RJ 10243 (A0205-014) May 10, 2002
Computer Science

IBM Research Report

In-Place Reconstruction of Delta Compressed Files

Randal Burns
Department of Computer Science
Johns Hopkins University
3400 N. Charles St.
Baltimore, MD 21218

Larry Stockmeyer
IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099

Darrell D. E. Long
Department of Computer Science
University of California at Santa Cruz
Santa Cruz, CA 95064

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

In-Place Reconstruction of Delta Compressed Files

Randal Burns*
Department of Computer Science
Johns Hopkins University
3400 N. Charles St., Baltimore, MD 21218
randal@cs.jhu.edu

Larry Stockmeyer
IBM Research Division
Almaden Research Center
650 Harry Rd., San Jose, CA 95120
stock@almaden.ibm.com

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

Abstract

In-place reconstruction of delta compressed data allows information on devices with limited storage capability to be updated efficiently over low-bandwidth channels. Delta compression encodes a version of data compactly as a small set of changes from a previous version. Transmitting updates to data as delta versions saves both time and bandwidth. In-place reconstruction rebuilds the new version of the data in the storage or memory space the current version occupies – no additional scratch space is needed. By combining these technologies, we support large-scale, highly-mobile applications on inexpensive hardware.

We develop an algorithm that modifies a delta compressed version to be in-place reconstructible; it trades a small amount of compression to achieve in-place reconstruction. This algorithm brings the benefits of delta compression to space-constrained machines and devices on low-bandwidth networks. Our treatment includes an implementation and experimental results that show our algorithm to be efficient in space and time and verifies that compression losses are small. Also, we give results on the computational complexity of performing this modification while minimizing lost compression. We take a data-driven approach to determine important performance features, classifying files distributed on the Internet based on their in-place properties, and exploring the scaling relationship between files and data structures used by in-place algorithms. We conclude that in-place algorithms are I/O bound and that the performance of algorithms is more sensitive to the size of inputs and outputs, rather than the computational time to modify the delta version.

keywords: Delta compression, data distribution, in-place reconstruction, storage systems, network computing, mobile computing

*The work of this author was performed in part while at the IBM Almaden Research Center.

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

1 Introduction

We develop algorithms for data distribution and version management to be used for highly-mobile and resource-limited computers over low-bandwidth networks. The software infrastructure for Internet-scale file sharing is not suitable for this class of applications, because it makes demands for network bandwidth and storage/memory space that many small computers and devices cannot meet.

While file sharing is proving to be the new prominent application for the Internet, it is limited in that data are not writable nor are versions managed. The many recent commercial and freely available systems underscore this point, examples include Freenet [1] and GnuTella [2]. Writable replicas greatly increase the complexity of file sharing – problems include update propagation and version control.

Delta compression has proved a valuable tool for managing versions and propagating updates in distributed systems and should provide the same benefits for Internet file sharing. Delta-compression has been used to reduce latency and network bandwidth for Web serving [4, 22] and backup and restore [7].

Our in-place reconstruction technology addresses one of delta compression's major shortcomings. Delta compression makes memory and storage demands that are not reasonable for low-cost, low-resource devices and small computers. In-place reconstruction allows a version to be updated by a delta in the memory or storage that it currently occupies; reconstruction needs no additional scratch space or space for a second copy. An in-place reconstructible delta file is a permutation and modification of the original delta file. This conversion comes with a small compression penalty. In-place reconstruction brings the latency and bandwidth benefits of delta compression to the space-constrained, mass-produced devices that need them the most, such as personal digital assistants, cellular phones, and wireless handhelds.

A distributed inventory management system based on mobile-handheld devices is an archetypal application for in-place technology. Many limited-capacity devices track quantities throughout an enterprise. To reduce latency, these devices cache portions of the database for read-only and update queries. Each device maintains a radio link to update its cache and run a consistency protocol. In-place reconstruction allows the devices to keep their copies of data consistent using delta compression without requiring scratch space, thereby increasing the cache utilization at target devices. Any available scratch space can be used to reduce compression loss, but no scratch space is required for correct operation. We observe that in-place reconstruction applies to both structured data (databases) and unstructured data (files), because they manipulate a delta encoding, as opposed to the original data. While algorithms for delta compressing structured data are different [9], they employ encodings that are suitable for in-place techniques.

1.1 Delta Compression and In-Place Reconstruction

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, and cellular devices. The data contents of these devices are often updated by transmitting the new version over a network. However, low bandwidth channels and heavy Internet traffic often makes the time to perform software update prohibitive.

Differential or delta compression (see, for example, [14, 6, 9, 15, 8, 3]), encoding a new version of a file compactly as a set of changes from a previous version, reduces the size of the transmitted file 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 second copy of the file. Two copies of the file must be available concurrently, as the delta file reads data from the old file version while materializing the new file version in another region of storage. This presents a problem because network attached devices often cannot store two file versions at the same time. Furthermore, adding storage to network attached devices is not viable, because keeping these devices simple limits their production costs.

We modify 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 encodes a sequence of instructions, or *commands*, for 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 version to the new version and added explicitly when portions of the new version do not appear in the reference version.

If we were to attempt naively to reconstruct an arbitrary delta file in-place, the resulting output would 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 algorithm reads have already been altered and the algorithm rebuilds an incorrect file.

We present a graph-theoretic algorithm for modifying delta files that detects situations where a delta file attempts to read from an already written region and permutes the order that the algorithm applies commands in a delta file to reduce the occurrence of conflicts. The algorithm eliminates the remaining conflicts by removing commands that copy data and adding explicitly these data to the delta file. Eliminating data copied between versions increases the size of the delta encoding but allows the algorithm to output an in-place reconstructible delta file.

Experimental results verify the viability and efficiency of modifying delta files for in-place reconstruction. Our findings indicate that our algorithm exchanges a small amount of compression for in-place reconstructibility.

Experiments also reveal an interesting property of these algorithms that conflicts with algorithmic analysis. We show in-place reconstruction algorithms to be I/O bound. In practice, the most important performance factor is the output size of the delta file. Two heuristics for eliminating data conflicts were studied in our experiments, and they show that the heuristic that loses less compression is superior to the more time-efficient heuristic that loses more compression. Any time saved in detecting and eliminating conflicts is lost when writing a larger delta file out to storage.

The graphs constructed by our algorithm form an apparently new class of directed graphs, which we call CRWI (conflicting read write interval) digraphs. Our modification algorithm is not guaranteed to minimize the amount of lost compression, but we do not expect an efficient algorithm to have this property, because we show that minimizing the lost compression is an NP-hard problem. We also consider the complexity of finding an optimally-compact in-place reconstructible delta file "from scratch", that is, directly from a reference file and a version file. We show that this problem is NP-hard; in contrast, without the requirement of in-place reconstructibility, a optimally-compact delta file can be found in polynomial time [27, 21, 23].

In Section 2, we summarize the preceding work in the field of delta compression. We describe how we encode delta files in Section 3. In Section 4, we present an algorithm that modifies delta encoded files to be in-place reconstructible. In Section 4.3, we further examine the exchange of run-time and compression performance. In Section 4.6, we present limits on the size of the graphs our algorithm generates. Section 5 presents experimental results for the execution time and compression performance of our algorithm. In Section 6 we begin to explore the properties of CRWI digraphs by giving a simple sufficient condition for a graph to be a CRWI digraph. Sections 7 and 8 contain results on the computational complexity of problems related to finding in-place reconstructible delta encodings. We present conclusions in Section 9 and mention some directions for future research in Section 10.

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 [11]. Considering data as lines of text fails to encode minimum sized delta files, as it does not examine data at a fine *granularity* and finds only matching data that are *aligned* at the beginning of a new line.

The problem of representing the changes between versions of data was formalized as string-to-string correction with block move [27] – detecting maximally matching regions of a file at an arbitrarily fine granularity without alignment. However, delta compression continued to rely on the alignment of data, as

in database records [25], and the grouping of data into block or line granularity, as in source code control systems [24, 28], to simplify the combinatorial task of finding the common and different strings between versions.

Efforts to generalize delta compression to un-aligned data 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 [21] or the greedy method [27, 23, 19] and performed this task using time quadratic in the length of the input files.

Delta compression algorithms were improved to run in linear time and linear space. Algorithms with these properties have been derived from suffix trees [30, 20, 18] and as a generalization of Lempel-Ziv data compression [14, 15, 8]. Like algorithms based on greedy methods and dynamic programming, these algorithms generate optimally compact delta encodings.

Recent advances produced algorithms that run in linear time and constant space [3]. These differencing algorithms trade a small amount of compression, verified experimentally, in order to improve performance.

Any of the linear run-time algorithms allow delta compression to scale to large input files without known structure and permits the application of delta compression to file system backup and restore [7].

Recently, applications distributing HTTP objects using delta files have emerged [4, 22]. This permits web servers to both reduce the amount of data 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 toward making small network devices HTTP compliant indicate the need to distribute data to network devices efficiently.

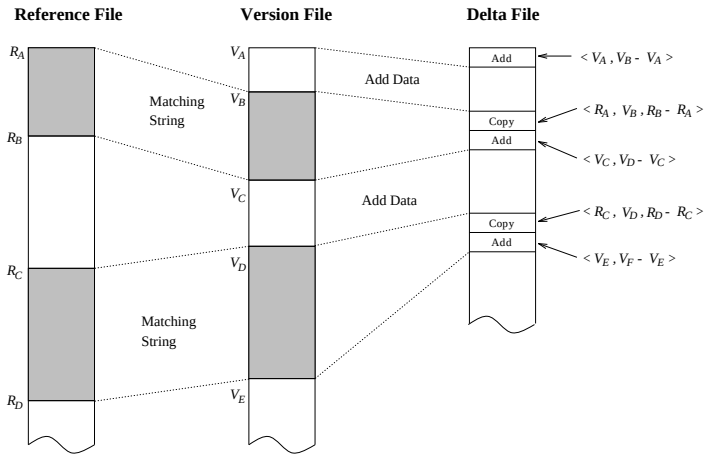
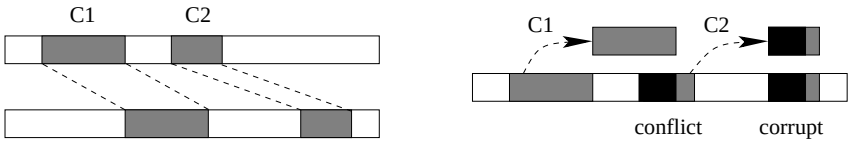


Figure 1: Encoding delta files. Common strings are encoded as *copy* commands $\langle f, t, l \rangle$ and new strings in the new file are encoded as *add* commands $\langle t, l \rangle$ followed by the string of length l of added data.

