In-Place Reconstruction of Version Differences

Randal Burns
Department of Computer Science
Johns Hopkins University
randal@cs.jhu.edu

Larry Stockmeyer
Department of Computer Science
IBM Almaden Research Center
stock@acm.org

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

Technical Report HSSL-2002-01
Hopkins Storage Systems Lab
Department of Computer Science
Johns Hopkins University

Abstract

In-place reconstruction of differenced data allows information on devices with limited storage capacity to be updated efficiently over low-bandwidth channels. Differencing encodes a version of data compactly as a set of changes from a previous version. Transmitting updates to data as a version difference saves both time and bandwidth. In-place reconstruction rebuilds the new version of the data in the storage or memory the current version occupies – no scratch space is needed for a second version. By combining these technologies, we support highly-mobile applications on space-constrained hardware.

We present an algorithm that modifies a differentially encoded version to be in-place reconstructible. The algorithm trades a small amount of compression to achieve this property. Our treatment includes experimental results that show our implementation to be efficient in space and time and verify that compression losses are small. Also, we give results on the computational complexity of performing this modification while minimizing lost compression.

keywords: differencing, differential compression, version management, data distribution, in-place reconstruction, mobile computing

1 Introduction

We develop a system for data distribution and version management to be used in highly-mobile and resource-limited computers operating on low-bandwidth networks. The system combines differencing with a technology called *in-place reconstruction*. Differencing encodes a file compactly as a set of changes from a previous version. The system sends the difference encoding to a target computer in order to update the file, saving bandwidth and transfer time when compared with transmitting the whole file. In-place reconstruction updates the file in the memory or storage space the current version occupies. In-place reconstruction brings the benefits of differencing to the computers that need it the most – resource-constrained devices such as wireless handhelds and cellular phones.

Differencing has been widely used to reduce latency and lower bandwidth requirements in distributed systems. The original applications of differencing focused on reducing the storage required to maintain sequences of versions. Examples include source code control systems [21, 25, 16], editors [9], and databases [22]. In the last decade, researchers have realized that these algorithms compress data quickly and can be used to reduce bandwidth requirements and transfer time for applications that exchange data across networks. Examples include backup and restore [4], database consistency [6], and Internet protocols [2, 19, 5].

For completeness, we often group delta compression [10, 11, 5] with differencing [1, 16]. Delta compression is a generalization of differencing and data compression [27], in that a version of a file may be compressed with respect to matching strings from within the file being encoded, as well as from the other version. Although the results of this paper concern differential compression, our methods apply to delta encoding as well.

To date, differencing has not been employed effectively for resource-constrained mobile and wireless devices. While the problem space is ideal, it has not been used because reconstructing a differential encoding requires storage space (disk or memory) to manifest a new version of data while keeping the old version as a reference. This problem is particularly acute for mass-produced devices that use expensive non-volatile memories, such as personal digital assistants, wireless handhelds, and cellular phones. For these devices, it is important to keep manufacturing costs low. Therefore, it is not viable to add storage to a device solely for the purpose of differencing.

In-place reconstruction makes differential compression available to resource-constrained devices on any network; mobile and wireless networks are the most natural and interesting application. In-place reconstruction allows a version to be updated by a differential encoding in the memory or storage that it currently occupies; reconstruction does not need additional scratch space for a second copy. An in-place reconstructible differential encoding is a permutation and modification of the original encoding. This conversion comes with a small compression penalty. In-place reconstruction brings the latency and bandwidth benefits of differencing to the space-constrained, mass-produced devices that need them the most. The combination of differencing and in-place reconstruction keeps the cost of manufacturing mobile devices low, by reducing the demand on networking and storage hardware.

For one example application, we choose updating/patching the operating system of phones in a cellular network. Currently, the software and firmware in cellular phones remains the same over the life of the phone, or at least until the customer brings a phone in for service. Suppose that the authentication mechanism in the phone was compromised – perhaps the crypto was broken [13] or more likely keys were revealed [23]. In either case, updating software becomes essential for the correct operation of system. In particular, without trustworthy authentication, billing cannot be performed reliably. Using in-place reconstruction, the system patches the software quickly over the cellular network. The update degrades performance minimally by making the update size as small as possible. This example fits our system model well. Phones are mass-produced and, therefore, resource-constrained in order to keep manufacturing costs low. Also, cellular networks are low-bandwidth and cellular devices compete heavily for bandwidth. In-place reconstruction makes these devices manageable over networks, instead of immutable.

For another example, we choose a distributed inventory management system based on mobile-handheld devices. 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 runs a consistency protocol. In-place reconstruction allows the devices to keep their copies of data consistent using differencing without requiring scratch space, thereby increasing the cache utilization at target devices. We observe that in-place reconstruction applies to both structured data (databases) and unstructured data (files), because they manipulate a differential encoding, as opposed to the original data. Algorithms for differencing structured data [6] employ encodings that are suitable for in-place techniques.

Any application that has multiple resource-constrained computers sharing data interactively will want to use this technology and, in particular, applications that involve computer-human workflows using cellular or radio-frequency devices. Examples include security and law enforcement, property management, airport services, health care, and shipping/delivery.

1.1 Differencing and In-Place Reconstruction

We modify a differentially encoded file so that it is suitable for reconstructing the new version of the file in-place. A difference 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 difference 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 difference file in-place, the resulting output would often be corrupt. This occurs when the 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 been altered and the algorithm rebuilds an incorrect file.

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

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

Experiments also reveal an interesting property of these algorithms not expressed by 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 encoding. 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.

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 difference “from scratch”, *i.e.* 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, an optimally-compact difference file can be found in polynomial time [24, 18, 20].

