Compactly Encoding Unstructured Inputs with Differential Compression

MIKLOS AJTAI

IBM Almaden Research Center, San Jose, California

RANDAL BURNS

Johns Hopkins University, Baltimore, Maryland

RONALD FAGIN

IBM Almaden Research Center, San Jose, California

DARRELL D. E. LONG

University of California—Santa Cruz, Santa Cruz, California

AND

LARRY STOCKMEYER

IBM Almaden Research Center, San Jose, California

Abstract. The subject of this article is *differential compression*, the algorithmic task of finding common strings between versions of data and using them to encode one version compactly by describing it as a set of changes from its companion. A main goal of this work is to present new differencing algorithms that (i) operate at a fine granularity (the atomic unit of change), (ii) make no assumptions about the format or alignment of input data, and (iii) in practice use linear time, use constant space, and give good compression. We present new algorithms, which do not always compress optimally but use considerably less time or space than existing algorithms. One new algorithm runs in $O(n)$ time and $O(1)$ space in the worst case (where each unit of space contains $\lceil \log n \rceil$ bits), as compared to

---

The work of R. Burns was performed while at the IBM Almaden Research Center.

The work of D. D. E. Long was performed while a Visiting Scientist at the IBM Almaden Research Center.

Authors' addresses: M. Ajtai, R. Fagin, and L. Stockmeyer, IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120, e-mail: {ajtai,fagin,stock}@almaden.ibm.com; R. Burns, Department of Computer Science, Johns Hopkins University, Baltimore, MD 21218, e-mail: randal@cs.jhu.edu; D. D. E. Long, Department of Computer Science, University of California—Santa Cruz, Santa Cruz, CA 95064, e-mail: darrell@cse.ucsc.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this worked owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2002 ACM 0004-5411/02/0500-0318 $5.00

algorithms that run in $O(n)$ time and $O(n)$ space or in $O(n^2)$ time and $O(1)$ space. We introduce two new techniques for differential compression and apply these to give additional algorithms that improve compression and time performance. We experimentally explore the properties of our algorithms by running them on actual versioned data. Finally, we present theoretical results that limit the compression power of differencing algorithms that are restricted to making only a single pass over the data.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*version control*; E.4 [**Data**]: Coding and Information Theory—*data compaction and compression*; E.5 [**Data**]: Files—*backup/recovery*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems; H.2.7 [**Database Management**]: Database Administration—*data warehouse and repository*; H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*web-based services*

General Terms: Algorithms, Experimentation, Performance, Theory

Additional Key Words and Phrases: Delta compression, differencing, differential compression

1. *Introduction*

Differential compression allows applications to encode compactly a *new version* of data with respect to a previous or *reference* version of the same data. A differential compression algorithm locates substrings common to both the new version and the reference version, and encodes the new version by indicating (1) substrings that can be located in the reference version and (2) substrings that are added explicitly. This encoding, called a *delta version* or *delta encoding*, is often compact and may be used to reduce both the cost of storing the new version and the time and network usage associated with distributing the new version. In the presence of the reference version, the delta encoding can be used to rebuild or materialize the new version.

The first applications using differencing algorithms took two versions of text data as input, and gave as output those lines that changed between versions [de Jong 1972]. Software developers and other authors used this information to control modifications to large documents and to understand the fashion in which data changed. An obvious extension to this text differencing system was to use the output of the algorithm to update an old (reference) version to the more recent version by applying the changes encoded in the delta version. The delta encoding may be used to store a version compactly and to transmit a version over a network by transmitting only the changed lines.

By extending this concept of delta management over many versions, practitioners have used differencing algorithms for efficient management of document control and source code control systems such as SCCS [Rochkind 1975] and RCS [Tichy 1985]. Programmers and authors make small modifications to active documents and check them in to the document control system. All versions of data are kept, so that no prior changes are lost, and the versions are stored compactly, using only the changed lines, through the use of differential compression.

Early applications of differencing used algorithms whose worst-case running time is quadratic in the length of the input files. For large inputs, performance at this asymptotic bound proves unacceptable. Therefore, differencing was limited to small inputs, such as source code text files.

The quadratic time of differencing algorithms was acceptable when limiting differential compression applications to text, and assuming granularity and alignment in data to improve the running time. However, new applications requiring the management of versioned data have created the demand for more efficient differencing

algorithms that operate on unstructured inputs, that is, data that have no assumed alignment or granularity. We mention three such applications.

(1) Delta versions may be used to distribute software over low bandwidth channels like the Internet [Burns and Long 1998]. Since the receiving machine has an old version of software, firmware or operating system, a small delta version is adequate to upgrade this client. On hierarchical distributed systems, many identical clients may require the same upgrade, which amortizes the costs of computing the delta version over many transfers of the delta version.

(2) Recently, interest has appeared in integrating delta technology into the HTTP protocol [Banga et al. 1997; Mogul et al. 1997]. This work focuses on reducing the data transfer time for text and HTTP objects to decrease the latency of loading updated web pages. More efficient algorithms allow this technology to include the multimedia objects prevalent on the Web today.

(3) In a client/server backup and restore system, clients may perform differential compression, and may exchange delta versions with a server instead of exchanging whole files. This reduces the network traffic (and therefore the time) required to perform the backup and it reduces the storage required at the backup server [Burns and Long 1997]. Indeed, this is the application that originally inspired our research. Our differential compression algorithms are the basis for the Adaptive Differencing technology in IBM's Tivoli Storage Manager product.

Although some applications must deal with a sequence of versions, as described above, in this article we focus on the basic problem of finding a delta encoding of one version, called simply the "version", with respect to a prior reference version, called simply the "reference". We assume that data is represented as a string of symbols, for example, a string of bytes. Thus, our problem is, given a reference string R and a version string V, to find a compact encoding of V using the ability to copy substrings from R.

1.1. PREVIOUS WORK. Differential compression emerged as an application of the string-to-string correction problem [Wagner and Fischer 1973], the task of finding the minimum cost edit that converts string R (the reference string) into string V (the version string). Algorithms for the string-to-string correction problem find a minimum cost edit, and encode a conversion function that turns the contents of string R into string V. Early algorithms of this type compute the longest common subsequence (*LCS*) of strings R and V, and then regard all characters not in the LCS as the data that must be added explicitly. The LCS is not necessarily connected in R or V. This formulation of minimum cost edit is reasonable when there is a one-to-one correspondence of matching substrings in R and V, and matching substrings appear in V in the same order that they appear in R.

Smaller cost edits exist if we permit substrings to be copied multiple times and if copied substrings from R may appear out of sequence in V. This problem, which is termed the "string-to-string correction problem with block move" [Tichy 1984], presents a model that represents both computation and I/O costs for delta compression well.

Traditionally, differencing algorithms have been based upon either dynamic programming [Miller and Myers 1985] or the greedy algorithm [Reichenberger 1991]. These algorithms solve the string-to-string correction problem with block move optimally, in that they always find an edit of minimum cost. These algorithms use time quadratic in the size of the input strings, and use space that grows linearly.

A linear time, linear space algorithm, derived from the greedy algorithm, sacrifices compression optimality to reduce asymptotic bounds [MacDonald 2000].

There is a greedy algorithm based on suffix trees [Weiner 1973] that solves the delta encoding problem optimally using linear time and linear space. Indeed, the delta encoding problem (called the "file transmission problem" in Weiner [1973]) was one of the original motivations for the invention of suffix trees. The space to construct and store a suffix tree is linear in the length of the input string (in our case, the reference string). Although much effort has been put into lowering the multiplicative constant in the linear space bound (two recent papers are Grossi and Vitter [2000] and Kurtz [1999]), the space requirements prevent practical application of this algorithm for differencing large inputs.

Linear time and linear space algorithms that are more space-efficient are formulated using Lempel–Ziv [Ziv and Lempel 1977, 1978] style compression techniques on versions. The Vdelta algorithm [Hunt et al. 1998] generalizes the library of the Lempel–Ziv algorithm to include substrings from both the reference string and the version string, although the output encoding is produced only when processing the version string. The Vdelta algorithm relaxes optimal encoding to reduce space requirements, although no sublinear asymptotic space bound is presented. Based on the description of this algorithm in Hunt et al. [1998], the space appears to be at least the length of the L–Z compressed reference string (which is typically some fraction of the length of the reference string) even when the reference and version strings are highly correlated. Using a similar technique, there is an algorithm [Chan and Woo 1999] that encodes a file as a set of changes from many similar files (multiple reference versions). These algorithms have the advantage that substrings to be copied may be found in the version string as well as the reference string. However, most delta-encoding algorithms, including all those presented in this article, can be modified slightly to achieve this advantage.