3 Encoding Delta Files

Differencing algorithms encode the changes between two file versions compactly by finding strings common to both versions. We term these files a *version file* that contains the data to be encoded and a *reference file* to which the version file is compared. Differencing algorithms encode a file by partitioning the data in the version file into strings that are encoded using copies from the reference file and strings that are added explicitly to the version file (Figure 1). Having partitioned the version file, the algorithm outputs a delta file that encodes this version. 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



(a) Delta copy (b) In-place copy

Figure 2: Data conflict and corruption when performing copy command C1 before C2.

l (length) encodes the length of the string. The l bytes of data to be added follow the command. 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 moves 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* commands. The intervals in the version file encoded by these commands are disjoint. Therefore, any permutation of the command execution order materializes the same output version file.

4 In-Place Modification Algorithms

An in-place modification algorithm changes an existing delta file into a delta file that reconstructs correctly 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 (file address range to which the command writes data) of other *copy* commands. The algorithm represents potential data conflicts in a digraph. The algorithm topologically sorts the digraph to produce an ordering on *copy* commands that reduces data conflicts. We eliminate the remaining conflicts by converting *copy* commands to *add* commands. The algorithm outputs the permuted and converted commands as an in-place reconstructible delta file. Actually, as described in more detail below, the algorithm performs permutation and conversion of commands concurrently.

4.1 Conflict Detection

Since we reconstruct files in-place, we concern ourselves with ordering commands that attempt to read a region to which another command writes. For this, we adopt the term *write before read* (*WR*) conflict [5]. For *copy* commands $\langle f_i, t_i, l_i \rangle$ and $\langle f_j, t_j, l_j \rangle$, with $i < j$, a *WR* conflict occurs when

$$[t_i, t_i + l_i - 1] \cap [f_j, f_j + l_j - 1] \neq \emptyset. \tag{1}$$

In other words, *copy* command i and j conflict if i writes to the interval from which j reads data. By denoting, for each *copy* command $\langle f_k, t_k, l_k \rangle$, the command's read interval as $\mathit{Read}_k = [f_k, f_k + l_k - 1]$ and its write interval as $\mathit{Write}_k = [t_k, t_k + l_k - 1]$, we write the condition (1) for a WR conflict as $\mathit{Write}_i \cap \mathit{Read}_j \neq \emptyset$. In Figure 2, commands C1 and C2 executed in that order generate a data conflict (blacked area) that corrupts data when a file is reconstructed in place.

This definition considers only *WR* conflicts between *copy* commands and neglects *add* commands. *Add* commands write data to the version file; they do not read data from the reference file. Consequently, an algorithm avoids all potential *WR* conflicts associated with adding data by placing *add* commands at the end of a delta file. In this way, the algorithms completes all reads associated with *copy* commands before executing the first *add* command.

Additionally, we define *WR* conflicts so that a *copy* command cannot conflict with itself. Yet, a single *copy* command's read and write intervals intersect sometimes and would seem to cause a conflict. We deal

with read and write intervals that overlap by performing the copy in a *left-to-right* or *right-to-left* manner. For command $\langle f, t, l \rangle$, if $f \geq t$, we copy the string byte by byte starting at the left-hand side when reconstructing the original file. Since, the f (from) offset always exceeds the t (to) offset in the new file, a *left-to-right* copy never reads a byte over-written 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 backward. 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.

For many delta files, no possible permutation eliminates all *WR* conflicts. Consequently, we require the conversion of *copy* commands to *add* commands to create correct in-place reconstructible files for all inputs.

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

$$(\forall j) \left[\mathit{Read}_j \cap \left(\bigcup_{i=1}^{j-1} \mathit{Write}_i \right) = \emptyset \right], \tag{2}$$

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

4.2 CRWI Digraphs

To find a permutation that reduces *WR* conflicts, we represent potential conflicts between the *copy* commands in a digraph and topologically sort this digraph. A topological sort on digraph $G = (V, E)$ 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 vertex u to vertex 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. We call a digraph obtained from a delta file in this way a *conflicting read write interval* (*CRWI*) digraph. A topologically sorted version of this graph adheres to the requirement for in-place reconstruction (Equation 2). To the best of our knowledge, the class of CRWI digraphs has not been defined previously. While we know little about its structure, it is clearly smaller than the class of all digraphs. For example, the CRWI class does not include any complete digraphs with more than two vertices.

4.3 Strategies for Breaking Cycles

As total topological orderings are possible only on acyclic digraphs and CRWI digraphs may contain cycles, we enhance a standard topological sort to break cycles and output a total topological order on a *subgraph*. Depth-first search implementations of topological sort [10] are modified easily to detect cycles. Upon detecting a cycle, our modified sort breaks the cycle by removing a vertex. When completing this enhanced sort, the sort outputs 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.

As the string that contains the encoded data follows converted *add*, this replacement reduces compression in the delta file. We define the amount of compression lost upon deleting a vertex to be the *cost* of deletion. Based on this cost function, we formulate the optimization problem of finding the minimum cost set of vertices to delete to make a digraph acyclic. A *copy* command is an ordered triple $\langle f, t, l \rangle$. An *add* command is an ordered double $\langle t, l \rangle$ followed by the l bytes of data to be added to the new version of the file. Replacing a *copy* command with an *add* command increases the delta file size by $l - \|f\|$, where $\|f\|$ denotes the size of the encoding of offset f. Thus, the vertex that corresponds to the copy command $\langle f, t, l \rangle$ is assigned cost $l - \|f\|$.

When converting a digraph into an acyclic digraph by deleting vertices, an in-place conversion algorithm minimizes the amount of compression lost by selecting a set of vertices with the smallest total cost. This problem, called the FEEDBACK VERTEX SET problem, was shown by Karp [16] to be NP-hard for general digraphs. In Section 7 we show that it remains NP-hard even when restricted to CRWI digraphs. Thus, we do not expect an efficient algorithm to minimize the cost in general. For our implementation of in-place conversion, we examine two efficient, but not optimal, 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 vertex. This policy performs no extra work when breaking cycles. The *local-minimum* policy detects a cycle and loops through all vertices in the cycle to determine and then delete the minimum cost vertex. The local-minimum policy may perform as much additional work as the total length of cycles found by the algorithm. Although these policies perform well in our experiments, we note in Section 4.7 that they do not guarantee that the total cost of deletion is within a constant factor of the optimum.

4.4 Generating Conflict Free Permutations

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, we 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: $i < j \longleftrightarrow t_i < t_j$. Sorting the copy commands allows us to perform binary search when looking for a copy command at a given write offset.

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

$$\overrightarrow{v_i v_j} \longleftrightarrow \mathit{Read}_i \cap \mathit{Write}_j \neq \emptyset \longleftrightarrow [f_i, f_i + l_i - 1] \cap [t_j, t_j + l_j - 1] \neq \emptyset.$$

4. Perform a topological sort on the vertices of the digraph. This sort also detects cycles in the digraph and breaks them. When breaking a cycle, select one vertex on the cycle, using either the local-minimum or constant-time cycle breaking policy, and remove it (we give further details below). We replace the data encoded in its *copy* command with an equivalent *add* command, which is put into set A. The output of the topological sort orders the remaining *copy* commands so that they obey the property in Equation 2.

5. 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.

For completeness, we give a brief description of how a standard depth-first search (DFS) algorithm was modified to perform step 4 in our implementation, as these details affect both the results of our experiments and the asymptotic worst-case time bounds. As described, the algorithm outputs the un-removed copy commands in reverse topologically sorted order; to output them in topologically sorted order simply reverse the edge relation. A DFS algorithm uses a stack to visit the vertices of a digraph in a certain order. The algorithm marks each vertex either *un-visited*, *on-stack*, or *finished*. Initially, every vertex is marked *un-visited*. Until no more *un-visited* vertices exist, the algorithm chooses a un-visited vertex u and calls VISIT(u). The procedure VISIT(u) marks u as *on-stack*, pushes u on the stack, and examines each vertex w such that there is an edge $\vec{uw}$ in the graph. For each such w: (1) if w is marked *finished* then w is not processed further; (2) if w is marked *un-visited* then VISIT(w) is performed; (3) if w is marked *on-stack* then the vertices between u and w on the stack form a directed cycle, which must be broken. For the *constant-time* policy, u is popped from the stack and removed from the graph. Letting p denote the new top of the stack, the execution of VISIT(p) continues as though u were marked *finished*. For the *local-minimum* policy, the algorithm loops through all vertices on the cycle to find one of minimum cost, that is, one whose removal causes the smallest increase in the size of the delta file; call this vertex r. Vertices u through r are popped from the stack and marked *un-visited*, except r which is removed. If there is a vertex p on the top of the stack, then the execution of VISIT(p) continues as though r were marked *finished*. Recall that we are describing an execution of VISIT(u) by examining all w such that there is an edge $\vec{uw}$. After all such w have been examined, u is marked *finished*, u is popped from the stack, and the copy command corresponding to vertex u is written in reverse sorted order. Using the constant-time policy, this procedure has the same running time as DFS, namely, $O(|V| + |E|)$. Using the local-minimum policy, when the algorithm removes a vertex, it retains some of the work (marking) that the DFS has done. However, in the worst case, the entire stack pops after each vertex removal, causing running time proportional to $|V|^2$. (While we can construct examples where the time is proportional to $|V|^2$, we do not observe this worst-case behavior in our experiments.)

4.5 Algorithmic Performance

Suppose that the algorithm is given a delta file consisting of a set C of *copy* commands and a set A of *add* commands. The presented algorithm uses time $O(|C| \log |C|)$ both for sorting the *copy* commands by write order and for finding conflicting commands, using binary search on the sorted write intervals for the $|V|$ vertices 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 $O(|E|)$. As noted above, step 4 takes time $O(|V| + |E|)$ using the constant-time policy and time $O(|V|^2)$ using the local-minimum policy. The total worst-case execution time is thus $O(|C| \log |C| + |E| + |A|)$ for the constant-time policy and $O(|V|^2 + |A|)$ for the local-minimum policy. The algorithm uses space $O(|E| + |C| + |A|)$. Letting n denote the total number of commands in the delta file, the graph contains as many vertices as *copy* commands. Therefore, $|V| = |C| = O(n)$. The same is true of *add* commands, $|A| = O(n)$. However, we have no bound for the number of edges, except the trivial bound $O(|V|^2)$ for general digraphs. (In Section 4.6, we demonstrate by example that our algorithm can generate a digraph having a number of edges meeting this bound.) On the other hand, we also show that the number of edges in digraphs generated by our algorithm is linear in the length of the version file $\mathcal{V}$ that the delta file encodes (Lemma 1). We denote the length of $\mathcal{V}$ by $L_{\mathcal{V}}$.

Substituting these bounds on $|E|$ into the performance expressions, for an input delta file containing n commands encoding a version file of length $L_{\mathcal{V}}$, the worst-case running time of our algorithm is $O(n \log n + \min(L_{\mathcal{V}}, n^2))$ using the constant-time policy and $O(n^2)$ using the local-minimum policy. In either case, the space is $O(n + \min(L_{\mathcal{V}}, n^2))$.