2 Related Work

Encoding versions compactly by detecting altered regions of data is a well known problem. The first applications of differential compression found changed lines in text data for analyzing the recent modifications to files [8]. Considering data as lines of text fails to encode a minimum sized difference, 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 [24] – detecting maximally matching regions of a file at an arbitrarily fine granularity without alignment. However, differencing continued to rely on the alignment of data, as in database records [22], and the grouping of data into block or line granules, as in source code control systems [21, 25], to simplify the combinatorial task of finding the common and different strings between versions.

Efforts to generalize differencing 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 [18] or the greedy method [24, 20, 16] and performed this task using time quadratic in the length of the input files.

Differential compression algorithms were improved to run in linear time and linear space. Algorithms with these properties have been derived from suffix trees [26, 17, 15]. Like algorithms based on greedy methods and dynamic programming, these algorithms generate optimally compact encodings.

Delta compression is a more general form of differencing. It includes the concept of finding matching data within the file being encoded as well as comparing that file to other similar files [10, 11, 5]. Delta compression runs in linear time. Related to delta compression is a coding technique that unifies differential and general compression [14].

Recent advances produced differencing algorithms that run in linear time and constant space [1]. These algorithms trade a small amount of compression in order to improve performance.

Any of the linear run-time algorithms allow differencing to scale to large inputs without known structure and permit the application of differential compression to data management systems. These include binary source code control [16] and backup and restore [4].

Applications distributing HTTP objects using delta compression have emerged [2, 19, 5]. They permit 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.

3 Encoding Difference Files

Differencing algorithms encode the changes between two file versions compactly by finding strings common to both versions. We term the first file a *version file* that contains the data to be encoded and the second 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 difference that encodes this version. This encoding consists of an ordered sequence of *copy* commands and *add* commands.

An *add* command is an ordered pair, $\langle t, l \rangle$, where t (to) encodes the string offset in the file version and l (length) encodes the length of the string. 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

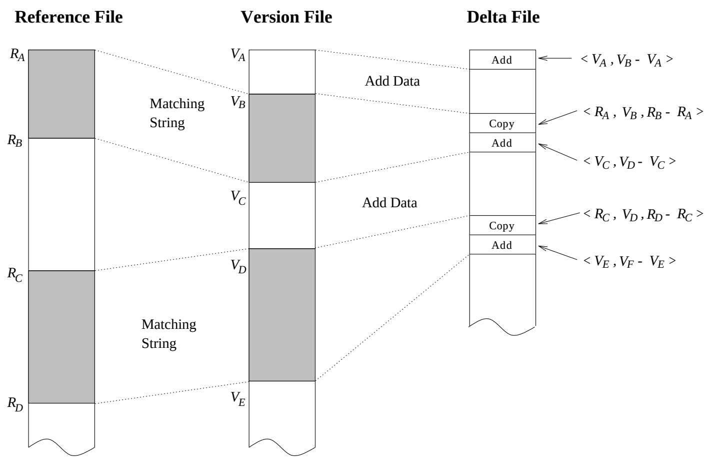


Figure 1: Encoding difference files. Common strings are encoded as *copy* commands $\langle f, t, l \rangle$ and new strings in the new file are encoded as *add* commands $\langle t, l \rangle$ followed by the length l string of added data.

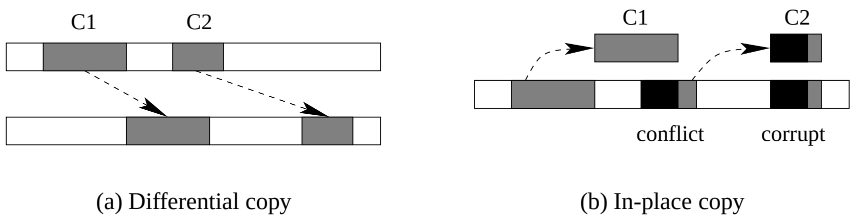


Figure 2: Data conflict and corruption when performing copy command C1 before C2.

reference file to the interval $[t, t + l - 1]$ in the version file.

In the presence of the reference file, a difference 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 Algorithm

An in-place modification algorithm changes an existing difference file into a difference file that reconstructs correctly a new file version in the space the current version occupies. At a high level, our technique examines the input difference file to find *copy* commands that conflict; in which one command reads data from the write interval (file address range to which the command writes data) of the other *copy* command. The algorithm represents potential data conflicts in a digraph. It topologically sorts the digraph to produce an ordering on *copy* commands that reduces conflicts. It eliminates the remaining conflicts by converting *copy* commands to *add* commands. The algorithm outputs the permuted and converted commands as an in-place reconstructible difference. Actually, as described in more detail below, the algorithm performs permutation and conversion of commands concurrently.

4.1 Conflict Detection

The algorithm orders commands that attempt to read a region to which another command writes. For this, we adopt the term *write before read* (*WR*) conflict [3]. For *copy* commands $\langle f_i, t_i, l_i \rangle$ and $\langle f_j, t_j, l_j \rangle$, with $i < j$, 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 (Equation 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 were the file 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 from adding data by placing *add* commands at the end of an encoding. In this way, the algorithm completes all reads from *copy* commands before executing the first *add* command.

Additionally, we define *WR* conflicts so that a *copy* command cannot conflict with itself, even though 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 a 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.

A difference file suitable for in-place reconstruction 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 $\overrightarrow{uv}$ then vertex u precedes vertex v in topological order.