Certain delta-encoding schemes take advantage of the structure within versions to reduce the size of their input. Techniques include: increasing the coarseness of granularity (which means increasing the minimum size at which changes may be detected), and assuming that data are aligned (detecting matching substrings only if the start of the substring lies on an assumed boundary). Some examples of these input reductions are:

—Breaking text data into lines and detecting only line-aligned common substrings.
—Coarsening the granularity of changes to a record in a database.
—Differencing file data at block granularity and alignment.

Sometimes data exhibit structure to make such decisions reasonable. In databases, modifications occur at the field and record level, and algorithms that take advantage of the syntax of change within the data can outperform (in terms of running time) more general algorithms. Examples of differencing algorithms that take advantage of the structure of data include a tree-based differencing algorithm for heterogeneous databases [Chawathe and Garcia-Molina 1997] and the MPEG differential encoding schemes for video [Tudor 1995]. However, the assumption of alignment within input data often leads to suboptimal compression; for example, in block-granularity file system data, inserting a single byte at the front of a file can cause the blocks to reorganize drastically, so that none of the blocks may match. We assume data to be arbitrary, without internal structure and without alignment.

By foregoing assumptions about the inputs, general differencing algorithms avoid the loss of compression that occurs when the assumptions are false.

For arbitrary data, existing algorithms provide a number of trade-offs among time, space, and compression performance. At one extreme, a straightforward version of a greedy algorithm gives optimal compression in quadratic time and constant space. Algorithms based on suffix trees provide optimal compression in linear time and linear space. The large (linear) space of suffix trees can be reduced to smaller (but still asymptotically linear in the worst case) space by using hashing techniques as in Reichenberger [1991] (which departs from worst-case linear time performance) or in Hunt et al. [1998] (which departs from optimal compression performance). In many cases, the space usage of these algorithms is acceptable, particularly when the input strings are small in comparison to computer memories. However, for applications to large inputs, neither quadratic time nor linear space are acceptable. Obtaining good compression in linear time when only a small amount of information about the input strings can be retained in memory is a challenge not answered by existing work.

1.2. THE NEW WORK. We describe a family of algorithms for differencing arbitrary inputs at a fine granularity that run in linear time and constant space. Given a reference string R and a version string V, these algorithms produce an encoding of V as a sequence of instructions either to add a substring explicitly, or to copy a substring from some location in R. This allows substrings to be copied out of order from R, and allows the same substring in R to participate in more than one copy. These algorithms improve existing technology, since (1) they are more efficient than previous algorithms; (2) they need not assume alignment to limit the growth of the time and space required to find matching substrings; and (3) they can detect and encode changes at an arbitrary granularity. This family of algorithms is constructed in a modular way. The underlying structure consists of two basic strategies and two additional techniques. The algorithms that implement the basic strategies (the "one-pass" and "1.5-pass" strategies) run in linear time and constant space. One of the techniques ("encoding correction", or for short, "correction") can be applied to improve the compression performance of either basic algorithm, while increasing somewhat the running time and implementation complexity of the algorithm. The other technique ("checkpointing") is useful when dealing with very large inputs. This modular structure gives the designer of a differential compression algorithm flexibility in choosing an algorithm based on various requirements, such as running time and memory limitations, compression performance, and the sizes of inputs that are expected. Moreover, the techniques of correction and checkpointing have the potential for new applications beyond those discussed in this article.

We now outline and summarize the article. In Section 2, we give an overview of the differencing problem, and we describe certain basic techniques, such as hashing, that are common to our new differencing algorithms as well as to several previous ones. In Section 3, we describe a "greedy" differencing algorithm modified from Reichenberger [1991]. This algorithm always finds an optimally small delta encoding, under a certain natural definition of the size of a delta encoding. The principal drawback of this algorithm is that its computational resources are high, namely, quadratic time and linear space. We present this algorithm mainly to provide a point of comparison for the new algorithms. In particular, since the new algorithms

do not always find an optimally small delta encoding, we compare the compression performance of the new algorithms with that of the greedy algorithm to evaluate experimentally how well the new algorithms perform.

In Section 4, we present our first new algorithm. It uses linear time and constant space in the worst case. We call this algorithm the "one-pass algorithm", because it can be viewed as making one pass through each of the two input strings. Even though the movement through a string is not strictly a single pass, in that the algorithm can occasionally jump back to a previous part of the string, we prove that the algorithm uses linear time.

In Section 5, we give a technique that helps to mitigate a deficiency of the one-pass algorithm. Because the one-pass algorithm operates with only limited information about the input strings at any point in time, it can make bad decisions about how to encode a substring in the version string; for example, it might decide to encode a substring of the version string as an explicitly added substring, when this substring can be much more compactly encoded as a copy that the algorithm discovers later. The new technique, which we call *correction*, allows an algorithm to go back and correct a bad decision if a better one is found later. In Section 6, the technique of correction is integrated into the one-pass algorithm to obtain the "correcting one-pass algorithm." The correcting one-pass algorithm can use superlinear time on certain adversarial inputs. However, the running time is observed to be linear on experimental data.

Section 7 contains an algorithm that uses a strategy for differencing that is somewhat different than the one used in the one-pass algorithms. Whereas the one-pass and correcting one-pass algorithms move through the two input strings concurrently, the "1.5-pass algorithm" first makes a pass over the reference string in order to collect partial information about substrings occurring in the reference string. It then uses this information in a pass over the version and reference strings to find matching substrings. (The greedy algorithm uses a similar strategy. However, during the first pass through the reference string, the greedy algorithm collects complete information about substrings, and it makes use of this information when encoding the version string. This leads to its large time and space requirements.) Because correction has proven its worth in improving the compression performance of the one-pass algorithm, we describe and evaluate the "correcting 1.5-pass algorithm," which uses the 1.5-pass strategy together with correction.

In Section 8 we introduce another general tool for the differencing toolkit. This technique, which we call *checkpointing*, addresses the following problem. When the amount of memory available is much smaller than the size of the input, a differencing algorithm can have poor compression performance, because the information stored in memory about the inputs must necessarily be imperfect. The effect of checkpointing is to effectively reduce the inputs to a size that is compatible with the amount of memory available. As one might expect, there is the possibility of a concomitant loss of compression. But checkpointing permits the user to trade off memory requirements against compression performance in a controlled way.

Section 9 presents the results of our experiments. The experiments involved running the algorithms on more than 30,000 examples of actual versioned files having sizes covering a large range. In these experiments, the new algorithms run in linear time, and their compression performance, particularly of the algorithms that employ correction, is almost as good as the (optimal) compression performance of the greedy algorithm.

In Section 10, we give a result that establishes a limitation on the compression performance of strict single-pass algorithms, which can make only a single pass through each input string with no reversals or random-accesses allowed. We show that if memory is restricted, then any such differencing algorithm must have very bad compression performance on "transposed data," where there are strings X and Y such that the reference string is XY and the version string is YX.

2. *An Overview of Version Differencing*

We now introduce and develop the differencing problem. At a high level, a differencing algorithm encodes a version V with respect to a reference R by finding regions of V identical to regions of R, and encoding this data with a reference to the location and size of the data in R. For many applications of interest, including files, databases, and binary executables, the input data are composed of strings or one-dimensional ordered lists of symbols. We refer to the input and output of the algorithm as "strings". Because one of our main goals is to perform differential compression at a fine granularity, we take a "symbol" to be one byte when implementing our algorithms. However, the definitions of our algorithms do not depend on this particular choice.

A differencing algorithm accepts as inputs two strings, a *reference string* R and a *version string* V, and outputs a *delta string* $\Delta_{(R:V)}$. A differencing algorithm A encodes the version string with respect to the reference string:

$$\Delta_{(R:V)} = A(R, V).$$

There is also a reconstruction algorithm, denoted A^{-1}, that reconstructs the version string in the presence of the reference and delta strings:

$$V = A^{-1}(R, \Delta_{(R:V)}).$$

In this article, all specific delta encoding algorithms A are deterministic, and the reconstruction algorithm A^{-1} (also deterministic) is the same for all of our encoding algorithms. In Section 10, we give a lower bound result that is made stronger by allowing the encoding algorithm to be probabilistic.