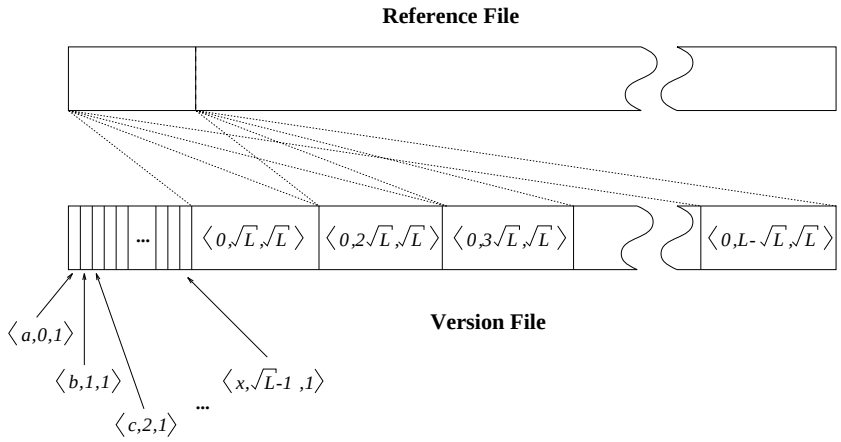


Figure 3: Reference and version file that have $O(|C|^2)$ conflicts.

4.6 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 asserted previously (Section 4.5) that the number of edges in a CRWI digraph constructed grows quadratically with the number of *copy* commands and is bounded above by the length of the version file. We now verify these assertions.

No digraph has more than $O(|V|^2)$ edges. To establish that this bound is tight for CRWI digraphs, we show an example of a delta file whose CRWI digraph realizes this bound. Consider a version file of length L that is broken up into blocks of length $\sqrt{L}$ (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 version 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 version file consists of $\sqrt{L}$ copies of length 1 from any location in the reference file. A delta file for this reference and version file consists of $\sqrt{L}$ "short" *copy* commands, each of length 1, and $\sqrt{L} - 1$ "long" *copy* commands, each of length $\sqrt{L}$. Since each short command writes into each long command's read interval, a CRWI digraph for this delta file has an edge from every vertex representing a long command to every vertex representing a short command. This digraph has $\sqrt{L} - 1$ vertices each with out-degree $\sqrt{L}$ for total edges in $\Omega(L) = \Omega(|C|^2)$.

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

Lemma 1 *For a delta file encoding a version file $\mathcal{V}$ of length $L_{\mathcal{V}}$, the number of edges in the digraph representing potential* WR *conflicts at most $L_{\mathcal{V}}$.*

Proof. The CRWI digraph has an edge representing a potential *WR* conflict from *copy* command i to *copy* command j when

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

Copy command i has a read interval of length l_i. Recalling that the write intervals of all *copy* commands are disjoint, there are at most l_i edges directed out of copy command i – this occurs when the region $[f_i, f_i + l_i - 1]$ in the version file is encoded by l_i *copy* commands of length 1. We also know that, for any delta encoding, the sum of the lengths of all read intervals is less than or equal to $L_{\mathcal{V}}$, as no encoding reads more symbols than it writes. As all read intervals sum to $\leq L_{\mathcal{V}}$, and no read interval generates more out-edges than its length, the number of edges in the digraph from a delta file encoding $\mathcal{V}$ is less than or equal to $L_{\mathcal{V}}$. ■

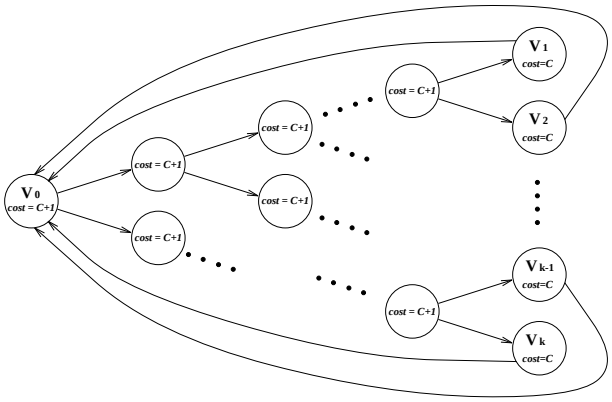


Figure 4: A CRWI digraph constructed from a binary tree by adding a directed edge from each leaf to the root vertex. Each leaf has cost C and each other vertex has cost $C + 1$. The local-minimum cycle breaking policy performs poorly on this CRWI digraph, removing each leaf vertex, instead of the root vertex.

If each *copy* command in the delta file encodes a string of length at least ℓ, then a similar proof shows that there are at most $L_{\mathcal{V}}/\ell$ edges.

By bounding the number of edges in CRWI digraphs, we verify the performance bounds presented in Section 4.5.

4.7 Non-Optimality of the Local-Minimum Policy

An adversarial example shows that the cost of a solution (a set of deleted vertices) found using the local-minimum policy is not bounded above by any constant times the optimal cost. Consider the digraph of Figure 4; Lemma 2 in Section 6 shows that this is a CRWI digraph. The local-minimum policy for breaking cycles looks at the k cycles $(v_0, \ldots, v_i, v_0)$ for $i = 1, 2, \ldots, k$. For each cycle, it chooses to delete the minimum cost vertex – vertex v_i with cost C. As a result, the algorithm deletes vertices $v_1, v_2, \ldots, v_k$, incurring total cost kC. However, deleting vertex v_0, at cost $C + 1$, is the globally optimal solution. If we further assume that the original delta file contains only the $2k - 1$ copy commands in Figure 4 and that the size of each *copy* command is c, then the size of the delta file generated by the local-minimum solution is $(2k - 1)c + kC$, the size of the optimal delta file is $(2k - 1)c + C + 1$, and the ratio of these two sizes approaches $1 + C/(2c)$ for large k. As C/c can be arbitrarily large, this ratio is not bounded by a constant.

The merit of the local-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 local-minimum solution when compared to the constant-time policy. This motivates a performance investigation of the run-time and compression associated with these two policies (Section 5).

5 Experimental Results

As we are interested in using in-place reconstruction to distribute software, we extracted a large body of Internet available software and examined the compression and execution time performance of our algorithm on these files. Sample files 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 loss due to making delta files in-place reconstructible;
- comparing the constant-time and local-minimum policies for breaking cycles;
- showing in-place conversion algorithms to be efficient when compared with delta compression algorithms on the same data; and

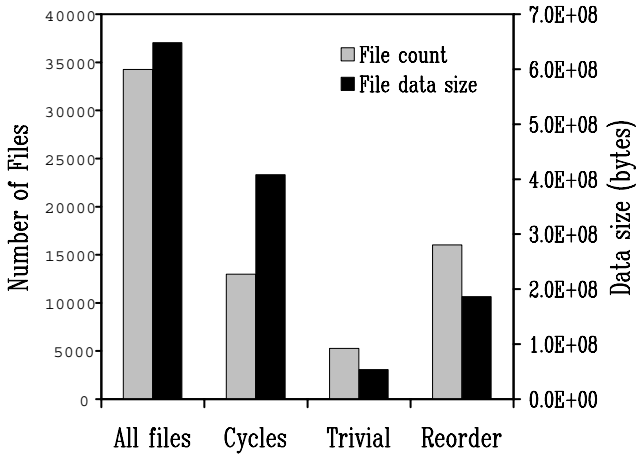


Figure 5: File counts and data size.

- characterizing the graphs created by the algorithm.

In all cases, we obtained the original delta files using the correcting 1.5-pass delta compression algorithm [3].

We categorize the delta files in our experiments into 3 groups that describe what operations were required to make files in-place reconstructible. Experiments were conducted over more than 34,000 delta files totaling 6.5MB (Megabytes). Of these files (Figure 5), 63% of the files contained cycles that needed to be broken. 29% did not have cycles, but needed to have *copy* commands reordered. The remaining 8% of files were trivially in-place reconstructible; *i.e.*, none of the *copy* commands conflicted. For trivial files, performing copies before adds creates an in-place delta.

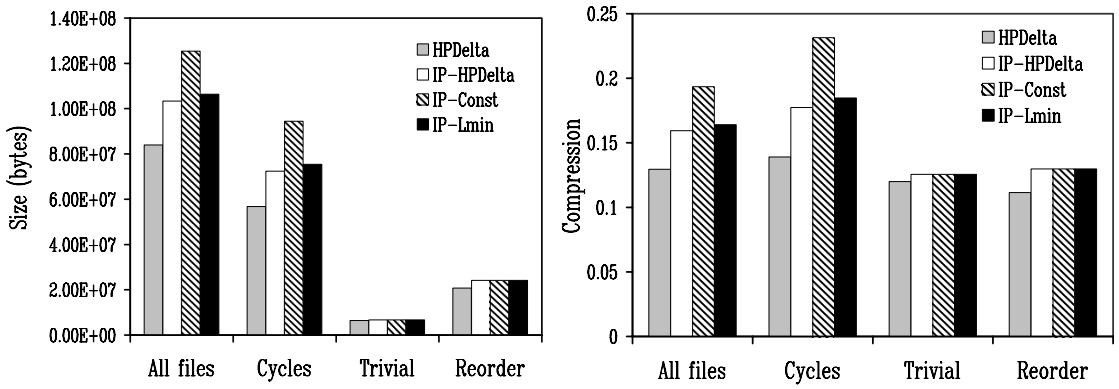
The amount of data in files is distributed differently across the three categories than are the file counts. Files with cycles contain over 4MB of data with an average file size of 31.4KB. Files that need copy commands reordered hold 1.9MB of data, with an average file size of 11.6KB. Trivially in-place reconstructible files occupy 585KB of data with an average file size of 10.2KB.

The distribution of files and data across the three categories confirms that efficient algorithms for cycle breaking and command reordering are needed to deliver delta compressed data in-place. While most delta files do not contain cycles, those that do have cycles contain the majority of the data.

We group compression results into the same categories. Figure 6(a) shows the relative size of the delta files and Figure 6(b) shows compression (size of delta files as a fraction of the original file size). For each category and for all files, we report data for four algorithms: the unmodified correcting 1.5-pass delta compression algorithm [3] (HPDelta); the correcting 1.5-pass delta compression algorithm modified so that code-words are in-place reconstructible (IP-HPDelta); the in-place modification algorithm using the local-minimum cycle breaking policy (IP-Lmin); and the in-place modification algorithm using the constant-time cycle breaking policy (IP-Const).

The HPDelta algorithm is a linear time, constant space algorithm for generating delta compressed files. It outputs *copy* and *add* commands using a code-word format similar to industry standards [17].

The IP-HPDelta algorithm is a modification of HPDelta to output code-words that are suitable for in-place reconstruction. Throughout this paper, we have described *add* commands $\langle t, l \rangle$ and *copy* commands $\langle f, t, l \rangle$, where both commands encode explicitly the to t or write offset in the version file. However, delta algorithms that reconstruct data in write 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 end offset of the previous command implies the write offset of the current command implicitly. The code-words of IP-HPDelta are modified to make the write offset explicit. The explicit write offset allows our algorithm to



(a) Delta size

(b) Delta compression

Figure 6: Compression performance

reorder copy commands. This extra field in each code-word introduces a per-command overhead in a delta file. The amount of overhead varies, depending upon the number of commands and the original size of the delta file. Encoding overhead incurs a 3% compression loss over all files.