Our technique constructs a digraph so that each *copy* command in the difference file has a corresponding vertex in the digraph. On this set of vertices, we construct an edge relation with a directed edge $\overrightarrow{uv}$ from vertex u to vertex v when *copy* command u's read interval intersects *copy* command v's write interval. Edge $\overrightarrow{uv}$ indicates that by performing command u before command v, the difference file avoids a WR conflict. We call a digraph obtained from a difference file in this way a *conflicting read-write interval* (*CRWI*) digraph. A topologically sorted version of this graph obeys 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*. A depth-first search implementation of topological sort [7] is modified to detect cycles. Upon detecting a cycle, our modified sort breaks the cycle by removing a vertex. The sort outputs a digraph containing a subset of all vertices in topological order and a set of vertices that were removed. The algorithm re-encodes the data contained in the *copy* commands of the removed vertices as *add* commands in the output.

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. Replacing a *copy* command ($\langle f, t, l\rangle$) with an *add* command ($\langle t, l\rangle$) increases the encoding size by $l - \|f\|$, where $\|f\|$ is 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 turning a digraph into an acyclic digraph by deleting vertices, an in-place conversion algorithm could minimize 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 [12] to be NP-hard for general digraphs. In Section 8, 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. In our implementation, 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 difference files into in-place reconstructible difference files takes the following steps to find and eliminate *WR* conflicts between a reference file and a version file.

Algorithm

1. Given an input difference 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: $i < j \iff t_i < t_j$. 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:

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

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 policy and remove it. Replace the data encoded in its *copy* command with an equivalent *add* command, which is put into set A.
5. Output the remaining *copy* commands to the difference file in topologically sorted order.
6. Output all *add* commands in set A to the difference file.

The resulting difference 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 encoding.

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 DFS algorithm outputs the un-removed copy commands in reverse topologically sorted order. A topological order is achieved by reversing the output of the DFS algorithm. 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 $\textsc{Visit}(u)$. The procedure $\textsc{Visit}(u)$ marks u as *on-stack*, pushes u on the stack, and examines each vertex w for which there is an edge $\overrightarrow{uw}$ 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 $\textsc{Visit}(w)$ 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 $\textsc{Visit}(p)$ 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 difference file; call this vertex r. Vertices r through u 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 $\textsc{Visit}(p)$ continues as though r were marked *finished*. Recall that we are describing an execution of $\textsc{Visit}(u)$ by examining all w such that there is an edge $\overrightarrow{uw}$. 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 difference 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 difference 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 difference 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 difference 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 $O(n + \min(L_{\mathcal{V}}, n^2))$.

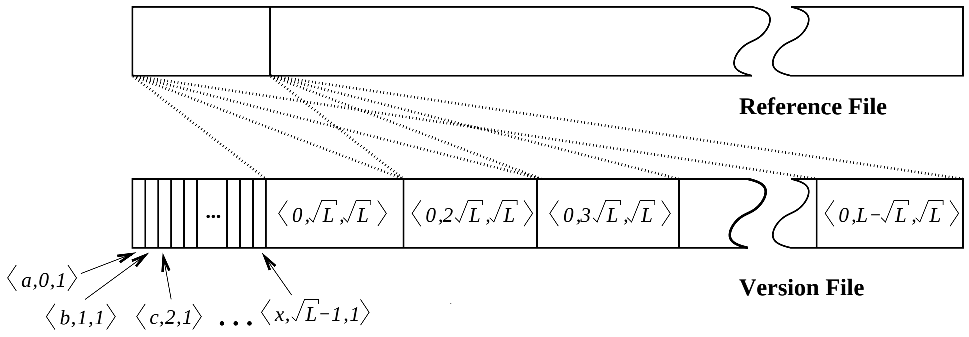

Reference File
Version File
$\langle 0,\sqrt{L},\sqrt{L} \rangle$
$\langle 0,2\sqrt{L},\sqrt{L} \rangle$
$\langle 0,3\sqrt{L},\sqrt{L} \rangle$
$\langle 0,L-\sqrt{L},\sqrt{L} \rangle$
$\langle a,0,1 \rangle$
$\langle b,1,1 \rangle$
$\langle c,2,1 \rangle$
...
$\langle x,\sqrt{L}-1,1 \rangle$

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 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 difference 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, $b_2, b_3, ..., b_{\sqrt{L}}$, 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 difference 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 difference 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 difference file that encodes a version file $\mathcal{V}$ of length $L_{\mathcal{V}}$, the number of edges in the digraph representing potential* WR *conflicts is 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 encoding, the sum of the lengths of all read intervals is less than or equal to $L_{\mathcal{V}}$. 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 difference file that encodes $\mathcal{V}$ is less than or equal to $L_{\mathcal{V}}$. ■

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

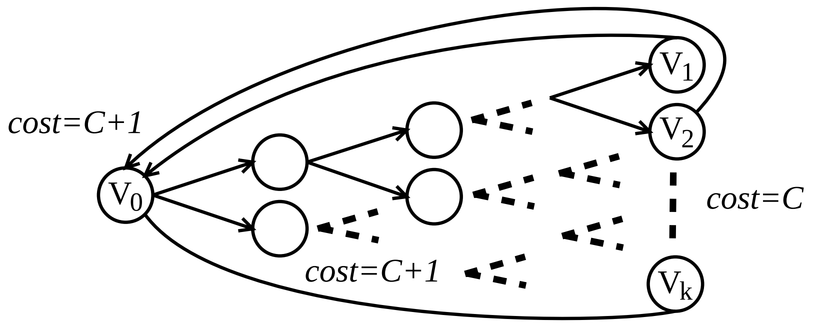


Figure 4: A CRWI digraph constructed from a binary tree by adding a directed edge from each leaf to the root vertex (only some paths shown). Each leaf has cost C and each other vertex has cost $C + 1$. The local-minimum cycle breaking policy performs poorly on this CRWI digraph, removing each leaf vertex,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

