential* WR *conflicts is less than or equal to L_V.*

Proof: There is an edge encoding a potential *WR* conflict from copy commands i to j when

$$[f_i, f_i + l_i - 1] \cap [t_j, t_j + l_j - 1] \neq \emptyset. \qquad (3)$$

Copy command i has a read interval of length l_i. Recalling that the write intervals of all copy commands are disjoint, command i conflicts with a maximum of l_i other copy commands – this occurs when the region $[f_i, f_i + l_i - 1]$ in the new version is encoded by l_i copy commands of length 1.

We also know that, for any delta encoding, the sum of the length of all read intervals is less than or equal to L_V, as no encoding reads more symbols than it writes.

As all read intervals sum to less than the length of the version file and no read interval may generate more edges than its length, the number of edges in the digraph from a delta file encoding V is less than or equal to L_V. ■

By bounding the number of edges in CRWI digraphs, we verify the performance bounds presented in §4.3.

7 Experimental Results

As we are interested in using in-place reconstruction methods to distribute software over the Internet, we extracted a large body of Internet available software and examined the performance of our algorithm on these files. They include multiple versions of the GNU tools and the BSD operating system distributions, among other data, with both binary and source files being compressed and permuted for in-place reconstruction. These data were examined with the goals of:

- determining the compression lost due to making delta files suitable for in-place reconstruction;
- comparing the relative compression performance of the constant time and locally minimum policies for breaking cycles; and
- showing the in-place conversion algorithms to be efficient when compared to delta compression algorithms on the same data.

Previous studies have indicated that delta compression algorithms can compact data for the transmission of software [1]. Delta compression algorithms compatible with in-place reconstruction compress a large body of distributed software by a factor of 4 to 10 and reduce the amount of time required to transmit these files over low bandwidth channels accordingly.

Over our sample software distributions, we found that the delta algorithm we used compressed data, on average, to 15.3% its original size and that after running our algorithm to make these files suitable for in-place reconstruction, we lost only 2.4% compression when breaking cycles using the locally minimum policy and 5.9% compression when breaking cycles in constant time.

Almost all of the lost compression can be attributed to an encoding inefficiency inherent to in-place reconstruction. We have described add commands $\langle t, l \rangle$ and copy commands $\langle f, t, l \rangle$, where both commands explicitly encode the to t or write offset in the version file. However, delta algorithms that reconstruct data in order need not explicitly encode a write offset – an add command can simply be $\langle l \rangle$ and a copy command $\langle f, l \rangle$. Since commands are applied in write order, the write offset is implicitly defined by the end offset of the previous command. By adding write offsets, a delta compression algorithm loses 1.9% of its compression before modifying the delta file for in-place reconstruction. We see

Algorithm	Δ Compress No Write Offsets	Δ Compress Write Offsets	In-Place (Constant Time)	In-Place (Local Minimum)
Compression	15.3%	17.2%	17.7%	21.2%
Encoding Loss		1.9%	1.9%	1.9%
Loss from Cycles			0.5%	4.0%
Total Loss		1.9%	2.4%	5.9%

Table 1: Compression performance of Δ compression algorithms and in-place conversion algorithms. For in-place conversion, lost compression is partitioned into loss from encoding inefficiencies and loss from breaking cycles for in-place reconstructibility.

in Table 1 the difference between the delta compression algorithm run without write offsets, compressing input files to 15.3% their original size, and the same algorithm with write offsets, compressing input files to 17.2% their original size. Except for the codewords the algorithms use, they are identical, finding the same matching strings in the input files.

We adopt the format of the add and copy codewords directly from a standard differential compression algorithms [11, 1]. While this decision eases implementation, the codewords are poorly suited to in-place reconstruction. The encoding scheme uses only a single byte to encode the length of add commands and therefore generates many short add commands. For in-place reconstructibility, the add command includes a write offset and becomes much more expensive. The many small add commands produced by the delta compression algorithm create an unnecessary encoding overhead. A redesign of the delta compression codewords for in-place reconstructibility would further reduce lost compression.

Subtracting the encoding inefficiency, we observe that the algorithm loses an additional 4.0% compression when breaking cycles in constant time and 0.5% compression for the locally minimum policy (Table 1). The locally minimum policy reclaims nearly all lost compression with respect to the constant time policy and we will later see that, in practice, it incurs no additional run-time expense. While we cannot compare the compression performance of the locally minimum policy to a solution to the NP-hard global optimization problem, 0.5% bounds the amount of possible improvement on these files. Since this improvement is significantly less than the encoding overhead for write offsets (1.9%), we feel that the locally minimum policy performs extremely well in practice.

The delta compression algorithm used to create delta files [1] before permutation operates using time $O(L_V + L_R)$ and space in $O(1)$ for reference file R of size L_R and version file V of size L_V. Our in-place conversion algorithm ($O(n \log n + L_V)$) does not guarantee as small an asymptotic bound, since the length of n is $O(L_V)$. Despite worst case analysis, we can claim in-place conversion to be efficient experimentally with data indicating that generating an in-place reconstructible file takes significantly less time than generating the input delta file.