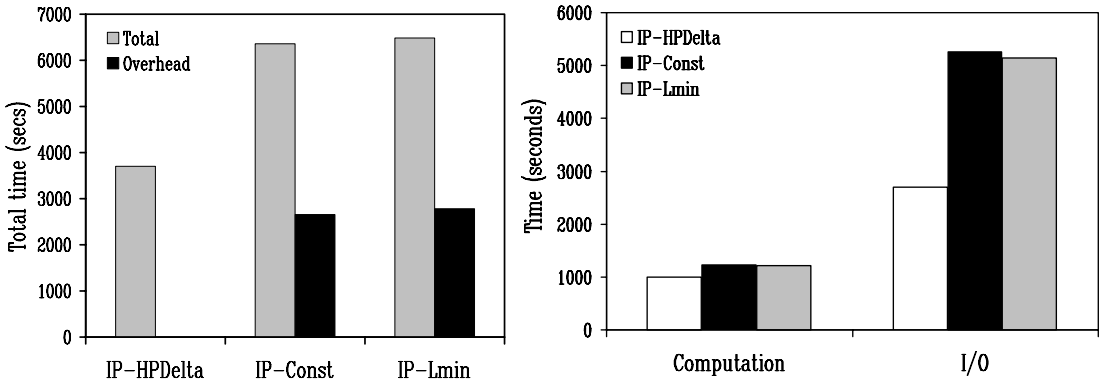
From the IP-HPDelta algorithm, we derive the IP-Const and IP-Lmin algorithms. They run the IP-HPDelta algorithm to generate a delta file and then permute and modify the commands according to our technique to make the delta file in-place reconstructible. The IP-Const algorithm implements the constant-time policy and the IP-Lmin algorithm implements the local-minimum policy.

Experimental results indicate the amount of compression lost due to in-place reconstruction and divides the loss into encoding overhead and cycle breaking. Over all files, HPDelta compresses data to 12.9% its original size. IP-HPDelta compresses data to 15.9%, losing 3% compression to encoding overhead. IP-Const loses an additional 3.4% compression by breaking cycles for a total compression loss of 6.4%. In contrast, IP-Lmin loses less than 0.5% compression for a total loss of less than 3.5%. The local-minimum policy performs excellently in practice, because compression losses are small when compared with encoding overheads. With IP-Lmin, cycle breaking accounts for less than 15% of the loss. For comparison, with IP-Const cycle breaking more than doubles the compression loss.

For reorder and trivial in-place delta files, no cycles are present and no compression lost. Encoding overhead makes up all lost compression – 0.5% for trivial delta files and 1.8% for reordered files.

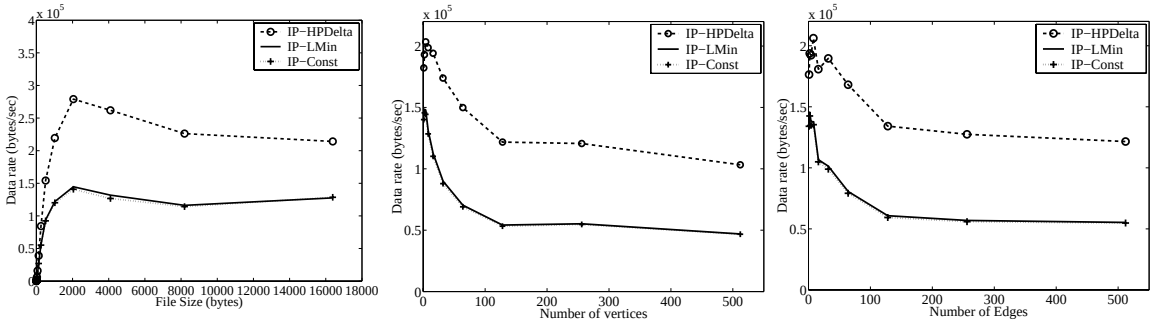
Files with cycles exhibit an encoding overhead of 3.8% and lose 5.4% and 0.7% to cycle breaking for the IP-Const and IP-Lmin respectively. Because files with cycles contain the majority of the data, the results for files with cycles dominate the results for all files.

In-place algorithms incur execution time overheads when performing additional I/O when permuting the commands in a delta file. An in-place algorithm must generate a delta file and then modify the file to have the in-place property. Since a delta file does not necessarily fit in main memory, in-place algorithms create an intermediate file that contains the output of the delta compression algorithm. This intermediate output serves as the input for the algorithm that modifies/permutes commands. We present execution-time results in Figure 7(a) for both in-place algorithms – IP-Const and IP-Lmin. IP-Lmin and IP-Const perform all of the steps of the base algorithm (IP-HPDelta) before manipulating the intermediate file. Results show that the extra work incurs an overhead of about 75%. However, figure 7(b) shows that almost all of this overhead comes from additional I/O. We conclude that the algorithmic tasks for in-place reconstruction are small when compared with the effort compressing data (about 10% the run-time) and miniscule compared to the costs of performing file I/O.



(a) Overhead (b) Breakdown

Figure 7: Run-time results



(a) File Size (b) Vertices (c) Edges

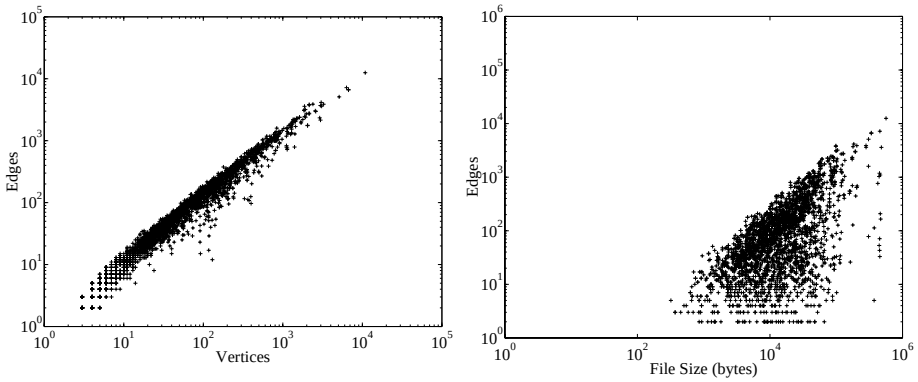
Figure 8: Run-time results

Despite inferior worst-case run-time bounds, the local-minimum policy runs faster than the constant-time policy in practice. Because file I/O dominates the run-time costs and because IP-Lmin creates a smaller delta file, it takes less total time than the theoretically superior IP-Const. In fact, IP-Const spends 2.2% more time performing I/O as a direct result of the files being 2.9% larger. IP-Lmin even uses slightly less time performing computation than IP-Const, which has to manipulate more data in memory.

Examining run-time results in more detail continues to show that IP-Lmin outperforms IP-Const, even for the largest and most complex input files. In Figure 8, we see how run-time performance varies with the input file size and with the size of the graph the algorithm creates (number of edges and vertices); these plots measure run time by data rate – file size (bytes) divided by run time (seconds).

Owing to start-up costs, data rates increase with file size up to a point, past which rates tend to stabilize. The algorithms must load and initialize data structures. For small files, these costs dominate, and data rates are lower and increase linearly with the file size (Figure 8(a)). For files larger than 2000 bytes, rates tend to stabilize, exhibiting some variance, but neither increasing or decreasing as a trend. These results indicate that for inputs that amortize start-up costs, in-place algorithms exhibit a data rate that does not vary with the size of the input – a known property of the HPDelta algorithm [3]. IP-Lmin performs slightly better than IP-Const always.

The performance of all algorithms degrades as the size of the CRWI graphs increase. Figure 8(b) shows the relative performance of the algorithms as a function of the number of vertices, and Figure 8(c) shows



(a) Edges versus Vertices

(b) Edges versus File Size

Figure 9: Edges in delta files that contain cycles.

this for the number of edges. For smaller graphs, performance degrades quickly as the graph size increases. For larger graphs, performance degrades more slowly. The graph size corresponds directly to the number of copy commands in a delta file. The more commands, the more I/O operations the algorithm must execute. Often more vertices means more small I/O rather than fewer large I/O, resulting in lower data rates.

Surprisingly, IP-Lmin continues to out-perform IP-Const even for the largest graphs. Analysis would indicate that the performance of IP-Lmin and IP-Const should diverge as the number of edges increase. But no evidence of divergent performance exists. We attribute this to two factors: (1) graphs are relatively small and (2) all algorithms are I/O bound.

In Figure 9, we look at some statistical measures of graphs constructed when creating in-place delta files, restricted to those graphs that contain cycles. While graphs can be quite large, a maximum of 11503 vertices and 16694 edges, the number of edges scales linearly with the number of vertices and less than linearly with input file size. The constructed graphs do not exhibit edge relations that approach the $O(|V|^2)$ upper bound. Therefore, data rate performance should not degrade as the number of edges increases. For example consider two files as inputs to the IP-Lmin algorithm – one with a graph that contains twice the edges of the other. Based on our result, we expect the larger graph to have twice as many vertices and encode twice as much data. While the larger instance does twice the work breaking cycles, it benefits from reorganizing twice as much data, realizing the same data rate.

The linear scaling of edges with vertices and file size matches our intuition about the nature of delta compressed data. Delta compression encodes multiple versions of the same data. Therefore, we expect matching regions between these files (encoded as edges in a CRWI graph) to have spatial locality; *i.e.*, the same string often appears in the same portion of a file. These input data do not exhibit correlation between all regions of a file that results in dense edge relations. Additionally, delta compression algorithms localize matching between files, correlating or synchronizing regions of file data [3]. All of these factors result in the linear scaling that we observe.

6 A Sufficient Condition for CRWI Digraphs

In this section we give a simple sufficient condition (Lemma 2) for a digraph to be a CRWI digraph. We use this result in Sections 7 and 8. We state and prove the lemma in greater generality than we need in Sections 7 and 8, because the more general lemma is not harder to prove and it is of separate interest as a sufficiency condition for CRWI digraphs.