We allow our differencing algorithms to make random (i.e., nonsequential) accesses to strings V and R. When stating worst-case time bounds, such as $O(n)$, we assume that an algorithm may jump to an arbitrary data offset in a string in unit time.[1] A space bound applies only to the amount of memory used by the algorithm; it does not include the space needed to store the input strings. We use memory for load and store operations that must be done quickly to achieve reasonable time performance. When stating space bounds, we assume that one unit of space is sufficient to hold an integer that represents an arbitrary offset within an input string.

We evaluate the goodness of a differencing algorithm using three metrics:

(1) the time used by the algorithm,

(2) the space used by the algorithm, and

[1] However, the timing results from our experiments reflect the the actual cost of jumps, which can depend on where the needed data is located in the storage hierarchy when the jump is made.

(3) the compression achieved, that is, the ratio of the size of the delta string $\Delta_{(R:V)}$ to the size of the version string V to be encoded.

It is reasonable to expect trade-offs among these metrics; for example, better compression can be obtained by spending more computational resources to find the delta string. Our main goal is to find algorithms whose computational resources scale well to very large inputs, and that come close to optimal compression in practice. In particular, we are interested in algorithms that use linear time and constant space in practice. The constant in our constant space bounds might be a fairly large number, considering the large amount of memory in modern machines. The point is that this number does not increase as inputs get larger, so the algorithms do not run up against memory limitations when input length is scaled up. On the other hand, we want the constant multiplier in the linear time bounds to be small enough that the algorithms are useful on large inputs.

We define some terms that will be used later when talking about strings. Any contiguous part of a string X is called a *substring* (of X). If a and b are offsets within a string X and $a < b$, the *substring from a up to b* means the substring of X whose first (respectively, last) symbol is at offset a (respectively, $b-1$). We denote this substring by $X[a, b)$. The offset of the first symbol of an input string (R or V) is offset zero.

2.1. General Methods for Differential Compression. All of the algorithms we present share certain traits. These traits include the manner in which they perform substring matching and the technique they use to encode and to reconstruct strings. These shared attributes allow us to compare the compression and computational resources of different algorithms. So, before we present our methods, we introduce key concepts for version differencing common to all presented algorithms.

2.1.1. *Delta Encoding and Algorithms for Delta Encoding.* An encoding of a string X with respect to a reference string R can be thought of as a sequence of commands to a reconstruction algorithm that reconstructs X in the presence of R. (We are mainly interested in the case where X is the version string, but it is useful to give the definitions in greater generality.) The commands are performed from left to right to reconstruct the symbols of X in left-to-right order. Each command "encodes" a particular substring of X at a particular location in X. There are two types of commands. A *copy command* has the form (C, l, a), where C is a character (which stands for "copy") and where l and a are integers with $l > 0$ and $a \geq 0$; it is an instruction to copy the substring S of length l starting at offset a in R. An *add command* has the form (A, l, S), where A is a character (which stands for "add"), S is a string, and l is the length of S; it is an instruction to add the string S at this point to the reconstruction of X. Each copy or add command can be thought of as *encoding* the substring S of X that it represents (in the presence of R). Let $\sigma = \langle c_1, c_2, \ldots, c_t \rangle$ be a sequence where $t \geq 1$ and c_i is an copy or add command for $1 \leq i \leq t$, and let X be a string. We say that σ is an *encoding of X (with respect to R)* if $X = S_1 S_2 \ldots S_t$ where S_i is the string encoded by c_i for $1 \leq i \leq t$. For example, if

$$R = \text{ABCDEFGHIJKLMNOP}$$
$$V = \text{QWIJKLMNOBCDEFGHZDEFGHIJKL},$$

then the following sequence of five commands is an encoding of V with respect to R; the substring encoded by each command is shown below the command:

$(A, 2, \text{QW})$	$(C, 7, 8)$	$(C, 7, 1)$	$(A, 1, \text{Z})$	$(C, 9, 3)$
QW	IJKLMNO	BCDEFGH	Z	DEFGHIJKL·

The reference string R will usually be clear from context. An encoding of the version string V will be called either "a delta encoding" generally or "an encoding of V" specifically. A delta encoding or a delta string is sometimes called a "delta" for short.

This high-level definition of delta encoding is sufficient to understand the operation of our algorithms. At a more practical level, a sequence of copy and add commands must ultimately be translated, or coded, into a delta string, a string of symbols such that the sequence of commands can be unambiguously recovered from the delta string. We employ a length-efficient way to code each copy and add command as a sequence of bytes; it was described in a draft to the World Wide Web Consortium (W3C) for delta encoding in the HTTP protocol. This draft is no longer available. However, a similar delta encoding standard is under consideration by the Internet Engineering Task Force (IETF) [Korn and Vo 1999]. This particular byte-coding method is used in the implementations of our algorithms. But conceptually, the algorithms do not depend on this particular method, and other byte-coding methods could be used. To emphasize this, we describe our algorithms as producing a sequence of commands, rather than a sequence of byte-codings of these commands. Although use of another byte-coding method could affect the absolute compression results of our experiments, it would have little effect on the relative compression results, in particular, the compression of our algorithms relative to an optimally compressing algorithm. For completeness, the byte-coding method used in the experiments is described in the Appendix.

2.1.2. *Footprints—Identifying Matching Substrings.* A differencing algorithm needs to match substrings of symbols that are common between two strings, a reference string and a version string. In order to find these matching substrings, the algorithm remembers certain substrings that it has seen previously. However, because of storage considerations, these substrings may not be stored explicitly.

In order to identify compactly a fixed length substring of symbols, we reduce a substring S to an integer by applying a hash function F. This integer $F(S)$ is the substring's *footprint*. A footprint does not uniquely represent a substring, but two matching substrings always have matching footprints. In all of our algorithms, the hash function F is applied to substrings D of some small, fixed length p. We refer to these length-p substrings as *seeds*.

By looking for matching footprints, an algorithm can identify a matching seed in the reference and version strings. By extending the match as far as possible forwards, and in some algorithms also backwards, from the matching seed in both strings, the algorithm hopes to grow the seed into a matching substring much longer than p. For the first two algorithms to be presented, the algorithm finds a matching substring M in R and V by first finding a matching seed D that is a prefix of M (so that $M = DM'$ for a substring M'). In this case, for an arbitrary substring S, we can think of the footprint of S as being the footprint of a seed D that is a prefix of S (with p fixed for each execution of an algorithm, this length-p seed prefix is unique). However, for the second two algorithms (the "correcting" algorithms), the

algorithm has the potential to identify a long matching substring M by first finding a matching seed lying anywhere in M; that is, $M = M'DM''$, where M' and M'' are (possibly empty) substrings. In this case, a long substring does not necessarily have a unique footprint.

We often refer to the *footprint of offset a in string X*; by this we mean the footprint of the (unique length-p) seed starting at offset a in string X. This seed is called the *seed at offset a*.

Differencing algorithms use footprints to remember and locate seeds that have been seen previously. In general, our algorithms use a hash table with as many entries as there are footprint values. All of our algorithms use a hash table for the reference string R, and some use a separate hash table for the version string V as well. A hash table entry with index f can hold the offset of a seed that generated the footprint f. When a seed hashes to a footprint that already has a hash entry in the other string's hash table, a potential match has been found. To verify that the seeds in the two strings match, an algorithm looks up the seeds, using the stored offsets, and performs a symbol-wise comparison. (Since different seeds can hash to the same footprint, this verification must be done.) Having found a matching seed, the algorithm tries to extend the match and encodes the matching substring by a copy command. *False matches*, different seeds with the same footprint, are ignored.