Bounding the number of edges in CRWI digraphs verifies 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; Theorem 1 in Section 7 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 difference 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 difference file generated by the local-minimum solution is $(k - 1)c + kC$, the size of the optimal difference file is $2(k - 1)c + C + 1$, and the ratio of these two sizes approaches $1/2 + C/2c$ 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 difference 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 in-place reconstruction is used for distributing data to mobile and resource-limited devices, we extracted a large body of experimental data that consists of versions of software intended for handhelds and personal digital assistants. Files include applications, boot loaders, and operating system components. In-place differencing was measured against these data with the goals of:

- determining the compression loss due to making difference 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 differencing algorithms; and,
- characterizing the graphs created by the algorithm.

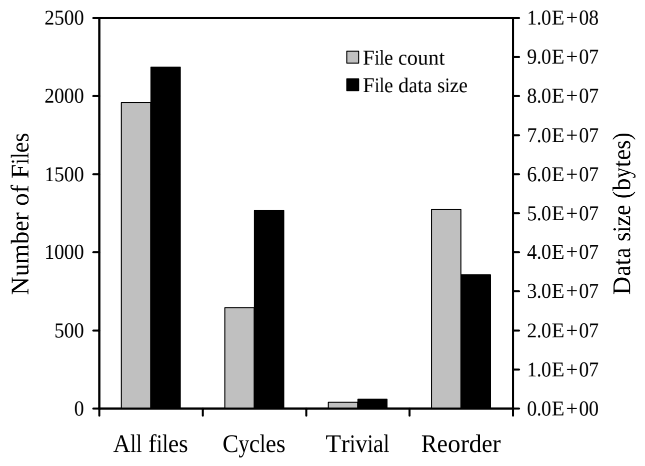


Figure 5: File counts and data size.

In all cases, we obtained the original difference files using the correcting 1.5-pass differential compression algorithm [1].

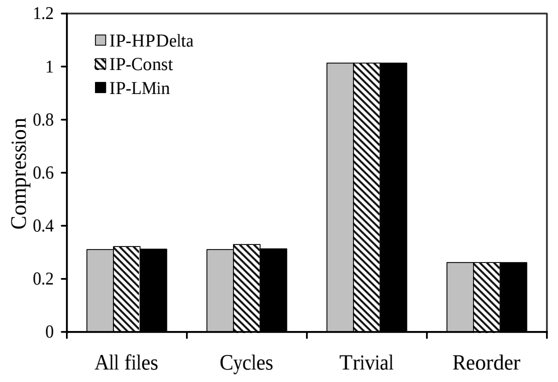
The experimental data we collected and employed are characteristic of the intended application. Because our interest lies in distributing files to resource-limited devices, we collected versions of open-source software intended for the Compaq iPAQ handheld device, a personal digital assistant that can run versions of the Linux operating system. Data were obtained in April 2002 from www.handhelds.org, a Web site designed to facilitate the "creation of open source software for use on handheld and wearable computers." To collect data, we downloaded the software archive and ran scripts that search the archive for multiple versions of the same files. The original and processed data are available from the Hopkins Storage Systems lab at http://hssl.cs.jhu.edu/ipdata/. All experimental data are files that are distributed to handheld devices: boot loaders, applications, flash updates, and their associated data files. We did not include source code or other data not intended for distribution to handhelds.

We categorize the difference files in our experiments into 3 groups that describe what operations were required to make files in-place reconstructible. Experiments were conducted on 1959 files totaling more that 87.4 Megabytes – an average file size of approximately 44 kilobytes. Of these files (Figure 5), 33% of the files contained cycles that needed to be broken. 65% did not have cycles, but needed to have *copy* commands reordered. The remaining 2% 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 difference.

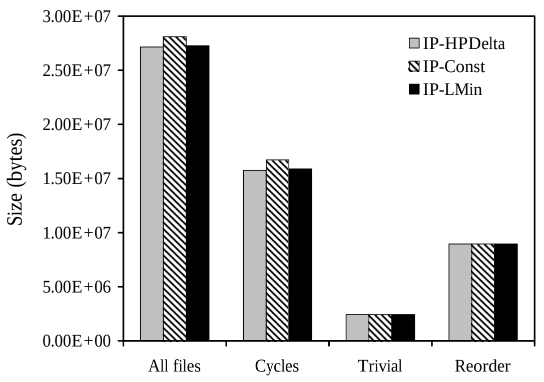
The amount of data in files is distributed differently across the three categories than are the file counts. Files with cycles contain over 58.0% (50.7 MB) of data with an average file size of 78 KB. Files that need copy commands reordered hold 39.3% (34.3 MB) of data, with an average file size of 27 KB. Trivially in-place reconstructible files occupy 2.7% (2.4 MB) of data with an average file size of 60 KB.

The distribution of files and data across the three categories confirms that efficient algorithms for cycle breaking and command reordering are needed to deliver differentially compressed data in-place. While most difference 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 compression (size of difference files as a fraction of the original file size) and Figure 6(b) shows the total size of the difference files. For each category and for all files, we report data for three algorithms, all of which are derived from the correcting 1.5-pass differencing algorithm (HPDelta) [1]. These algorithms are: the correcting 1.5-pass differencing 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 mod-



(a) Delta compression



(b) Delta size

Figure 6: Compression performance showing the compression achieved and the total number of bytes of compressed data for each class of files.

ification algorithm using the constant-time cycle breaking policy (IP-Const). The HPDelta algorithm is a linear time, constant space algorithm for generating differentially compressed files.