We begin by recalling the definition of a CRWI digraph and defining the CRWI digraphs meeting two restrictions. An *interval* is of the form $I = [i, j] \stackrel{\text{def}}{=} \{i, i+1, \ldots, j\}$ where i and j are integers with

$0 \leq i \leq j$. Let $|I|$ denote the *length* of I, that is, $j - i + 1$. A *read-write interval set* (*RWIS*) has the form $(\mathbf{R}, \mathbf{W})$ where $\mathbf{R} = \{R(1), \ldots, R(n)\}$ and $\mathbf{W} = \{W(1), \ldots, W(n)\}$ are sets of intervals such that the intervals in $\mathbf{W}$ are pairwise disjoint and $|R(v)| = |W(v)|$ for $1 \leq v \leq n$. Given a RWIS $(\mathbf{R}, \mathbf{W})$ as above, define the digraph $graph(\mathbf{R}, \mathbf{W})$ as follows: (i) the vertices of $graph(\mathbf{R}, \mathbf{W})$ are $1, \ldots, n$; and (ii) for each pair v, w of vertices with $v \neq w$, there is an edge $\vec{vw}$ in $graph(\mathbf{R}, \mathbf{W})$ iff $R(v) \cap W(w) \neq \emptyset$.

A digraph $G = (V, E)$ is a *CRWI digraph* if $G = graph(\mathbf{R}, \mathbf{W})$ for some RWIS $(\mathbf{R}, \mathbf{W})$. Furthermore, G is a *disjoint-read CRWI digraph* if in addition the intervals in $\mathbf{R}$ are pairwise disjoint. The motivation for this restriction is that if a version string $\mathcal{V}$ is obtained from a reference string $\mathcal{R}$ by moving, inserting and deleting substrings, then a delta encoding of $\mathcal{V}$ could have little or no need to copy data from the same region of $\mathcal{R}$ more than once. An NP-hardness result with the disjoint-read restriction tells us that the ability of a delta encoding to copy data from the same region more than once is not essential to the hardness of the problem. Let $\mathbf{N}^+$ denote the positive integers. A digraph G with cost function $Cost : V \rightarrow \mathbf{N}^+$ is a *length-cost CRWI digraph* if there is an RWIS $(\mathbf{R}, \mathbf{W})$ such that $G = graph(\mathbf{R}, \mathbf{W})$ and $|R(v)| = Cost(v)$ for all $1 \leq v \leq n$. The motivation for the length-cost restriction is that replacing a copy of a long string s by an add of s causes the length of the delta encoding to increase by approximately the length of s. If in addition the intervals in $\mathbf{R}$ are pairwise disjoint, then G is a *disjoint-read length-cost CRWI digraph*. We let $(G, Cost)$ denote the digraph G with cost function *Cost*.

For a digraph G and a vertex v of G, let $indeg(v)$ denote the number of edges directed into v, and let $outdeg(v)$ denote the number of edges directed out of v. Define $indeg(G)$ to be the maximum of $indeg(v)$ over all vertices v of G, and define $outdeg(G)$ to be the maximum of $outdeg(v)$ over all vertices v of G. The digraph G has the *1-or-1 edge property* if, for each edge $\vec{vw}$ of G, either $outdeg(v) = 1$ or $indeg(w) = 1$ (or both).

Lemma 2

1. *Let G be a digraph. If G has the 1-or-1 edge property then G is a CRWI digraph. If in addition $indeg(G) \leq 2$, then G is a disjoint-read CRWI digraph.*

2. *Let $G = (V, E)$ be a digraph and let $Cost : V \rightarrow \mathbf{N}^+$ with $Cost(v) \geq 2$ for all $v \in V$. If G has the 1-or-1 edge property and $outdeg(G) \leq 2$, then $(G, Cost)$ is a length-cost CRWI digraph. If in addition $indeg(G) \leq 2$, then $(G, Cost)$ is a disjoint-read length-cost CRWI digraph.*

We give the formal proof of this lemma, which is somewhat tedious, in Appendix A. Here we briefly outline how the assumption that G has the 1-or-1 edge property is used in the proof. Suppose that $indeg(w) \geq 2$ and let $v_1, v_2, \ldots, v_d$ be the vertices such that there is an edge $\vec{v_i w}$ for $1 \leq i \leq d$ (see Figure 11 in the Appendix). By the 1-or-1 edge property, $outdeg(v_i) = 1$ for all i. Then we choose the read intervals $R(v_1), R(v_2), \ldots, R(v_d)$ consecutively and choose the write interval $W(w)$ so that it intersects all of these read intervals. Because $outdeg(v_i) = 1$ for all i, there does not exist a vertex $w' \neq w$ such that $R(v_i)$ intersects $W(w')$. Therefore, the order of the intervals $R(v_1), R(v_2), \ldots, R(v_d)$ does not matter, and we are not forced to choose $W(w')$ so that it intersects $W(w)$. Similarly, suppose that $outdeg(v) \geq 2$ and let $w_1, w_2, \ldots, w_d$ be the vertices such that there is an edge $\vec{v w_i}$ for $1 \leq i \leq d$ (see Figure 11). By the 1-or-1 edge property, $indeg(w_i) = 1$ for all i, so there does not exist a $v' \neq v$ such that $R(v')$ intersects $W(w_i)$. Therefore, we can choose $W(w_1), W(w_2), \ldots, W(w_d)$ consecutively and their order does not matter.

While Lemma 2 shows that the 1-or-1 edge property is a sufficient condition for a digraph to be a CRWI digraph, it is not necessary. This is shown by the graph in Figure 10(a), which does not have the 1-or-1 edge property but is a CRWI digraph, in fact, a disjoint-read length-cost CRWI digraph for any cost function with $Cost(v) \geq 2$ for all v. On the other hand, the conditions $outdeg(G) \leq 2$ and $indeg(G) \leq 2$ alone are not sufficient. This is shown by the graph in Figure 10(b), which is not a CRWI digraph.

(a) (b)

Figure 10: (a) A disjoint-read length-cost CRWI digraph that does not have the 1-or-1 edge property. (b) A graph with $outdeg \leq 2$ and $indeg \leq 2$ that is not a CRWI digraph.

7 Optimal Cycle Breaking on CRWI Digraphs is NP-hard

In this section we prove the result mentioned in Section 4.3, that given a CRWI digraph G and a cost function on its vertices, finding a minimum-cost set of vertices whose removal breaks all cycles in G is an NP-hard problem. Moreover, NP-hardness holds even when the problem is restricted to the case that (G, Cost) is a disjoint-read length-cost CRWI digraph and all costs are the same.

For a digraph $G = (V, E)$, a *feedback vertex set* (*FVS*) is a set $S \subseteq V$ such that the digraph obtained from G by deleting the vertices in S and their incident edges is acyclic. Define $\phi(G)$ to be the minimum size of an FVS for G. Karp [16] has shown that the following decision problem is NP-complete.

FEEDBACK VERTEX SET
Instance: A digraph G and a $K \in \mathbf{N}^+$.
Question: Is $\phi(G) \leq K$?

His proof does not show that the problem is NP-complete when G is restricted to be a CRWI digraph. Because we are interested in the vertex-weighted version of this problem where G is a CRWI digraph, we define the following decision problem.

WEIGHTED CRWI FEEDBACK VERTEX SET
Instance: A CRWI digraph $G = (V, E)$, a function $\mathit{Cost} : V \to \mathbf{N}^+$, and a $K \in \mathbf{N}^+$.
Question: Is there a feedback vertex set S for G such that $\sum_{v \in S} \mathit{Cost}(v) \leq K$?

The following lemma is the basis for the proof of NP-completeness of this problem.

Lemma 3 *There is a polynomial-time transformation that takes an arbitrary digraph $G' = (V', E')$ and produces a digraph $G = (V, E)$ such that G has the 1-or-1 edge property, $outdeg(G) \leq 2$, $indeg(G) \leq 2$, $|V| \leq 4|V'|^2$, and $\phi(G) = \phi(G')$.*

Proof. The digraph G contains the directed subgraph D_v for each $v \in V'$. The subgraph D_v consists of the vertex $\tilde{v}$, a directed binary in-tree $T_{\text{in},v}$ with root $\tilde{v}$ and $indeg(v)$ leaves (*i.e.*, all edges are directed from the leaves toward the root $\tilde{v}$), and a directed binary out-tree $T_{\text{out},v}$ with root $\tilde{v}$ and $outdeg(v)$ leaves (*i.e.*, all edges are directed from the root $\tilde{v}$ toward the leaves). If $indeg(v) = 0$ (resp., $outdeg(v) = 0$) then $T_{\text{in},v}$ (resp., $T_{\text{out},v}$) is the single vertex $\tilde{v}$. For each edge $\vec{xy}$ of G', add to G an edge from a leaf of $T_{\text{out},x}$ to a leaf of $T_{\text{in},y}$, such that each leaf is an end-point of exactly one such "added edge". By construction, $outdeg(G) \leq 2$ and $indeg(G) \leq 2$. To see that the 1-or-1 edge property holds: Let $e = \vec{vw}$ be an arbitrary edge of G; if e is an edge of some in-tree, then $outdeg(v) = 1$; if e is an edge of some out-tree, then $indeg(w) = 1$; and if e is an added edge, then $outdeg(v) = indeg(w) = 1$. To show that $|V| \leq 4|V'|^2$, it is enough to note that, for each $v \in V'$ having $indeg(v) + outdeg(v) \neq 0$, the number of vertices of D_v is at most $2(indeg(v) + outdeg(v)) \leq 4|V'|$.

It remains to show that $\phi(G) = \phi(G')$. Say first that S' is a FVS for G' with $|S'| = \phi(G')$. It is clear that $S = \{\, \tilde{v} \mid v \in S' \,\}$ is a FVS for G with $|S| = |S'|$, because every path from a leaf of $T_{\text{in},v}$ to a leaf of $T_{\text{out},v}$ must pass through $\tilde{v}$. Therefore, $\phi(G) \leq |S| = |S'| = \phi(G')$. Say now that S is a FVS for G with

$|S| = \phi(G)$. Define $S' \subseteq V'$ by placing v in S' iff at least one vertex of D_v is in S. Obviously, $|S'| \leq |S|$. It is easy to see that S' is a FVS for G', because if C' is a cycle in G' that passes through vertices $v_1, \ldots, v_m$ and none of these vertices belong to S', then no vertex of D_{v_i} for $1 \leq i \leq m$ can belong to S. So there is a cycle in G, obtained from C', that passes through no vertex of S; this contradicts the assumption that S is a FVS for G. Therefore, $\phi(G') \leq |S'| \leq |S| = \phi(G)$. ■

Theorem 1 WEIGHTED CRWI FEEDBACK VERTEX SET *is NP-complete. Moreover, for each constant $C \geq 2$, it remains NP-complete when restricted to instances where $(G, Cost)$ is a disjoint-read length-cost CRWI digraph, $Cost(v) = C$ for all v, $indeg(G) \leq 2$, and $outdeg(G) \leq 2$.*

Proof. The problem clearly belongs to NP. To prove NP-completeness we give a polynomial-time reduction from FEEDBACK VERTEX SET to WEIGHTED CRWI FEEDBACK VERTEX SET. Let G' and K' be an instance of FEEDBACK VERTEX SET, where G' is an arbitrary digraph. Transform G' to G using Lemma 3. Let $Cost \equiv C$. Because G has the 1-or-1 edge property, $outdeg(G) \leq 2$, and $indeg(G) \leq 2$, Lemma 2 says that $(G, Cost)$ is a disjoint-read length-cost CRWI digraph. Clearly the minimum cost of an FVS for G is $C \cdot \phi(G)$, and $C \cdot \phi(G) = C \cdot \phi(G')$ by Lemma 3. Therefore, the output of the reduction is $(G, Cost)$ and CK. ■