2.1.3. *Selecting a Hash Function.* A good hash function for generating footprints must (1) be run-time efficient and (2) generate a near-uniform distribution of footprints over all footprint values. Our differencing algorithms (as well as some previously known ones) need to calculate footprints of all substrings of some fixed length p (the seeds) starting at all offsets r in a long string X, for $0 \le r \le |X| - p$; thus, two successive seeds overlap in $p-1$ symbols. Even if a single footprint can be computed in cp operations where c is a constant, computing all these footprints in the obvious way (computing each footprint separately) takes about cpn operations, where n is the length of X. Karp and Rabin [1987] have shown that if the footprint is given by a modular hash function (to be defined shortly), then all the footprints can be computed in $c'n$ operations where c' is a small constant independent of p. In our applications, c' is considerably smaller than cp, so the Karp–Rabin method gives dramatic savings in the time to compute all the footprints, when compared to the obvious method. We now describe the Karp–Rabin method.

If $x_0, x_1, \ldots, x_{n-1}$ are the symbols of a string X of length n, let X_r denote the substring of length p starting at offset r. Thus,

$$X_r = x_r x_{r+1} \cdots x_{r+p-1}.$$

Identify the symbols with the integers $0, 1, \ldots, b-1$, where b is the number of symbols. Let q be a prime, the number of footprint values. To compute the modular hash value (footprint) of X_r, the substring X_r is viewed as a base-b integer, and this integer is reduced modulo q to obtain the footprint; that is,

$$F(X_r) = \left(\sum_{i=r}^{r+p-1} x_i b^{r+p-1-i} \right) \mod q. \tag{1}$$

If $F(X_r)$ has already been computed, it is clear that $F(X_{r+1})$ can be computed in a constant number of operations by

$$F(X_{r+1}) = \left(\left(F(X_r) - x_r b^{p-1} \right) \cdot b + x_{r+p} \right) \mod q. \tag{2}$$

All of the arithmetic operations in (1) and (2) can be done modulo q, to reduce the size of intermediate results. Since b^{p-1} is constant in (2), the value $b^{p-1} \bmod q$ can be precomputed once and stored.

In all of our new algorithms, each hash table location f holds at most one offset (some offset of a seed having footprint f), so there is no need to store multiple offsets at the same location in the table. However, in our implementation of the greedy algorithm (Section 3.1) all offsets of seeds that hash to the same location are stored in a linked list; in Knuth [1973] this is called *separate chaining*.

Using this method, a footprint function is specified by two parameters: p, the length of substrings (seeds) to which the function is applied; and q, the number of footprint values. We now discuss some of the issues involved in choosing these parameters. The choice of q involves a trade-off between space requirements and the extent to which the footprint values of a large number of seeds faithfully represent the seeds themselves in the sense that different seeds have different footprints. Depending on the differencing algorithm, having a more faithful representation could result in either better compression or better running time, or both. Typically, a footprint value gives an index into a hash table, so increasing q requires more space for the hash table. On the other hand, increasing q gives a more faithful representation, because it is less likely that two different seeds will have the same footprint. The choice of p can affect compression performance. If p is chosen too large, the algorithm can miss "short" matches because it will not detect matching substrings of length less than p. Choosing p too small can also cause poor performance, but for a more subtle reason. Footprinting allows an algorithm to detect matching seeds of length p, but our algorithms are most successful when these seeds are part of much longer matching substrings; in this case, a matching seed leads the algorithm to discover a much longer match. If p is too small, the algorithm can find many "spurious" or "coincidental" matches that do not lead to longer matches. For example, suppose that we are differencing text files, the reference and version strings each contain a long substring S, and the word "the" appears in S. If p is three symbols, it is possible that the algorithm will match "the" in S in the version string with some other occurrence of "the" outside of S in the reference string, and this will not lead the algorithm to discover the long matching substring S. Increasing p makes it less likely that a substring of length p in a long substring also occurs outside of the long substring in either string. This intuition was verified by experiments where p was varied. As p increased, compression performance first got better and then got worse. The optimum value of p was between 12 and 18 bytes, depending on the type of data being differenced. In our implementations, p is taken to be 16 bytes.

2.2. A NOTATION FOR DESCRIBING DIFFERENCING ALGORITHMS. We have covered some methods that are common to all algorithms that we present. We now develop a common framework for describing the algorithms. The input consists of a reference string R and a version string V. The output of a differencing algorithm is a delta encoding, that encodes V as a sequence of add and copy commands. All of the algorithms in this paper encode V in a left-to-right manner. A fact that we use several times is that, at any point during an execution of one of these algorithms, V is divided into two substrings (so that $V = EU$), where the *encoded prefix* E has already been encoded, and the *unencoded suffix* U has not yet been encoded. For example, at the start of the algorithm, $U = V$, and E is empty.

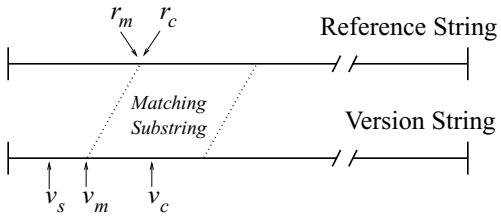


FIG. 1. A configuration of the pointers used for string differencing algorithms, which indicates that the algorithm has scanned from v_s to v_c in the version string and found a matching substring starting at r_m and v_m.

Within the strings R and V, our algorithms use data pointers to mark locations in the input strings. These pointers include:

- v_c Current offset in the version string.
- r_c Current offset in the reference string.
- v_s Starting offset in the version string. This pointer stores the first offset of the unencoded suffix in the version string.
- v_m Matching offset in the version string. When matching substrings are found in V and R, the starting offsets of these substrings are stored in the matching pointers.
- r_m Matching offset in the reference string.

Figure 1 indicates a configuration of these pointers. In this scenario, all data previous to pointer v_s was encoded and was output to the delta encoding. The algorithm then examined data in the interval from v_s to v_c and found matching substrings starting at offsets v_m and r_m. At this point, the algorithm might encode the region from v_s to the end of the match that starts at v_m and r_m, update all pointers, and look for the next matching substring.

In addition to the pointers, we define some functions and quantities to help describe algorithms. These include:

- p Seed length—the length of substrings over which a footprint is calculated.
- q The number of footprint values—each footprint value f is an integer in the range $0 \leq f < q$.
- $F_X(a, a+p)$ The footprint of the substring $X[a, a+p)$ from a up to $a+p$ in the input string X (this substring has length p).
- H_X Hash table, indexed by footprints of seeds in the string X.
- $H_X[i]$ The ith element in H_X. In general, hash table entries contain the starting offsets of seeds indexed by footprint value. For example, $H_V[F_V(a, a+p)]$ might hold a, or it might hold the starting offset of some other seed in V that has the same footprint as the seed $V[a, a+p)$.
- $|X|$ Length or cardinality operator—indicates the length of X when X is a string, and indicates the cardinality of X when X is a set.
- n $= |R| + |V|$, the combined lengths of the reference and version strings.

In later sections, we give pseudocode for four algorithms. The pseudocode is written so that same-numbered steps perform a similar function in all four algorithms; the details vary depending on the algorithm. The basic outline of the steps follows. All four algorithms use the pointer v_c, and two use r_c. Actions involving r_c are in parentheses and are done only if r_c is used.

Algorithm Outline

(1) *Initialize hash table(s)*. Create empty hash table(s). In some algorithms, put information about R in a hash table.

(2) *Initialize pointers*. Set pointers v_c (and r_c) and v_s to offset zero.

(3) *Generate new footprint(s)*. Generate a new footprint at v_c (and at r_c) if there is enough input string to the right of v_c (to the right of r_c) to generate a new footprint. If at least one new footprint was generated, continue at Step (4). If not, go to Step (8) to finish the encoding of V and terminate.

(4) *Try to find a matching seed*. Use the newly generated footprint(s) and the hash table(s) to try to find a matching seed in R and V. In some algorithms, the new footprint(s) are also used to update the hash tables. If a matching seed is not found, increment v_c (and increment r_c) by one, and repeat Step (3). If a matching seed is found, continue at Step (5) to extend the match.

(5) *Extend the match*. Attempt to extend the matching seed to a longer matching substring in R and V, by comparing symbols between R and V. In the greedy and one-pass algorithms, the extension is done forward from the matching seed. In the correcting algorithms, the extension is done both forward and backward from the matching seed. (If no extension is possible, the matching substring is the seed.)

(6) *Encode the match*. Encode the substring of V from v_s to the end of the matching substring by producing the appropriate command sequence; this will always end with a copy command that encodes the matching substring. Update v_s to the new start of the unencoded suffix.