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, differencing algorithms, such as HPDelta, reconstruct data in write order and do not 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, allowing our algorithm to reorder commands. This extra field in each code-word introduces a per-command overhead in a difference file. The amount of compression loss varies, depending upon the number of commands and the original size of the difference file. Overhead in these experiments ran to more than 4.4% – which corresponds to output delta files that are 16% larger than with HPDelta. The codewords used in these experiments are not well tuned for in-place reconstruction, spending 4 bytes per codeword to describe a write offset. In the future, in-place differencing will require the careful codeword design that has been done for delta compression [14] to minimize these losses. For now, our experiments focus on compression loss from cycle breaking, *i.e.* compression loss attributable to in-place algorithms.

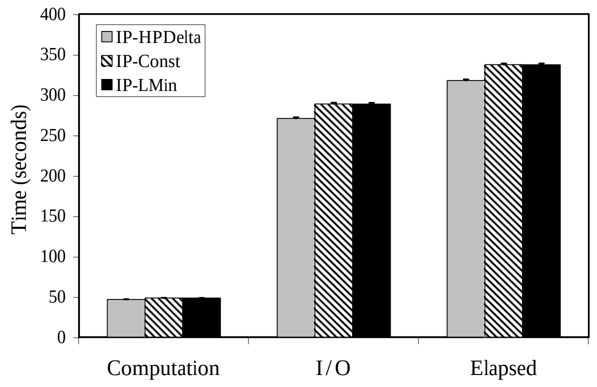
From the IP-HPDelta algorithm, we derive the IP-Const and IP-LMin algorithms. They run the IP-HPDelta algorithm to generate a difference file and then permute and modify the commands according to our techniques. The IP-Const algorithm implements the constant-time policy and the IP-LMin algorithm implements the local-minimum policy.

Experimental results describe the amount of compression lost due to in-place reconstruction. Over all files, IP-HPDelta compresses data to 31.1% their orginal size (Figure 6(a)). This number does not include data compression, which can be performed after the difference is taken. Compared to IP-HPDelta, IP-Const output is 3.6% larger, 28.10 MB as compared to 27.14 MB. The loss is attributed to breaking cycles. In contrast, IP-LMin generates output only 0.5% larger, 27.27 MB versus 27.14. The local-minimum policy performs excellently in practice – compression losses are one seventh that of the constant-time policy.

Because files with cycles contain the majority of the data (Figure 6(b)), the results for files with cycles dominate the results for all files. In reorder and trivially in-place difference files, no cycles are present and

	Computation		I/O		Elapsed	
	μ	0.95%	μ	0.95%	μ	0.95%
IP-HPDelta	46.71	0.20	270.71	1.38	317.64	1.41
IP-Const	48.42	0.21	288.70	1.43	337.30	1.44
IP-LMin	48.40	0.20	288.58	1.44	337.26	1.44

(a) Data with 0.95% confidence intervals



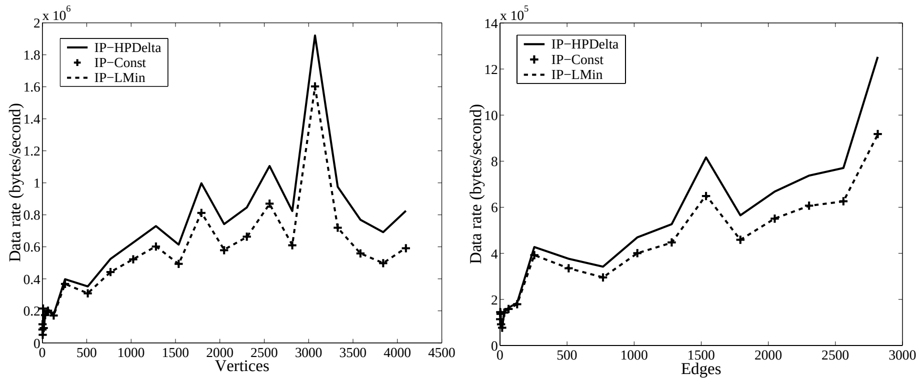
(b) Chart with 0.95% confidence intervals

Figure 7: Run-time results

no compression is lost. The class of files that are trivially in-place are incompressible using differencing. This class is dominated by few large files with little similarity between versions.

In-place algorithms incur execution time overheads when performing additional I/O and when permuting the commands in a difference file. An in-place algorithm generates a difference file and then modifies the file to have the in-place property. In-place algorithms create an intermediate file that contains the output of the differential compression algorithm. This intermediate output serves as the input for the algorithm that modifies/permutes commands. We present execution-time results in Figure 7 for both in-place algorithms – IP-Const and IP-LMin. Figure 7(b) includes 95% confidence intervals, which are barely discernible. 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 6%, *i.e.* the total run takes 20 seconds longer. Almost all of this overhead comes from additional I/O. We conclude that tasks for in-place reconstruction are small when compared with the effort of compressing data – the algorithmic tasks take only 2 seconds of additional time over the whole experiment. Despite inferior worst-case run-time bounds, the local-minimum policy performs nearly identically t0 (and marginally better than) the constant-time policy in practice.

Examining run-time results in more detail continues to show that IP-LMin tracks the performance of IP-Const, even for the largest and most complex inputs. In Figure 8, we see how run-time performance varies with the size of the graph the algorithm creates (number of edges and vertices); these plots measure data rate – file size (bytes) divided by run time (seconds). Graph size is the complexity measure for which IP-Const and IP-LMin should vary, but no such variance can be seen. Results show that in-place conversion algorithms are I/O bound, as are differencing algorithms [1]. Reducing computational effort when breaking cycles benefits an algorithm very little, as computation is a small fraction of total performance. Whereas minimizing the size of the output benefits an algorithm more, as I/O dictates overall performance.