Given an NP-hard optimization problem, it is natural to ask whether the problem can be approximately solved by a polynomial-time algorithm. The worst-case approximation performance is typically measured by the worst-case ratio of the cost of the solution found by the algorithm to the optimum cost; see, for example, [13, §6.1]. The currently best known polynomial-time approximation algorithm for the min-cost FVS problem on general digraphs has ratio $O(\log n \log\log n)$ where n is the number of vertices in the input digraph; this is shown by Even *et al.* [12], building on work of Seymour [26]. An obvious question is whether this ratio can be improved, perhaps to a constant, by restricting G to CRWI digraphs. Unfortunately, the restriction to CRWI digraphs cannot help much, in the sense that an improvement in $r(n)$ for CRWI G would give a related improvement in $r(n)$ for general G. A modification to the proof of Theorem 1, again using Lemma 3, shows the following: If there is a polynomial-time approximation algorithm with ratio $r(n)$ for the min-cost FVS problem where the input $(G, Cost)$ is restricted to be a disjoint-read length-cost CRWI digraph, then there is a polynomial-time approximation algorithm with ratio $r'(n) = r(4n^2)$ for the min-cost FVS problem where $(G, Cost)$ is arbitrary. For example, if r is constant then r' is constant, and if $r(n) = O(\log n \log\log n)$ and r is sufficiently smooth then $r'(n) = O(r(n))$.

8 Complexity of Optimal In-Place Reconstructible Delta Encoding

The subject of the paper up to this point has been the problem of post-processing a given delta encoding of a version file $\mathcal{V}$ so that $\mathcal{V}$ can be reconstructed in-place from the reference file $\mathcal{R}$ using the modified delta encoding. A more general problem is to find an in-place reconstructible delta encoding of a given version file $\mathcal{V}$ in terms of a given reference file $\mathcal{R}$. Thus, this paper views the general problem as a two-step process, and concentrates on methods for and complexity of the second step.

Two-Step In-Place Delta Encoding
Input: A reference file $\mathcal{R}$ and a version file $\mathcal{V}$.

1. Using an existing delta encoding algorithm, find a delta encoding Δ of $\mathcal{V}$ in terms of $\mathcal{R}$.
2. Modify Δ by permuting commands and possibly changing some *copy* commands to *add* commands so that the modified delta encoding is in-place reconstructible.

A practical advantage of the two-step process is that we can utilize existing differencing algorithms to perform step 1. A potential disadvantage is the possibility that there is an efficient (in particular, a

polynomial-time) algorithm that finds an optimally-compact in-place reconstructible delta encoding for any input $\mathcal{V}$ and $\mathcal{R}$. Then the general problem would be made more difficult by breaking it into two steps as above, because solving the second step optimally is NP-hard. However, we show that this possibility does not occur: Finding an optimally-compact in-place reconstructible delta encoding is itself an NP-hard problem. For this result we define an in-place reconstructible delta encoding Δ to be one that contains no WR conflict. (As noted in Section 10, there is an alternate and more lenient definition, which we do not study in this paper.) It is interesting to compare the NP-hardness of minimum-cost in-place delta encoding with the fact that minimum-cost delta encoding (not necessarily in-place reconstructible) can be solved in polynomial time [27, 21, 23].

This NP-hardness result is proved using the following simple measure for the cost of a delta encoding. This measure simplifies the analysis while retaining the essence of the problem.

Simple Cost Measure: The cost of a *copy* command is 1, and the cost of an *add* command $\langle t, l \rangle$ is the length l of the added string.

BINARY IN-PLACE DELTA ENCODING
Instance: Two strings $\mathcal{R}$ and $\mathcal{V}$ of bits, and a $K \in \mathbf{N}^+$.
Question: Is there a delta encoding Δ of $\mathcal{V}$ in terms of $\mathcal{R}$ such that Δ contains no WR conflict and the simple cost of Δ is at most K?

Taking $\mathcal{R}$ and $\mathcal{V}$ to be strings of bits means that *copy* commands in Δ can copy any binary substrings from $\mathcal{R}$; in other words, the granularity of change is one bit. This makes our NP-completeness result stronger, as it easily implies NP-completeness of the problem for any larger (constant) granularity. The following theorem is proved in Appendix B.

Theorem 2 BINARY IN-PLACE DELTA ENCODING *is NP-complete.*

9 Conclusions

We have presented algorithms that modify delta files so that the encoded version may be reconstructed in the absence of scratch memory or storage space. Such an algorithm 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 encoding the data to be transmitted compactly. In-place reconstruction exchanges a small amount of compression in order to do so without scratch space.

Experimental results indicate that converting a delta file into an in-place reconstructible delta file has limited impact on compression, less than 4% in total with the majority of compression loss from encoding overheads rather than modifications to the delta file. We also find that for bottom line performance, keeping delta files small to reduce I/O matters more than execution time differences in cycles breaking heuristics, because in-place reconstruction is I/O bound. The algorithm to convert a delta file to an in-place reconstructible delta file requires less time than generating the delta file in the first place.

Our results also add to the theoretical understanding of in-place reconstruction. We have given a simple sufficient condition, the 1-or-1 edge property, for a digraph to be a CRWI digraph (for some delta file). Two problems of maximizing the compression of an in-place reconstructible delta file have been shown NP-hard: first, when the input is a delta file and the objective is to modify it to be in-place reconstructible; and second, when the input is a reference file and a version file and the objective is to find an in-place reconstructible delta file for them. The first result justifies our use of efficient, but not optimal, heuristics for cycle breaking.

In-place reconstructible delta file compression provides the benefits of delta compression for data distribution to an important class of applications – devices with limited storage and memory. In the current network computing environment, this technology decreases greatly the time to distribute software without increasing the development cost or complexity of the receiving devices. Delta compression provides Internet-scale file sharing with improved version management and update propagation, and in-place reconstruction delivers the technology to the resource constrained computers that need it most.

10 Future Directions

Detecting and breaking conflicts at a finer granularity can reduce lost compression when breaking cycles. In our current algorithms, we eliminate cycles by converting *copy* commands into *add* commands. However, typically only a portion of the offending *copy* command actually conflicts with another command; only the overlapping range of bytes. We propose, as a simple extension, to break a cycle by converting part of a *copy* command to an *add* command, eliminating the graph edge (rather than a whole vertex as we do today), and leaving the remaining portion of the *copy* command (and its vertex) in the graph. This extension does not fundamentally change any of our algorithms, only the cost function for cycle breaking.

As a more radical departure from our current model, we are exploring reconstructing delta files with bounded scratch space, as opposed to zero scratch space as with in-place reconstruction. This formulation, suggested by Martín Abadi, allows an algorithm to avoid *WR* conflicts by moving regions of the reference file into a fixed size buffer, which preserves reference file data after that region has been written. The technique avoids compression loss by resolving data conflicts without eliminating *copy* commands.

Reconstruction in bounded space is logical, as target devices often have a small amount of available space that can be used advantageously. However, in-place reconstruction is more generally applicable. For bounded space reconstruction, the target device must contain enough space to rebuild the file. Equivalently, an algorithm constructs a delta file for a specific space bound. Systems benefit from using the same delta file to update software on many devices. For example distributing an update calendar program to many PDAs. In such cases, in-place reconstruction offers a lowest common denominator solution in exchange for a little lost compression.

An even more radical departure follows from the fact that the absence of WR conflicts is a sufficient, but not necessary, condition for in-place reconstructibility. One could relax the definition of in-place reconstructibility to require only that application of the delta file to the reference file in-place produces the version file. For example, the reference file $\mathcal{R} = 0^l1^l2^l$ containing three "blocks" of length l can be converted in-place to the version file $\mathcal{V} = 1^l0^l1^l$ by copying the first two blocks 0^l1^l to the last two blocks, producing $0^l0^l1^l$; and then copying the last block 1^l to the first block, producing $\mathcal{V}$. These two *copy* commands conflict because the first *copy* writes to the third block and the second *copy* then reads from this block. It is easy to see that there is no way to convert $\mathcal{R}$ to $\mathcal{V}$ using two non-conflicting *copy* commands. This more lenient definition could yield smaller in-place reconstructible delta files when blocks of data are repeated in $\mathcal{V}$.

Although departing from our current model could yield smaller delta files, the message of this paper remains that the compression loss due to in-place reconstructibility is modest even within this simple model.

We also are developing algorithms that can perform peer-to-peer style delta compression [29] in an in-place fashion. This allows delta compression to be used between two versions of a file stored on separate machines and is often a more natural formulation, because it does not require a computer to maintain the original version of data to employ delta compression. This works well for file systems, most of which do not handle multiple versions.

Our ultimate goal is to use in-place algorithms as a basis for a data distribution system. The system will operate both in hierarchical (client/server) and peer-to-peer modes. It will also conform to Internet standards [17] and, therefore, work seamlessly with future versions of HTTP.

Appendix

A Proof of Lemma 2

We prove parts 1 and 2 together. For both parts we assume that $G = (V, E)$ has the 1-or-1 edge property and $\mathit{Cost}(v) \geq 2$ for all $v \in V$. We show how to choose the read intervals and write intervals such that $|R(v)| = |W(v)|$ for all $v \in V$, the write intervals are pairwise disjoint, and $R(v) \cap W(w) \neq \emptyset$ iff $\vec{vw} \in E$. If in addition $\mathit{indeg}(G) \leq 2$, then the chosen read intervals are pairwise disjoint. If in addition $\mathit{outdeg}(G) \leq 2$, then the choices also satisfy the length-cost condition $|R(v)| = \mathit{Cost}(v)$ for all $v \in V$.

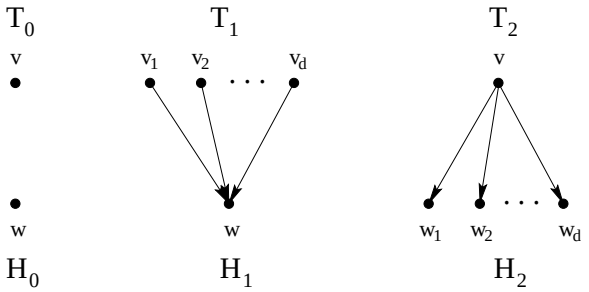
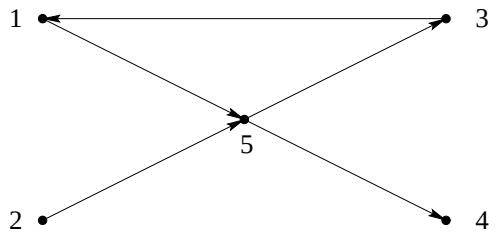


Figure 11: Examples of vertices in the sets H_j, T_j for $j = 0, 1, 2$. All edges directed out of $v, v_1, \ldots, v_d$ or into $w, w_1, \ldots, w_d$ are shown. Edges directed into $v, v_1, \ldots, v_d$ or out of $w, w_1, \ldots, w_d$ are not shown.