(7) *Update and return to top of main loop*. Update v_c (and r_c). In one algorithm, modify the hash tables. Return to Step (3).

(8) *Finish up*. If there is an unencoded suffix of V, encode this suffix with an add command.

3. *Perfect Differencing*

An algorithm for perfect differencing finds the most compact delta encoding possible. This definition of "perfect" depends on the cost measure for copying and adding substrings. For this discussion, we consider perfect differencing using the following *simple cost measure*:

— A copy command has cost one (regardless of the length and offset of the substring copied from R).

— An add command that adds a substring of length l has cost l.

— The cost of a delta encoding, a sequence of copy and add commands, is the sum of the costs of the commands in the sequence.

The simple cost measure eases analysis while retaining the essence of the problem. Practical methods of byte-coding commands (like the one described in the Appendix) generally have much more complex cost measures, which complicates the analysis of optimality. We now give a useful property of the simple cost measure. Let σ be a delta encoding. Under the simple cost measure, the following transformations of σ do not change the cost of σ: (i) an add command to add a substring S of length l ($l \geq 2$) is replaced by l add commands, each of which adds a single symbol of S; (ii) an add command to add a single symbol x, where x appears in R, is replaced by a copy command to copy x from R. Assuming that these transformations are applied whenever possible, a delta encoding of minimum cost is one that has the minimum number of copy commands. (In such an encoding, the only use of add commands is to add symbols that do not appear in R.)

We consider perfect differencing to be the following version of the string-to-string correction problem with block move [Tichy 1984]: Given R and V, find a delta encoding of V having minimum cost under the simple cost measure.

3.1. A GREEDY DIFFERENCING ALGORITHM. We describe a greedy algorithm based on that of Reichenberger [1991] within our framework. We use it as an example of a perfect differencing algorithm that will serve as the basis for a comparison of the time and compression trade-offs of other algorithms that require less execution time.

Before turning to the pseudocode for the greedy algorithm, we describe its operation informally. The greedy algorithm first makes a pass over the reference string R; it computes footprints and stores in a hash table, for each footprint f, all offsets in R that have footprint f. It then moves the pointer v_c through V, and computes a footprint at each offset. At each step it does an exhaustive search, using the hash table and the strings R and V, to find the longest substring of V starting at v_c that matches a substring appearing somewhere in R. The longest matching substring is encoded as a copy, v_c is set to the offset following the matching substring, and the process continues.

Let us refer now to the pseudocode in Figure 2. In Step (1), the algorithm summarizes the contents of the reference string in a hash table where each entry contains all offsets that have a given footprint; in Steps (3) to (6), it finds longest matching substrings in the version string and encodes them. As noted in Section 3.2, this algorithm is one in a class of algorithms that implement a common greedy method. When we speak of "the greedy algorithm" in the sequel, we mean the algorithm in Figure 2.

The space used by the algorithm is dominated by the space for the hash table. This table stores $|R| - p + 1$ offset values in linked lists. Since p is a constant, the space is proportional to $|R|$. To place an upper bound on the time complexity, break time into periods, where the boundaries between periods are increments of v_c, either in Step (4) or in Step (7). If a period ends with an increment of v_c by l (including the case $l = 1$ in Step (4), the time spent in the period is $O(l|R|)$. This is true because in the worst case, at each offset in R the algorithm spends time $O(l)$ to find a matching substring of length at most l starting at this offset. Because v_c never decreases, the total time is $O(|V||R|)$, that is, $O(n^2)$. The quadratic worst-case bound is met on certain inputs, for example, $R = DzDz \cdots Dz$ and $V = DD \cdots D$, where $|D| = p$ and the character z does not appear in D. In Section 9, we observe that the greedy algorithm does not exhibit quadratic running time in cases where

Greedy Algorithm

Given a reference string R and a version string V, generate a delta encoding of V as follows:

(1) For all offsets in input string R in the interval $[0, |R| - p]$, generate the footprints of seeds starting at these offsets. Store the offsets, indexed by footprint, in the hash table H_R. At each footprint value maintain a linked list of all offsets that hashed to this value, that is, handle colliding footprints by chaining entries at each value:

$$\text{for } a = 0, 1, \ldots, |R| - p : \text{add } a \text{ to the linked list at } H_R[F_R(a, a + p)].$$

(2) Start string pointers v_c and v_s at offset zero in V.

(3) If $v_c + p > |V|$ go to Step (8). Otherwise, generate a footprint $F_V(v_c, v_c + p)$ at v_c.

(4) (and (5)) In this algorithm it is natural to combine the seed matching and substring extension steps into one step. Examine all entries in the linked list at $H_R[F_V(v_c, v_c + p)]$ (this list contains the offsets in R that have footprint $F_V(v_c, v_c + p)$) to find an offset r_m in R that maximizes l, where l is the length of the longest matching substring starting at r_m in R and at v_c in V. If no substring starting at the offsets listed in $H_R[F_V(v_c, v_c + p)]$ matches a substring starting at v_c, increment v_c by one and return to Step (3). Otherwise, set v_m and r_m to the start offsets of the longest matching substring found. (In this algorithm, $v_m = v_c$ at this point.) Let l be the length of this longest matching substring.

(5) The longest match extension has already been done in the combined step above.

(6) If $v_s < v_m$, encode the substring $V[v_s, v_m)$ using an add command containing the substring $V[v_s, v_m)$ to be added. Encode the substring $V[v_m, v_m + l)$ as a copy of the substring of length l starting at offset r_m in R. Set $v_s \leftarrow v_m + l$.

(7) Set $v_c \leftarrow v_m + l$ and return to Step (3).

(8) All of the remaining unencoded input has been processed with no matching substrings found. If $v_s < |V|$, encode the substring $V[v_s, |V|)$ with an add command. Terminate the algorithm.

FIG. 2. Pseudocode for the greedy algorithm.

the reference and version strings are highly correlated, that is, where they have long substrings in common. However, we also observe that the greedy algorithm does exhibit quadratic running time on uncorrelated inputs.

3.2. THE GREEDY METHOD. The greedy algorithm in Figure 2 is one implementation of a general *greedy method* for solving perfect differencing. The key step in an algorithm using the greedy method is the combined Step (4) and (5). In general, this step must find r_m and l such that $R[r_m, r_m + l) = V[v_c, v_c + l)$ and l is as large as possible. In general, Step (1) builds a data structure that summarizes the substrings of R and is used to perform the key step. For example, in the greedy algorithm based on suffix trees, Step (1) creates a suffix tree [Weiner 1973; Gusfield 1997] for R (using time and space $O(n)$). This permits the key step to be performed in time $O(l + 1)$, and it is easy to see that this implies that the total running time is $O(n)$. There is also a simple implementation of the greedy method that uses constant space and quadratic time; in this algorithm, Step (1) is empty (no data structure is constructed) and the key step is done by trying all values of r_m for $0 \le r_m \le |R| - 1$, and for each r_m finding the largest l such that $R[r_m, r_m + l) = V[v_c, v_c + l)$. Even though both this simple algorithm and the algorithm of Figure 2 use quadratic time in the worst case, the latter algorithm typically uses significantly less time in

practice because it limits its search to those r_m where there is known to be a match of length at least p.

3.3. PROVING THAT THE GREEDY ALGORITHM FINDS AN OPTIMAL DELTA ENCODING. Tichy [1984] has proved that any implementation of the greedy method is a solution to perfect differencing. Here we give a version of Tichy's proof that appears simpler; in particular, no case analysis is needed. Although we give the proof for the specific greedy algorithm above, the only properties of the algorithm used are that it implements the greedy method.

We show that if $p \leq 2$, the greedy algorithm always finds a delta encoding of minimum cost, under the simple cost measure defined above. If $p > 2$, the algorithm might not find matching substrings of length 2, so it might not find a delta of minimum cost. (The practical advantage of choosing a larger p is that it decreases the likelihood of finding spurious matches, as described in Section 2.1.3.)