(a) Vertices (b) Edges

Figure 8: Run-time results

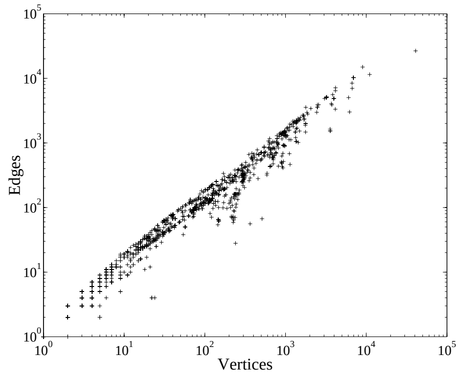
In Figure 9, we look at some statistical measures of graphs constructed when creating in-place difference files. While graphs can be quite large, a maximum of 26,626 vertices and 40,950 edges, the number of edges scales linearly with the number of vertices and less than linearly with input file size. 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. To illustrate, consider two pairs of versions as inputs to the IP-LMin algorithm in which one pair of versions generates a graph that contains twice the edges of the other. Based on Figure 9, 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.

The linear scaling of edges with vertices and file size matches our intuition about the nature of differentially compressed data. Differencing 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, which would result in dense edge relations. Additionally, differencing algorithms localize matching between files, correlating or synchronizing regions of file data [1]. All of these factors result in the linear scaling that we observe.

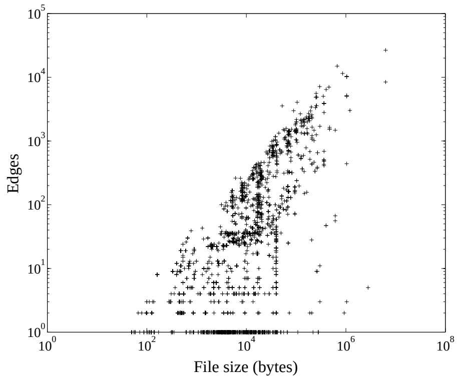
6 Generalization to In-Place Delta Compression

As mentioned in the Introduction, delta compression permits data to be copied from the version file, as well as from the reference file. Parts of the version file that have already been materialized during the reconstruction may be copied to other parts of the version file. Although in-place delta compression is not a subject of this paper, we note that the conversion of an arbitrary delta encoding to an in-place reconstructible delta encoding fits within our framework. We assume that the input delta encoding is designed to materialize the version file in space that is separate from the space occupied by the reference file. Thus, the *copy* commands can be partitioned into *copy-from-R* commands that read from the reference file and *copy-from-V* commands that read from the version file. For in-place reconstruction, as before, no part of the read interval of a *copy-from-R* command may be overwritten before the command is performed. But for a *copy-from-V* command, all of its read interval must be overwritten with that part of the version file before the command is performed.

An algorithm that converts an arbitrary delta encoding to an in-place reconstructible delta encoding



(a) Edges versus Vertices



(b) Edges versus File Size

Figure 9: Edges in difference files that contain cycles.

proceeds as follows. First apply the algorithm of Section 4 to the *copy-from-R* commands and the *add* commands in the input delta encoding. The output is a sequence of *copy-from-R* commands followed by *add* commands (including the *add* commands that were created by replacing a *copy-from-R* command by an equivalent *add* command). By the correctness of our algorithm, when this command sequence is applied in-place to the reference file, it materializes the version file except for those intervals that are write intervals of *copy-from-V* commands. The in-place reconstructible delta encoding is completed by placing the *copy-from-V* commands, in the same order that they appear in the input delta encoding, after the *add* commands.

7 A Sufficient Condition for CRWI Digraphs

Sections 7, 8, and 9 contain our results on the graph theory and computational complexity of in-place differential encoding. All proofs are available in the Appendix.

In this section we give a simple sufficient condition (Theorem 1) for a digraph to be a CRWI digraph. We use this result to prove the theorems in Sections 8 and 9. We begin by recalling the definition of a CRWI digraph and defining the CRWI digraphs meeting two restrictions. An *interval* is of the form $I = [i, j] \overset{\text{def}}{=} \{i, i+1, \ldots, j\}$ 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 $\mathit{graph}(\mathbf{R}, \mathbf{W})$ as follows: (i) the vertices of $\mathit{graph}(\mathbf{R}, \mathbf{W})$ are $1, \ldots, n$; and (ii) for each pair v, w of vertices with $v \neq w$, there is an edge $\overrightarrow{vw}$ in $\mathit{graph}(\mathbf{R}, \mathbf{W})$ iff $R(v) \cap W(w) \neq \emptyset$.

A digraph $G = (V, E)$ is a *CRWI digraph* if $G = \mathit{graph}(\mathbf{R}, \mathbf{W})$ 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 an 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 an 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* : $V \to \mathbf{N}^+$ is a *length-cost CRWI digraph* if there is an RWIS $(\mathbf{R}, \mathbf{W})$ such that $G = \mathit{graph}(\mathbf{R}, \mathbf{W})$ and $|R(v)| = \mathit{Cost}(v)$ 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 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 $\mathit{indeg}(v)$ denote the number of edges directed into v, and let $\mathit{outdeg}(v)$ denote the number of edges directed out of v. Define $\mathit{indeg}(G)$ to be the maximum of $\mathit{indeg}(v)$ over all vertices v of G, and define $\mathit{outdeg}(G)$ to be the maximum of $\mathit{outdeg}(v)$ over all vertices v of G. The digraph G has the *1-or-1 edge property* if, for each edge $\overrightarrow{vw}$ of G, either $\mathit{outdeg}(v) = 1$ or $\mathit{indeg}(w) = 1$ (or both).