T_0 = { 4 } T_1 = { 1, 2, 3 } T_2 = { 5 }

H_0 = { 2 } H_1 = { 1, 5 } H_2 = { 3, 4 }

Figure 12: Example of the sets H_j, T_j for a particular digraph.

Let T_0 (resp., T_1, T_2) be the set of vertices $v \in V$ with $outdeg(v) = 0$ (resp., $outdeg(v) = 1$, $outdeg(v) \geq 2$). The three sets T_0, T_1, T_2 partition V; that is, they are pairwise disjoint and their union equals V. Let H_0 be the set of vertices w such that $indeg(w) = 0$. Let H_1 be the set of w such that: (i) $indeg(w) \geq 1$, and (ii) all v with $\vec{vw} \in E$ have $outdeg(v) = 1$ (*i.e.*, $v \in T_1$). Let H_2 be the set of w such that there exists a v with $\vec{vw} \in E$ and $outdeg(v) \geq 2$; that is, w is the head of some edge whose tail v belongs to T_2. Note that H_0, H_1, H_2 partition V. In Figure 11 the sets H_j, T_j for $j = 0, 1, 2$ are illustrated in general, and Figure 12 shows these sets for a particular digraph. The intervals are chosen by the following procedure. For $j = 0, 1, 2$, executions of step j choose read (resp., write) intervals for vertices in T_j (resp., H_j). Because $T_0 \cup T_1 \cup T_2 = H_0 \cup H_1 \cup H_2 = V$, the procedure chooses a read and write interval for each vertex. Because T_0, T_1, T_2 are pairwise disjoint and H_0, H_1, H_2 are pairwise disjoint, no interval is chosen at executions of two differently numbered steps. We show, during the description of the procedure, that no interval is chosen at two different executions of the same-numbered step. It follows that each vertex has its read interval and its write interval chosen exactly once. (The steps 0, 1, 2 are independent; in particular, it is not important that they are done in the order 0, 1, 2.) While describing the procedure, it's operation is illustrated for the graph in Figure 12, assuming that all vertices have *Cost* $= 2$. Because this graph has $indeg = outdeg = 2$, the chosen read intervals will be pairwise disjoint and the length-cost condition will be satisfied.

Interval Choosing Procedure

Set $k = 0$. The parameter k is increased after each execution of step 0, 1, or 2, in order that all intervals chosen during one execution of a step are disjoint from all intervals chosen during later executions of these steps. By an "execution of step j" we mean an execution of the body of a while statement in step j.

0. (a) While $T_0 \neq \emptyset$:
   Let $v \in T_0$ be arbitrary (in this case, $R(v)$ should not intersect any write interval); choose $R(v) = [k, k + \mathit{Cost}(v) - 1]$; remove v from T_0; and set $k \leftarrow k + \mathit{Cost}(v)$.

   (b) While $H_0 \neq \emptyset$:
   Let $w \in H_0$ be arbitrary (in this case, $W(w)$ should not intersect any read interval); choose $W(w) = [k, k + \mathit{Cost}(w) - 1]$; remove w from H_0; and set $k \leftarrow k + \mathit{Cost}(w)$.

Illustration. In Figure 12, $T_0 = \{4\}$ and $H_0 = \{2\}$. Step 0 is executed twice. First, step 0(a) sets $R(4) = [0, 1]$, $T_0 \leftarrow \emptyset$, and $k \leftarrow 2$. Then, step 0(b) sets $W(2) = [2, 3]$, $H_0 \leftarrow \emptyset$, and $k \leftarrow 4$.

1. While $T_1 \neq \emptyset$:

   (a) Let v_1 be an arbitrary vertex in T_1 and let w be the (unique) vertex such that $\overrightarrow{v_1 w} \in E$. If $\mathit{indeg}(w) = d \geq 2$, let $v_1, v_2, \ldots, v_d$ be the vertices such that $\overrightarrow{v_i w} \in E$ for $1 \leq i \leq d$. We claim that $v_i \in T_1$ for all $1 \leq i \leq d$: for $i = 1$ this is true by assumption; if $2 \leq i \leq d$ and $\mathit{outdeg}(v_i) \geq 2$, this would contradict the 1-or-1 edge property because $\overrightarrow{v_i w} \in E$, $\mathit{outdeg}(v_i) \geq 2$, and $\mathit{indeg}(w) \geq 2$. A consequence of the claim is that $w \in H_1$. (See Figure 11.)

   (b) Choose $R(v_1) = [l, r]$ and $W(w) = [l', r + 1]$ such that $\min\{l, l'\} = k$, $|\,[l, r]\,| = \mathit{Cost}(v_1)$, and $|\,[l', r + 1]\,| = \mathit{Cost}(w)$. Because $\mathit{Cost}(w) \geq 2$, we have $l' \leq r$; so $r \in R(v_1) \cap W(w)$ (which implies $R(v_1) \cap W(w) \neq \emptyset$). If $d \geq 2$, choose $R(v_i) = [r + 1, r_i]$ such that $|\,[r + 1, r_i]\,| = \mathit{Cost}(v_i)$ for $2 \leq i \leq d$. So $r + 1 \in R(v_i) \cap W(w)$ for $2 \leq i \leq d$. Note that if $\mathit{outdeg}(G) \leq 2$ then $d \leq 2$, and $R(v_1) \cap R(v_2) = \emptyset$ if $d = 2$. Because this step is the only one where more than one read interval is chosen at the same execution of a step, if $\mathit{outdeg}(G) \leq 2$ then the chosen read intervals are pairwise disjoint.

   (c) Remove $v_1, \ldots, v_d$ from T_1. Because $\overrightarrow{v w} \in E$ implies that $v = v_i$ for some $1 \leq i \leq d$, this ensures that none of $W(w), R(v_1), \ldots, R(v_d)$ are re-chosen at another execution of step 1. Set $k \leftarrow \max\{r + 1, r_2, r_3, \ldots, r_d\} + 1$.

Illustration. In Figure 12, $T_1 = \{1, 2, 3\}$. At the first execution of step 1, say that $v_1 = 1$ is chosen. This defines (recall that all costs are 2 in the example) $w = 5$, $d = 2$, $v_2 = 2$, $R(1) = [4, 5]$, $W(5) = [5, 6]$, $R(2) = [6, 7]$, $T_1 \leftarrow \{3\}$, and $k \leftarrow 8$. At the next execution of step 1, the only choice is $v_1 = 3$ which defines $w = 1$, $d = 1$, $R(3) = [8, 9]$, $W(1) = [9, 10]$, $T_1 \leftarrow \emptyset$, and $k \leftarrow 11$. (Because $d = 1$ in this execution, it would also work to set $R(3) = W(1) = [8, 9]$ and $k \leftarrow 10$, but for simplicity this special case is not included in the procedure.)

2. While $T_2 \neq \emptyset$:

   (a) Let v be an arbitrary vertex in T_2, let $d = \mathit{outdeg}(v)$ (so $d \geq 2$), and let $w_1, \ldots, w_d$ be the vertices such that $\overrightarrow{v w_i} \in E$ for $1 \leq i \leq d$. Note that $w_i \in H_2$ for all i, by definition of H_2. By the 1-or-1 edge property, $\mathit{indeg}(w_i) = 1$ for all i. (See Figure 11.)

   (b) If $\mathit{outdeg}(G) \leq 2$, then $d = 2$, and our choice of intervals must satisfy the length-cost property. In this case, choose $R(v) = [l, r]$, $W(w_1) = [l', r - 1]$, and $W(w_2) = [r, r_2]$ such that $\min\{l, l'\} = k$ and the intervals have the correct lengths according to the cost function. Note that $\mathit{Cost}(v) \geq 2$ implies that $l \leq r - 1$, and this in turn implies that $r - 1 \in R(v) \cap W(w_1)$. Also $r \in R(v) \cap W(w_2)$ by definition.

   If $\mathit{outdeg}(G) \geq 3$, then the length-cost property does not have to hold, so we choose $R(v) = [k, k + d - 1]$ and $W(w_i) = [k + i - 1, k + i - 1]$ for $1 \leq i \leq d$.

(c) Remove v from T_2. Because $indeg(w_i) = 1$ for all $1 \leq i \leq d$, it follows that none of $R(v), W(w_1), \ldots, W(w_d)$ are re-chosen at another execution of step 2. Set k to one plus the maximum right end-point of the intervals $R(v), W(w_1), \ldots, W(w_d)$.

Illustration. In Figure 12, $T_2 = \{5\}$. This defines $v = 5$, $d = 2$, $w_1 = 3$, $w_2 = 4$, $R(v) = [12, 13]$, $W(3) = [11, 12]$, $W(4) = [13, 14]$, and $T_2 \leftarrow \emptyset$. ■

B Proof of Theorem 2

In this proof, "cost" means "simple cost", and a "conflict-free" Δ is one containing no WR conflict. It suffices to give a polynomial-time reduction from FEEDBACK VERTEX SET to BINARY IN-PLACE DELTA ENCODING. Let G' and K' be an instance of FEEDBACK VERTEX SET. We describe binary strings $\mathcal{R}$ and $\mathcal{V}$ and an integer K such that $\phi(G') \leq K'$ iff there is a conflict-free delta encoding Δ of $\mathcal{V}$ in terms of $\mathcal{R}$ such that the cost of Δ is at most K.

First, using the transformation of Lemma 3, obtain G where G has the 1-or-1 edge property, $outdeg(G) \leq 2$, $indeg(G) \leq 2$, and $\phi(G) = \phi(G')$. Let $G = (V, E)$ and $V = \{1, 2, \ldots, n\}$. Let $l = \lceil \log n \rceil$. For each $v \in V$, define the binary string α_v as

$$\alpha_v = 10100b_100b_200b_300 \ldots b_l00$$

where $b_1b_2b_3 \ldots b_l$ is the l-bit binary representation of $v - 1$. Note that the length of α_v is $3l + 5$ for all v, and that $v \neq w$ implies $\alpha_v \neq \alpha_w$. Let $L = 3l + 6$, and define $Cost(v) = L$ for all $v \in V$. It follows from Lemma 2 that $(G, Cost)$ is a disjoint-read length-cost CRWI digraph. Because the interval-finding procedure in the proof of Lemma 2 runs in polynomial time, we can construct in polynomial time a RWIS $(\mathbf{R}, \mathbf{W})$, with $\mathbf{R} = \{R(1), \ldots, R(n)\}$ and $\mathbf{W} = \{W(1), \ldots, W(n)\}$, such that $G = graph(\mathbf{R}, \mathbf{W})$, $|R(v)| = |W(v)| = L$ for all $v \in V$, and the intervals of $\mathbf{R}$ are pairwise disjoint (the intervals of $\mathbf{W}$ are pairwise disjoint by definition). Moreover, because $indeg(G) \leq 2$ and $L \geq 4$, it is easy to see that we can make the read intervals be at least distance 3 apart, that is, if $i \in R(v)$, $j \in R(w)$, and $v \neq w$, then $|i - j| \geq 3$. (Referring to the interval-choosing procedure in the proof of Lemma 2, this can be done by incrementing k by an additional 2 after every execution of a step; and in executions of step 2 where $d = 2$ choosing $R(v_1) = [k, k + L - 1]$, $R(v_2) = [k + L + 2, k + 2L + 1]$, and $W(w) = [k + L - 1, k + 2L - 2]$. Note that $R(v_2) \cap W(w) \neq \emptyset$ because $L \geq 4$ implies $k + 2L - 2 \geq k + L + 2$.)