We compare the amount of time required to generate a delta file from an input reference and version file to time used to create an in-place permutation of that delta file. Over all inputs, the in-place conversion algorithm completed in 56% the amount of total time used by the delta compression algorithm. The run-time of the in-place conversion algorithm only exceeded the delta compression run-time on 0.1% of all inputs and never took more that twice as much time.

The performance of the in-place conversion algorithm depends on the number of commands in the delta file. In practice, this value is significantly smaller than either the size of the reference and version file or the delta file itself.

Surprisingly, breaking cycles with the locally minimum policy has no apparent impact on the run-time performance of the algorithm. The time differences between these policies are insignificant as compared to the variations of system performance over many trials. While the performance difference is on average negligible, on some inputs the locally minimum policy runs noticeably slower. Infrequently, an input will contain many long cycles, and the locally minimum policy will create a slow down of up to 25% when compared to the constant time policy. However, the locally minimum policy recoups lost time on other inputs by encoding more compact delta files. Since the locally minimum policy converts smaller commands from copies to adds, it performs fewer and smaller I/O requests.

We determine experimentally that the locally minimum cycle breaking policy recovers nearly all the lost compression from breaking cycles that occurs with the constant time policy. Additionally, it does not increase the time required for the algorithm to complete. Therefore, locally minimum cycle breaking is the superior policy for every performance metric we have considered.

8 Conclusions

We have presented an algorithm that modifies delta files so that the encoded version may be reconstructed in the absence of scratch memory or storage space. Such an algo-

rithm facilitates the distribution of software to network attached devices over low bandwidth channels. Delta compression lessens the time required to transmit files over a network by compactly encoding the data to be transmitted. In-place reconstruction allows devices that do not have additional disk or memory storage resources to take advantage of delta compression technology.

The graph-theoretic algorithm to modify a delta file to be in-place reconstructible, rebuilt in the same storage the previous version occupies, rearranges the order of application of the commands in a delta file to create the new delta version. By rearranging the order of commands, data conflicts, where the delta file attempts to read from a region that it has already written, are avoided. Often, the algorithm finds an ordering of commands with no conflicts. Sometimes, the algorithm must convert data that the delta file encoded as a copy to an add command, placing that data wholly in the delta file. By converting copy commands to add commands, the algorithm trades a small degree of compression in order to achieve in-place reconstructibility.

Experimental results indicate that converting a delta file into an in-place reconstructible delta file has limited impact on compression, less than 2.5%, and that experimentally the algorithm to do so requires less time than the algorithm to generate the delta file in the first place.

In-place reconstructible delta file compression provides the benefits of delta compression for software distribution to a special class of applications – devices with limited storage and memory. In the current network computing environment, with the proliferation of network attached devices, this technology greatly decreases the time to distribute software without increasing the development cost or complexity of the receiving devices.

Acknowledgments

The authors wish to thank L. Stockmeyer for his contribution to the analysis and presentation of this algorithm, including the development of examples and the NP-hardness results concerning policies for breaking cycles. We also wish to thank R. Fagin and our anonymous reviewers who helped to focus the content of our presentation.

References

[1] M. Ajtai, R. Burns, R. Fagin, D. Long, and L. Stockmeyer. Compactly encoding arbitrary inputs with differential compression. IBM Research: *In Preparation*, 1998.

[2] G. Banga, F. Douglis, and M. Rabinovich. Optimistic deltas for WWW latency reduction. In *Proceedings of the 1998 Usenix Technical Conference*, 1998.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison–Wesley Publishing Co., 1987.

[4] R. C. Burns and D. D. E. Long. Efficient distributed backup with delta compression. In *Proceedings of the 1997 I/O in Parallel and Distributed Systems (IOPADS'97), 17 November 1997, San Jose, CA, USA*, November 1997.

[5] R. C. Burns and D. D. E. Long. A linear time, constant space differencing algorithm. In *Proceedings of the 1997 International Performance, Computing and Communications Conference (IPCCC'97), Feb. 5-7, Tempe/Phoenix, Arizona, USA*, February 1997.

[6] S. P. De Jong. Combining of changes to a source file. *IBM Technical Disclosure Bulletin*, 15(4):1186–1188, September 1972.

[7] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–104. Plenum Press, 1972.

[8] D. E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison–Wesley Publishing Co., 1968.

[9] W. Miller and E. W. Myers. A file comparison program. *Software – Practice and Experience*, 15(11):1025–1040, November 1985.

[10] J. C. Mogul, F. Douglis, A. Feldman, and B. Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. In *Proceedings of ACM SIGCOMM '97*, September 1997.

[11] C. Reichenberger. Delta storage for arbitrary non-text files. In *Proceedings of the 3rd International Workshop on Software Configuration Management, Trondheim, Norway, 12-14 June 1991*, pages 144–152. ACM, June 1991.

[12] M. J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, SE-1(4):364–370, December 1975.

[13] D. G. Severance and G. M. Lohman. Differential files: Their application to the maintenance of large databases. *ACM Transactions on Database Systems*, 1(2):256–267, September 1976.

[14] W. F. Tichy. The string-to-string correction problem with block move. *ACM Transactions on Computer Systems*, 2(4), November 1984.

[15] W. F. Tichy. RCS – A system for version control. *Software – Practice and Experience*, 15(7):637–654, July 1985.