Let R and V be given. Let M be a delta encoding of minimum cost for this R and V, and let G be the delta encoding found by the greedy algorithm. We let $|M|$ and $|G|$ denote the cost of M and G, respectively. Each symbol of V that does not appear in R must be encoded by an add command in both M and G. Because the greedy algorithm encodes V in a left-to-right manner, it suffices to show, for each maximal length substring S of V containing only symbols that appear in R, that the greedy algorithm finds a delta of minimum cost when given R and S. As noted above, the simple cost of a delta encoding does not increase if an add command of length l is replaced by l add commands that copy single symbols. Therefore, for R, V, M, and G as above, it suffices to show that $|G| = |M|$ in the case where every symbol of V appears in R and where M and G contain only copy commands. Because the cost of a copy command is one, $|M|$ (respectively, $|G|$) equals the number of copy commands in M (respectively, G).

For $j \geq 1$, let x_j be the largest integer such that $V[0, x_j)$ can be encoded by j copy commands. Also, let $x_0 = 0$. Let t be the smallest integer such that $x_t = |V|$. The minimality of t implies that $x_0 < x_1 < \cdots < x_t$. By the definition of t, the cost of M is t. To complete the proof, we show by induction on j that, for $0 \leq j \leq t$, the first j copy commands in G encode $V[0, x_j)$. Taking $j = t$, this implies that the cost of G is t, so $|G| = |M|$. The base case $j = 0$ is obvious, where we view $V[0, 0)$ as the empty string. Fix j with $0 \leq j < t$, and assume by induction that the first j copy commands in G encode $V[0, x_j)$. It follows from the definition of the greedy algorithm (more generally, the greedy method) that its $(j+1)$th copy command will encode the longest substring S that starts at offset x_j in V and is a substring of R. Because the longest prefix of V that can be encoded by j copies is $V[0, x_j)$, and $j+1$ copies can encode $V[0, x_{j+1})$, it follows that $V[x_j, x_{j+1})$ is a substring S_j of R. Therefore, the greedy algorithm will encode $S = S_j$ with its $(j+1)$th copy command, and the first $j+1$ copy commands in G encode $[0, x_{j+1})$. This completes the inductive step, and so completes the proof that $|G| = |M|$.

4. *Differencing in Linear Time and Constant Space*

While perfect differencing provides optimally compressed output, the existing methods for perfect differencing do not provide acceptable time and space

performance. Since we are interested in algorithms that scale well and can difference arbitrarily large inputs, we focus on the task of differencing in linear time and constant space. We now present such an algorithm. This algorithm, termed the *one-pass algorithm*, uses basic methods for substring matching, and will serve as a departure point for examining further algorithms that use additional methods to improve compression.

4.1. THE ONE-PASS DIFFERENCING ALGORITHM. The one-pass differencing algorithm finds a delta encoding in linear time and constant space. It finds matching substrings in a "next match" sense. That is, after copy-encoding a matching substring, the algorithm looks for the next matching substring forward in both input strings. It does this by flushing the hash tables after encoding a copy. The effect is that, in the future, the one-pass algorithm ignores the portion of R and V that precedes the end of the substring that was just copy-encoded. The next match policy detects matching substrings sequentially. As a consequence, in the presence of transposed data (with R as $\cdots X \cdots Y \cdots$ and V as $\cdots Y \cdots X \cdots$), the algorithm will not detect both of the matching substrings X and Y.

The algorithm scans forward in both input strings and summarizes the seeds that it has seen by footprinting them and storing their offsets in two hash tables, one for R and one for V. The algorithm uses the footprints in the hash tables to detect matching seeds, and it then extends the match forward as far as possible. Unlike the greedy algorithm, the one-pass algorithm does not store all offsets having a certain footprint; instead it stores, for each footprint, at most one offset in R and at most one in V. This makes the hash table for R smaller (size q rather than $|R|$) and more easily searched, but the compression is not always optimal. At the start of the algorithm, and after each flush of the hash tables, the stored offset in R (respectively, V) is the first one found in R (respectively, V) having the given footprint. Pseudocode for the one-pass algorithm is in Figure 3.

Retaining the first-found offset for each seed is the correct way to implement the next match policy in the following sense, assuming that the hash function is ideal. We say that a hash function F is *ideal* for R and V when, for all seeds s_1 and s_2 that appear in either R or V, if $s_1 \neq s_2$ then $F(s_1) \neq F(s_2)$. Consider an arbitrary time when Step (3) is entered, either for the first time or just after the hash tables have been flushed in Step (7). At this point, the hash tables are empty and the algorithm is starting "fresh" to find another match (r_m, v_m) with $r_c \leq r_m$ and $v_c \leq v_m$. We say that a pair (r, v) of offsets is a *match* if the seeds at offsets r and v are identical. It is not hard to see that if the match (r_m, v_m) is found at this iteration of Steps (3)–(7), then there does not exist a match $(r'_m, v'_m) \neq (r_m, v_m)$ with $r_c \leq r'_m \leq r_m$ and $v_c \leq v'_m \leq v_m$. This property can be violated if the hash function is not ideal, as discussed in Section 4.3.

The one-pass differencing algorithm focuses on finding pairs of *synchronized offsets* in R and V, which indicates that the data at the synchronized offset in R is the same as the data at the synchronized offset in V. The algorithm switches between *hashing mode* (Steps (3) and (4)) where it attempts to find synchronized offsets, and *identity mode* (Step (5)) where it extends the match forward as far as possible. When a match is found and the algorithm enters identity mode, the pointers are "synchronized". When the identity test fails at Step (5), the strings differ and the string offsets are again "out of synch". The algorithm then restarts hashing to regain the location of common data in the two strings.

One-Pass Algorithm

Given a reference string R and a version string V, generate a delta encoding of V as follows:

(1) Create empty hash tables, H_V and H_R, for V and R. Initially, all entries are $\perp$ (i.e., empty).

(2) Start pointers r_c, v_c, and v_s at offset zero. Pointer v_s marks the start of the suffix of V that has not been encoded.

(3) If $v_c + p > |V|$ and $r_c + p > |R|$ go to Step (8). Otherwise, generate footprint $F_V(v_c, v_c + p)$ when $v_c + p \leq |V|$ and footprint $F_R(r_c, r_c + p)$ when $r_c + p \leq |R|$.

(4) For footprints $F_V(v_c, v_c + p)$ and $F_R(r_c, r_c + p)$ that were generated:
   (a) Place the offset v_c (resp., r_c) into H_V (resp., H_R), provided that no previous entry exists. The hash tables are indexed by footprint. That is, if $H_V[F_V(v_c, v_c + p)] = \perp$ assign $H_V[F_V(v_c, v_c + p)] \leftarrow v_c$; similarly, if $H_R[F_R(r_c, r_c + p)] = \perp$ assign $H_R[F_R(r_c, r_c + p)] \leftarrow r_c$.
   (b) If there is a hash table entry at the footprint value in the other string's hash table, the algorithm has found a likely matching substring. For example, $H_V[F_R(r_c, r_c + p)] \neq \perp$ indicates a likely match between the seed at offset r_c in R and the seed at offset $H_V[F_R(r_c, r_c + p)]$ in V. In this case set $r_m \leftarrow r_c$ and $v_m \leftarrow H_V[F_R(r_c, r_c + p)]$ to the start offsets of the potential match. Check whether the seeds at offsets r_m and v_m are identical. If the seeds prove to be the same, matching substrings have been found. If this is the case, continue at Step (5) to extend the match (skipping the rest of Step (4b)). Symmetrically, if $H_R[F_V(v_c, v_c + p)] \neq \perp$, set $v_m \leftarrow v_c$ and $r_m \leftarrow H_R[F_V(v_c, v_c + p)]$. If the seeds at offsets r_m and v_m are identical, continue at Step (5) to extend the match. At this point, no match starting at v_c or starting at r_c has been found. Increment both r_c and v_c by one, and continue hashing at Step (3).

(5) At this step, the algorithm has found a matching seed at offsets v_m and r_m. The algorithm matches symbols forward in both strings, starting at the matching seed, to find the longest matching substring starting at v_m and r_m. Let l be the length of this substring.

(6) If $v_s < v_m$, encode the substring $V[v_s, v_m)$ using an add command containing the substring $V[v_s, v_m)$ to be added. Encode the substring $V[v_m, v_m + l)$ as a copy of the substring of length l starting at offset r_m in R. Set v_s to the offset following the end of the matching substring, that is, $v_s \leftarrow v_m + l$.