Theorem 1

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

The proof of Theorem 1 is given in Appendix A. While Theorem 1 shows that the 1-or-1 edge property is a sufficient condition for a digraph to be a CRWI digraph, it is not necessary.

8 Optimal Cycle Breaking on CRWI Digraphs is NP-hard

In this section we formally state 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 [12] 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 *Cost* : $V \rightarrow \mathbf{N}^+$, 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 theorem is proved in Appendix B by a simple transformation from FEEDBACK VERTEX SET to WEIGHTED CRWI FEEDBACK VERTEX SET.

Theorem 2 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, $\mathit{indeg}(G) \leq 2$, and $\mathit{outdeg}(G) \leq 2$.*

9 Complexity of Finding Optimal In-Place Difference Files

The subject of the paper up to this point has been the problem of post-processing a given differential 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 differential encoding. A more general problem is to find an in-place reconstructible differential 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 Differential Encoding
Input: A reference file $\mathcal{R}$ and a version file $\mathcal{V}$.

1. Using an existing differencing algorithm, find an 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 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 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 encoding is itself an NP-hard problem. For this result we define an in-place reconstructible encoding Δ to be one that contains no WR conflict. It is interesting to compare the NP-hardness of minimum-cost in-place differential encoding with the fact that minimum-cost differential encoding (not necessarily in-place reconstructible) can be solved in polynomial time [24, 18, 20].

This NP-hardness result is proved using the following simple measure for the cost of a differential 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 DIFFERENTIAL ENCODING
Instance: Two strings $\mathcal{R}$ and $\mathcal{V}$ of bits, and a $K \in \mathbf{N}^+$.
Question: Is there a differential 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 C.

Theorem 3 BINARY IN-PLACE DIFFERENTIAL ENCODING *is NP-complete.*

10 Conclusions

We have presented algorithms that modify difference 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. Differential 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 support that converting a differential encoding into an in-place reconstructible encoding has limited impact on compression. We also find that for bottom line performance, keeping difference files small to reduce I/O matters more than execution time differences in cycle breaking heuristics, because in-place reconstruction is I/O bound. The algorithm to convert a difference file to an in-place reconstructible difference file requires much less time than generating the difference 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. Two problems of maximizing the compression of an in-place reconstructible difference file have been shown NP-hard: first, when the input is a difference 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 difference file for them. The first result justifies our use of efficient, but not optimal, heuristics for cycle breaking.

In-place reconstructible differencing provides the benefits of differencing 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. Differential 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.

11 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 difference 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 difference for a specific space bound. Systems benefit from using the same difference file to update software on many devices. For example, distributing an updated calendar program to many PDAs. In such cases, in-place reconstruction offers a lowest common denominator solution that works for every receiver in exchange for a little lost compression.

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

References

[1] M. Ajtai, R. Burns, R. Fagin, D. D. E. Long, and L. Stockmeyer. Compactly encoding unstructured input with differential compression. *Journal of the ACM*, 49(3), 2002.

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

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

[4] R. C. Burns and D. D. E. Long. Efficient distributed backup and restore with delta compression. In *Proceedings of the Fifth Workship on I/O in Parallel and Distributed Systems*, 1997.

[5] M. C. Chan and T. Y. C. Woo. Cache-based compaction: A new technique for optimizing web transfer. In *Proceedings of the IEEE INFOCOM Conference on Computer Communications*, 1999.

[6] 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*, 1997.

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

[8] S. P. de Jong. Combining of changes to a source file. *IBM Technical Disclosure Bulletin*, 15(4), 1972.

[9] C. W. Fraser and E. W. Myers. An editor for revision control. *ACM Transactions on Programming Languages and Systems*, 9(2), 1987.

[10] 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*, 1996.

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

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

[13] C. Kaufman, R. Perlman, and M. Speciner. *Network Security: Private Communication in a Public World*. Prentice-Hall, 1995.

[14] D. G. Korn and K.-P. Vo. Engineering a differencing and compression data format. In *Proceedings of USENIX Annual Technical Conference*, 2002.

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

[16] J. MacDonald. Versioned file archiving, compression, and distribution. Technical Report available at http://-www.cs.berkeley.edu/∼jmacd/, UC Berkeley.

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

[18] W. Miller and E. W. Myers. A file comparison program. *Software – Practice and Experience*, 15(11), 1985.

[19] 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*, 1997.

[20] C. Reichenberger. Delta storage for arbitrary non-text files. In *Proceedings of the 3rd International Workshop on Software Configuration Management*, 1991.

[21] M. J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, SE-1(4), 1975.

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

[23] F. Stevenson. Cryptanalysis of contents scrambling system. Technical Report available at – http://www.lemuria.org/DeCSS/decss.html, 1999.

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

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

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

[27] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5), 1978.

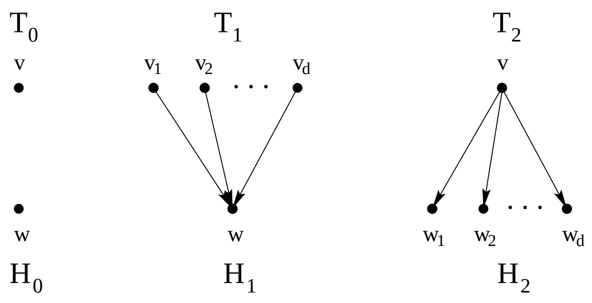


Figure 10: Examples of vertices in the sets H_j, T_j for $j = 0, 1, 2$. All edges directed out of $v, v_1, \ldots, v_d$ or into $w, w_1, \ldots, w_d$ are shown. Edges directed into $v, v_1, \ldots, v_d$ or out of $w, w_1, \ldots, w_d$ are not shown.