Let $\rho : \{1, \ldots, n\} \to \{1, \ldots, n\}$ be a permutation such that the intervals of $\mathbf{R}$ in left-to-right order (ordered as intervals) are $R(\rho(1)), R(\rho(2)), \ldots, R(\rho(n))$; thus, if $1 \leq j_1 < j_2 \leq n$, $i_1 \in R(\rho(j_1))$, and $i_2 \in R(\rho(j_2))$, then $i_1 < i_2$ (in fact, $i_1 \leq i_2 - 3$). Similarly, let σ be a permutation such that the intervals in $\mathbf{W}$ in left-to-right order are $W(\sigma(1)), W(\sigma(2)), \ldots, W(\sigma(n))$.

The binary strings $\mathcal{R}$ and $\mathcal{V}$ are of the form

$$\begin{aligned}
\mathcal{R} &= \overbrace{\alpha_1\,0\,\alpha_2\,0\,\ldots\,\alpha_n\,0}^{P_{\mathcal{R}}}\;0^*\;\alpha_{\rho(1)}\,1\,0\,0\,0^*\,\alpha_{\rho(2)}\,1\,0\,0\,0^*\;\ldots\;\alpha_{\rho(n-1)}\,1\,0\,0\,0^*\,\alpha_{\rho(n)}\,1 \\
\mathcal{V} &= 1^* \qquad \ldots \qquad 1^*\;\underbrace{\alpha_{\sigma(1)}\,1\,1^*\,\alpha_{\sigma(2)}\,1\,1^*\;\ldots\;\alpha_{\sigma(n-1)}\,1\,1^*\,\alpha_{\sigma(n)}\,1}_{S_{\mathcal{V}}}
\end{aligned}$$

where 0^* (resp., 1^*) denotes a string of zero or more 0's (resp., 1's), and where these "rubber-length" strings are adjusted so that: (i) the prefix $P_{\mathcal{R}}$ of $\mathcal{R}$ does not overlap the suffix $S_{\mathcal{V}}$ of $\mathcal{V}$, and (ii) for all $v, w \in V$, the substring $\alpha_v 1$ of $\mathcal{R}$ overlaps the substring $\alpha_w 1$ of $\mathcal{V}$ iff $\vec{vw}$ is an edge of G. That (ii) can be accomplished follows from the facts $G = graph(\mathbf{R}, \mathbf{W})$, all read and write intervals have length $L = 3l + 6$ (which equals the length of $\alpha_v 1$ for all v), and the read intervals are at least distance 3 apart so we can insert at least two zeroes between $\alpha_{\rho(i)}1$ and $\alpha_{\rho(i+1)}1$ for $1 \leq i < n$.

Three properties this $\mathcal{R}$ and $\mathcal{V}$ will be used:

(P1) $\mathcal{R}$ contains no occurrence of the substring 11;

(P2) for each $v \in V$, the string $\alpha_v 1$ appears exactly once as a substring of $\mathcal{R}$;

(P3) for each $v \in V$ with $v \neq \sigma(n)$, the string $\alpha_v 1$ always appears in $\mathcal{V}$ in the context $\ldots 1\alpha_v 11 \ldots$.

Property P1 is obvious by inspection. Property P2 follows from the facts: (i) 101 appears as a substring of $\mathcal{R}$ only as the first three symbols of α_w for each $w \in V$; and (ii) if $v \neq w$ then $\alpha_v \neq \alpha_w$. Property P3 follows because, for each $w \in V$, the string $\alpha_w 1$ both begins and ends with 1, and there are only 1's between $\alpha_{\sigma(i)}1$ and $\alpha_{\sigma(i+1)}$ for $1 \leq i < n$.

Let $L_{\mathcal{V}}$ denote the length of $\mathcal{V}$, and define $K = L_{\mathcal{V}} - nL + n + K'$. We show that

$$\phi(G) \leq K' \Leftrightarrow \text{there is a conflict-free delta encoding } \Delta \text{ of } \mathcal{V} \text{ such that the cost of } \Delta \text{ is at most } K.$$

($\Rightarrow$) Let $\phi(G) \leq K'$ and let S be a FVS for G with $|S| \leq K'$. We first describe an encoding Δ' of $\mathcal{V}$ that is not necessarily conflict-free. Each substring represented by 1^* is encoded by an *add* command; the total cost of these *add* commands is $L_{\mathcal{V}} - nL$. If $v \in V - S$, then $\alpha_v 1$ is encoded by a *copy* of $\alpha_{\rho(i)}1$ in $\mathcal{R}$, where i is such that $\rho(i) = v$; the total cost of these *copy* commands is $|V - S| = n - |S|$. If $v \in S$, then $\alpha_v 1$ is encoded by a *copy* of α_v from $P_{\mathcal{R}}$ followed by an *add* of "1"; the total cost of these commands is $2|S|$. Therefore, the total cost of Δ' is $L_{\mathcal{V}} - nL + n + |S| \leq L_{\mathcal{V}} - nL + n + K' = K$. For each $v \in S$, the read interval of the *copy* command that copies α_v from $P_{\mathcal{R}}$ does not intersect the write interval of any *copy* command in Δ'. Therefore, the CRWI digraph of Δ' is a subgraph of the graph obtained from G by removing, for each $v \in S$, all edges directed out of v. Because S is an FVS for G, the CRWI digraph of Δ' is acyclic. Therefore, a conflict-free delta encoding Δ of the same cost can be obtained by permuting the *copy* commands of Δ' and moving all *add* commands to the end.

($\Leftarrow$) Let Δ be a conflict-free delta encoding of $\mathcal{V}$ having cost at most $K = L_{\mathcal{V}} - nL + n + K'$. By properties P1 and P3, it follows that no *copy* command in Δ can encode a prefix (resp., suffix) of a substring $\alpha_v 1$ together with at least one of the 1's preceding it (resp., following it). Therefore, using property P1 again, the commands in Δ that encode substrings denoted 1^* must have total cost equal to the total length of these substrings, that is, cost $L_{\mathcal{V}} - nL$. The remaining commands can be partitioned into sets $C_1, C_2, \ldots, C_n$ such that the commands in C_v encode $\alpha_v 1$ for each $v \in V$. Let S be the set of $v \in V$ such that C_v contains at least two commands. We first bound $|S|$ and then argue that S is a FVS for G. By definition of S, the cost of Δ is at least $L_{\mathcal{V}} - nL + |V - S| + 2|S|$. Because the cost of Δ is at most $L_{\mathcal{V}} - nL + n + K'$ by assumption, we have $|S| \leq K'$. To show that S is a FVS, assume for contradiction that there is a cycle in G that passes only through vertices in $V - S$. If $v \in V - S$ then C_v contains one command γ_v, so γ_v must be a *copy* command that encodes $\alpha_v 1$. By property P2, the *copy* command γ_v must be to copy the substring $\alpha_v 1$ from the unique location where it occurs in $\mathcal{R}$ as $\alpha_{\rho(i)}1$ where i is such that $v = \rho(i)$. The strings $\mathcal{R}$ and $\mathcal{V}$ have been constructed such that, if $\overrightarrow{vw}$ is an edge of G (in particular, if $\overrightarrow{vw}$ is an edge on the assumed cycle through vertices in $V - S$), then the substring $\alpha_v 1$ of $\mathcal{R}$ overlaps the substring $\alpha_w 1$ of $\mathcal{V}$. So the existence of this cycle contradicts the assumption that Δ is conflict-free. ■

References

[1] The free network project – rewiring the Internet. Technical Report http://freenet.sourceforge.net/, 2001.

[2] The gnutella protocol specification. Technical Report http://www.gnutelladev.com/protocol/gnutella-protocol.html, 2001.

[3] M. Ajtai, R. Burns, R. Fagin, D. D. E. Long, and L. Stockmeyer. Compactly encoding unstructured input with differential compression. `www.almaden.ibm.com/cs/people/stock/diff7.ps`, IBM Research Report RJ 10187, April 2000 (revised Aug. 2001).

[4] G. Banga, F. Douglis, and M. Rabinovich. Optimistic deltas for WWW latency reduction. In *Proceedings of the 1997 Usenix Technical Conference*, pages 289–303, 1997.

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

[6] R. 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.

[7] 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.

[8] M. Chan and T. Woo. Cache-based compaction: A new technique for optimizing web transfer. In *Proceedings of the IEEE Infocom '99 Conference, New York, NY*, March 1999.

[9] S. S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, May 1997.

[10] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

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

[12] G. Even, J. Naor, B. Schieber, and M. Sudan. Approximating minimum feedback sets and multi-cuts in directed graphs. *Algorithmica*, 20:151–174, 1998.

[13] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, 1979.

[14] J. J. Hunt, K.-P. Vo, and W. F. Tichy. An empirical study of delta algorithms. In *Proceedings of the 6th Workshop on Software Configuration Management*, March 1996.

[15] J. J. Hunt, K.-P. Vo, and W. F. Tichy. Delta algorithms: An empirical analysis. *ACM Transactions on Software Engineering and Methodology*, 7(2):192–214, 1998.

[16] 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.

[17] D. G. Korn and K.-P. Vo. The VCDIFF generic differencing and compression format. Technical Report Internet-Draft draft-vo-vcdiff-00, Internet Engineering Task Force (IETF), 1999.

[18] S. Kurtz. Reducing the space requirements of suffix trees. *Software – Practice and Experience*, 29(13):1149–1171, 1999.

[19] J. P. MacDonald, P. N. Hilfinger, and L. Semenzato. PRCS: The project revision control system. In *Proceedings System Configuration Management*, 1998.

[20] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2), April 1978.

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

[22] 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.

[23] 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.

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

[25] 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.

[26] P. D. Seymour. Packing directed circuits fractionally. *Combinatorica*, 15:281–288, 1995.

[27] W. F. Tichy. The string-to-string correction problem with block move. *ACM Trans. Computer Systems*, 2(4):309–321, November 1984.

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

[29] A. Tridgell and P. Mackerras. The RSync algorithm. Technical Report TR-CS-96-05, The Australian National University, 1996.

[30] P. Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.