(7) Set r_c and v_c to the offset following the end of the match in R and V, that is, $r_c \leftarrow r_m + l$ and $v_c \leftarrow v_m + l$. Flush the hash tables by setting all entries to $\perp$. We use a non-decreasing counter (version number) with each hash entry to invalidate hash entries logically. This effectively removes information about the strings previous to the new current offsets v_c and r_c. Return to hashing again at Step (3).

(8) All input has been processed. If $v_s < |V|$, output an add command for the substring $V[v_s, |V|)$. Terminate the algorithm.

FIG. 3. Pseudocode for the one-pass algorithm.

In Section 10, we show that differencing algorithms that take a single pass over the input strings, with no random access allowed, achieve suboptimal compression when data are transposed (i.e., when substrings occur in a different order in the reference and version strings). The one-pass algorithm is not a strict single pass algorithm in the sense of Section 10, as it performs random access in both strings to verify the identity of substrings with matching footprints. However, it does exhibit the limitations of a strict single pass algorithm in the presence of transpositions, because after finding a match it limits its search space for

subsequent matches to only the offsets greater than the end of the previous matching substring.

4.2. TIME AND SPACE ANALYSIS. We now show that the one-pass algorithm uses time linear in $np + q$ and space linear in q; recall that q is the number of footprint values (i.e., the number of entries in each hash table), and $n = |R| + |V|$.[2] We always use the algorithm with p a (small) constant and q a (large) constant (i.e., p and q do not depend on n). In this case, the running time is $O(n)$ and the space is $O(1)$ in the worst case.

THEOREM 4.1. *If the one-pass algorithm is run with seed length p, the number of footprints equal to q, and input strings of total length n, then the algorithm runs in time $O(np + q)$ and space $O(q)$.*

PROOF. The constants implicit in all occurrences of the O-notation do not depend on p, q, R, V, or n.

The space bound is clear. At all times, the algorithm maintains two hash tables, each of which contains q entries. After finding a match, hash entries are flushed and the same hash tables are reused to find the next matching substring. Except for the hash tables, the algorithm uses space $O(1)$. So the total space is $O(q)$.

We now show the time bound. Initially, during Steps (1) and (2), the algorithm takes time $O(q)$. For the remainder of the run of the algorithm, divide time into periods, where the boundaries between periods are the times when the algorithm enters hashing mode; this occurs for the first time when it moves from Step (2) to Step (3), and subsequently every time it moves from Step (7) to Step (3). Focus on an arbitrary period, and let r_c^0 and v_c^0 be the values of r_c and v_c at the start of the period. In particular, the start v_s of the unencoded suffix is v_c^0. At this point, the hash tables are empty. We follow the run of the algorithm during this period, and bound the time used during the period in terms of the net amount that the pointers r_c and v_c advance during the period. (By "net," we mean that we subtract from the net advancement when a pointer moves backwards.) When r_c and v_c are advancing in hashing mode (repetitions of Steps (3) and (4)), the algorithm uses time $O(p)$ each time that these pointers advance by one. When a matching seed is found in Step (4b), either (i) $v_c^0 \le v_m = v_c$ and $r_c^0 \le r_m \le r_c$, or (ii) $r_c^0 \le r_m = r_c$ and $v_c^0 \le v_m \le v_c$. Let $M = \max(v_m - v_c^0, r_m - r_c^0)$. (The following also covers the case that the algorithm moves to the finishing Step (8) before a match is found.) Because either $v_m = v_c$ or $r_m = r_c$, the total time spent in hashing mode is $O(pM)$. The number of non-$\perp$ hash table entries at this point is at most $2M$. The match extension Step (5) takes time $O(l)$. The encoding Step (6) takes time $O(v_m - v_c^0)$, that is, $O(M)$. In Step (7), the pointers v_c and r_c are reset to the end of the match; let $v_c^1 = v_m + l$ and $r_c^1 = r_m + l$ be the values of v_c and r_c after this is done. These are also the values of these pointers at the start of the next period. It follows that:

(1) v_c and r_c advanced by a total net amount of at least $M + 2l$ during the period (i.e., $(v_c^1 - v_c^0) + (r_c^1 - r_c^0) \ge M + 2l$), and
(2) the time spent in the period was $O(pM + l)$.

[2] The term np is needed to handle the worst-case situation that Step (4b) is executed approximately n times, and at each execution of this step the algorithm spends time p to discover that seeds with the same footprint are not identical. In practice, we would not expect this worst-case situation to happen.

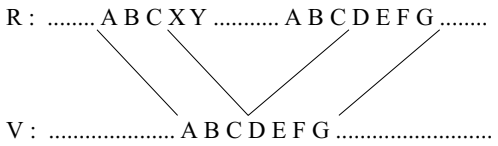


FIG. 4. An example of a spurious match.

It follows that the algorithm runs in time $O(np + q)$, as we now show. Let t be the number of periods, and for $1 \leq i \leq t$ let M_i and l_i be the values of M and l, respectively, for period i. Let M^* (respectively, l^*) be the sum of M_i (respectively, l_i) over $1 \leq i \leq t$. Because the pointers can advance by a total net amount of at most $n = |R| + |V|$ during the entire run, statement 1 above implies that $M^* + 2l^* \leq n$. Initially, during Steps (1) and (2), the algorithm takes time $O(q)$. By Statement (2) above, the algorithm uses time $O(pM^* + l^*)$ during all t periods. It follows that the total time is $O(np + q)$. □

4.3. SUBOPTIMAL COMPRESSION. In addition to failing to detect transposed data, the one-pass algorithm achieves less than optimal compression when it falsely believes that the offsets are synchronized or when the hash function exhibits less than ideal behavior. (Recall that a hash function is "ideal" if it is one-to-one on the set of seeds that appear in R and V.) We now consider each of these issues.

When we say that the algorithm falsely believes that the offsets are synchronized, we are referring to "spurious" or "coincidental" matches (as discussed in Section 2.1.3), which arise as a result of taking the next match rather than the best match. These collisions occur when the algorithm believes that it has found synchronized offsets between the strings when in actuality the collision just happens to be between substrings whose length-p prefixes match by chance, and a longer matching substring exists at another offset. An example of a spurious match is shown in Figure 4. Call the substring ABCDEFG the "long match" in R and V. In this simple example, let $p = 2$, and assume that the symbols A, ..., G, X, Y do not appear anywhere else in R and V. The one-pass algorithm might first find the match between the prefix ABC of the long match in V and the first occurrence of ABC in R, and encode ABC with a copy. Then the hash tables are flushed, so no match starting at the offset of A, B or C in V can be found in the future. The algorithm might later discover the match DEFG in V and R, and encode it with a copy. So the long match is encoded by at least two copy commands, rather than one.

Spurious matches are less likely when the seed length p is increased. However, increasing p increases the likelihood of missed matches, because matches shorter than p will not be detected.

We now discuss problems that arise when the hash function is not ideal. Hash functions generally hash a large number of seeds to a smaller number of footprints, so a footprint does not uniquely represent a seed. Consequently, the algorithm may also experience the *blocking* of footprints, as we now describe. For a newly generated footprint f of an offset b in string S ($S = R$ or V), if there is another offset $a < b$ already occupying f's entry in the hash table H_S, then b is ignored and $H_S[f] = a$ is retained. If the seeds at offsets a and b are different, we say that the footprint f is *blocked*. Referring again to Figure 4, a blocked footprint can

cause an even worse encoding of the long match, as follows. Suppose that the seeds XY and DE have the same footprint f. Assume as before that the algorithm finds the spurious match ABC and encodes it with a copy command. The hash tables are flushed, and the algorithm restarts with the current pointers at X in R and D in V. When the footprints of XY and DE are generated, the algorithm sets $H_R[f]$ to the offset of X. A match is not found because the seeds XY and DE differ, so the algorithm continues in hashing mode. If the algorithm later generates the footprint f of DE in R during the same execution of hashing mode, the offset of D will be discarded in H_R, because $H_R[f]$ already has a nonnull entry. Thus, the best the algorithm can do is to encode D with an add command, and EFG with a copy command.