Appendix

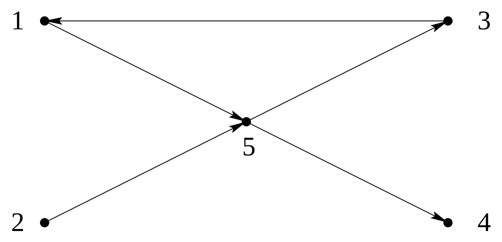
A Proof of Theorem 1

We prove parts 1 and 2 together. For both parts we assume that $G = (V, E)$ has the 1-or-1 edge property and *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 $indeg(G) \leq 2$, then the chosen read intervals are pairwise disjoint. If in addition $outdeg(G) \leq 2$, then the choices also satisfy the length-cost condition $|R(v)| =$ *Cost*(v) for all $v \in V$.

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 10 the sets H_j, T_j for $j = 0, 1, 2$ are illustrated in general, and Figure 11 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.)

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
        $R(v) = [k, k + Cost(v) - 1]$; remove $v$ from $T_0$; and set $k \leftarrow k + Cost(v)$.
    (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
        $W(w) = [k, k + Cost(w) - 1]$; remove $w$ from $H_0$; and set $k \leftarrow k + Cost(w)$.
1.  While $T_1 \neq \emptyset$:
```



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

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

Figure 11: Example of the sets H_j, T_j for a particular digraph.

(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 $indeg(w) = d \geq 2$, 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 $outdeg(v_i) \geq 2$, this would contradict the 1-or-1 edge property because $\overrightarrow{v_i w} \in E$, $outdeg(v_i) \geq 2$, and $indeg(w) \geq 2$. A consequence of the claim is that $w \in H_1$. (See Figure 10.)

(b) Choose $R(v_1) = [l, r]$ and $W(w) = [l', r+1]$ such that $\min\{l, l'\} = k$, $|\,[l, r]\,| = Cost(v_1)$, and $|\,[l', r+1]\,| = Cost(w)$. Because $Cost(w) \geq 2$, 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 $|\,[r+1, r_i]\,| = Cost(v_i)$ 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 $outdeg(G) \leq 2$ 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 $outdeg(G) \leq 2$ then the chosen read intervals are pairwise disjoint.

(c) Remove $v_1, \ldots, v_d$ from T_1. Because $\overrightarrow{vw} \in E$ 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 $k \leftarrow \max\{r+1, r_2, r_3, \ldots, r_d\} + 1$.

2. While $T_2 \neq \emptyset$:

(a) Let v be an arbitrary vertex in T_2, let $d = outdeg(v)$ (so $d \geq 2$), and let $w_1, \ldots, w_d$ be the vertices such that $\overrightarrow{vw_i} \in E$ 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, $indeg(w_i) = 1$ for all i. (See Figure 10.)

(b) If $outdeg(G) \leq 2$, 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 $Cost(v) \geq 2$ 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 $outdeg(G) \geq 3$, 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 $R(v), W(w_1), \ldots, W(w_d)$.

B Proof of Theorem 2

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

Lemma 2 *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 $\overrightarrow{xy}$ 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 $indeg(G) \leq 2$. We leave to the reader the easy verifications that G has the 1-or-1 edge property, $|V| \leq 4|V'|^2$, and $\phi(G) = \phi(G')$. ■

We now return to the proof that WEIGHTED CRWI FEEDBACK VERTEX SET is NP-complete. 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 2. Let $Cost \equiv C$. Because G has the 1-or-1 edge property, $outdeg(G) \leq 2$, and $indeg(G) \leq 2$, Theorem 1 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 2. Therefore, the output of the reduction is $(G, Cost)$ and CK.

C Proof of Theorem 3

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 DIFFERENTIAL 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 differential 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 2, 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

$$\alpha_v = 10100 b_1 00 b_2 00 b_3 00 \ \ldots \ b_l 00$$

where $b_1 b_2 b_3 \ldots b_l$ 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 Theorem 1 that $(G, Cost)$ is a disjoint-read length-cost CRWI digraph. Because the interval-finding procedure in the proof of Theorem 1 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 Theorem 1, 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 $1 \le j_1 < j_2 \le n$, $i_1 \in R(\rho(j_1))$, and $i_2 \in R(\rho(j_2))$, then $i_1 < i_2$ (in fact, $i_1 \le i_2 - 3$). 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 $\overrightarrow{vw}$ is an edge of G. That (ii) can be accomplished follows from the facts $G = \mathit{graph}(\mathbf{R}, \mathbf{W})$, 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 $1 \le i < n$.

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 $v \ne \sigma(n)$, 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 $v \ne w$ then $\alpha_v \ne \alpha_w$. 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 $1 \le i < n$.

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

$\phi(G) \le K' \;\Leftrightarrow\;$ there is a conflict-free differential encoding Δ of $\mathcal{V}$ such that the cost of Δ is at most K.

($\Rightarrow$) Let $\phi(G) \le K'$ and let S be a FVS for G with $|S| \le K'$. 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 $L_{\mathcal{V}} - nL + n + |S| \le L_{\mathcal{V}} - nL + n + K' = K$. 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 differential 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 differential 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.