5. *Compression Improvements to Existing Differencing Methods*

We now introduce a new technique called *encoding correction*, or for short, *correction*, which is an approach to improving the compression performance of our algorithms, including the one-pass algorithm. When a differencing algorithm runs, it outputs the appropriate add and copy commands to a delta encoding. We modify this scheme by sending all add and copy commands to a buffer. By caching commands, an algorithm has the opportunity to recall a subsequence of commands and exchange it for a better one. In many cases, an algorithm that uses this technique can make a quick decision as to an encoding, and if this decision turns out not to be the best decision, the encoding can be undone later by replacing it with a more favorable one. For example, it might happen that two or more adjacent copy commands can be replaced by a single new copy command, or that an explicitly added substring can be wholly or partially encoded by a new copy command. As discussed in the previous section, spurious matches and blocked footprints can cause the one-pass algorithm to make bad decisions. Correction helps to mitigate this bad effect.

When an algorithm corrects the existing encoding of a region of V, this does not necessarily mean that the algorithm has found the "best" encoding of this region. The encoding of this region might be corrected several times as longer and longer portions of this region are found to match substrings in R.

To implement correction, the algorithm inserts add and copy commands into a buffer rather than writing them directly to the delta encoding. The buffer, in this instance, is a FIFO queue of commands called the *encoding lookback buffer*, or for short, the *buffer*. When a substring of the version string is encoded, the appropriate command is written to the buffer. The buffer collects commands until it is full. Then, when writing a command to a full buffer, the oldest command gets pushed out and is written to the delta encoding. When a command "falls out of the buffer," it becomes immutable and has been committed to the delta encoding. In theory, the buffer can have unlimited size, so that commands never fall out. But in practice, it is useful for the buffer to be implemented in memory so that operations on the buffer can be done efficiently, and this limits the size of the buffer.

The objects in the buffer are not strictly add and copy commands, as defined in Section 2.1.1. In particular, an add command in the buffer does not contain the added substring. However, each command in the buffer is associated with its *version offset*, the starting offset of the substring in the version string that this command encodes. The length and version offset stored with an add command permit the

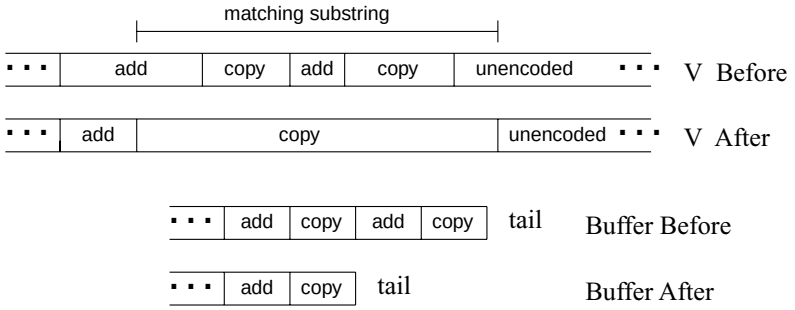


FIG. 5. An illustration of tail correction. Before the correction, a rightmost part of the encoded prefix of V is encoded by an add, copy, add, copy command sequence. The new matching substring in V is encoded as a copy command. This new copy command absorbs a partial add command, a whole copy command, a whole add command, and another whole copy command, and it also encodes part of the previously unencoded suffix of V. The corresponding commands near the tail of the buffer are also shown before and after the correction.

added substring to be found in V when the add command exits the buffer. In addition, the version offsets in the buffer are increasing and distinct. This permits the use of binary search to find commands in the buffer by their version offsets. For simplicity, we refer to the objects in the buffer as "commands," because each object in the buffer specifies a unique add or copy command.

Correction can cause algorithmic running time to depart from a strict linear worst-case bound. But on all experimental inputs (as discussed in Section 9) the one-pass algorithm with correction displays time performance similar to that of the original one-pass algorithm, and the compression is much improved.

5.1. EDITING THE ENCODING LOOKBACK BUFFER. An algorithm may perform two types of correction. *Tail correction* occurs when the algorithm encodes a previously unencoded portion of the version string with a copy command. The algorithm attempts to extend that matching string backwards in both the version and reference strings. If this backward matching substring extends into the prefix of the version string that is already encoded, note that the relevant commands are ones that have been issued most recently, so the relevant commands in the buffer start at the tail of the buffer (hence, the name *tail correction*). In this case, there is the potential to remove or shorten these commands by integrating them into the new copy command. The algorithm will integrate commands from the tail of the buffer into the new copy command as long as the commands in question are either:

—A copy command that can be wholly integrated into the new copy command. If the copy command in the buffer can only be partially integrated, the command should not be reclaimed, as no additional compression can be attained.

—Any wholly or partially integrated add command. Since an add command in the delta encoding contains the data to be added, reclaiming partial add commands benefits the compression. While no commands are reclaimed, the length of the data to be added is reduced and the resulting delta decreases in size accordingly. This is illustrated in Figure 5.

General correction may change any previous commands, not just the most recent commands. If a new matching substring M is found, both starting and ending in the

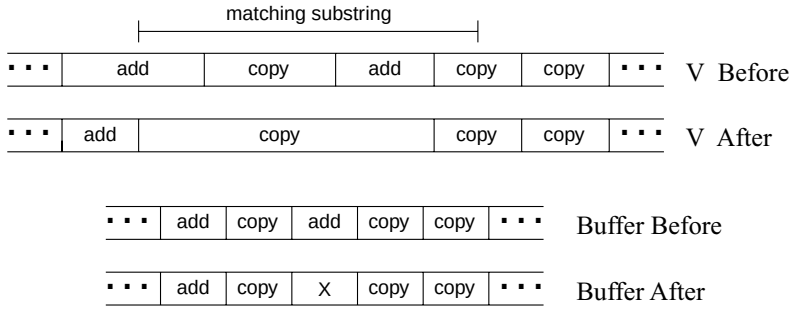


FIG. 6. An illustration of general correction. A new matching substring is found within the previously encoded prefix of V. A copy-encoding of part of the new matching substring absorbs a partial add command, a whole copy command, and a whole add command. The existing copy command that is only partially covered by the matching substring is not absorbed. The relevant regions of V and the buffer are shown before and after the correction. The X in the buffer after the correction is a dummy command, an entry in the buffer where a command was deleted and not replaced by another command.

prefix of V that has already been encoded, the algorithm determines if the existing commands that encode M can be improved, given that M can be copied from R. The algorithm searches through the buffer to find the commands that encode M in V. The algorithm then reencodes this region, reclaiming whole and partial add commands and whole copy commands, to the extent that this gives a better encoding. This is illustrated in Figure 6.

As should be clear from the description and examples of correction, an algorithm that uses this technique should, after finding a matching seed, try to extend the match both backwards and forwards from the matching seed in R and V.

5.2. IMPLEMENTING THE ENCODING LOOKBACK BUFFER. Correction requires that the encoding lookback buffer be both searchable and editable, as the algorithm must efficiently look up previous commands and potentially modify or erase those entries. The obvious implementation of the encoding lookback buffer is a linked list that contains the commands, in order, as they were emitted from a differencing algorithm. This data structure has the advantage of simply supporting the insert, modify, and delete operations on commands. However, finding elements in this list using linear search is time consuming. Consequently, we implemented the encoding lookback buffer as a *circular queue*, a FIFO queue built on top of a fixed-size contiguous region in memory. Two fixed pointers mark the boundaries of the allocated region. Within this region, the data structure maintains pointers to the logical head and tail of the FIFO queue. Simple pointer arithmetic around these four pointers supports the access of the head and tail elements, and appending to or deleting from the head or the tail, all in constant time. Commands in the circular queue can be looked up by their version offsets using binary search, with improved search performance compared to a linked list.

While the implementation of a FIFO queue in a fixed memory region greatly accelerates look-up operations, it does not directly support inserting or deleting elements in the middle of the queue. This is an issue only for general correction, because tail correction operates only on the head and tail of the queue. When performing general correction, we are trying to reduce the size of the delta encoding. Most often, this involves reducing the number of commands that encode any given region, and this can always be done without insertion into the middle of the queue.

Consider an operation that merges a pair of adjacent commands into a single copy command in the middle of the buffer. This operation performs a replace of one command in the pair and a delete of the other. We perform this action by editing one command to contain the new longer copy command, and marking the other command as a "dummy" command. (Whenever an algorithm creates a dummy, the usable length of the buffer is reduced by one until that entry is emitted or reused.) However, there is one case of general correction that is excluded by our implementation. Assume that we have encoded an add command, we later find that a portion of this add command can be reencoded as a copy command, and the add command is not at the head or tail of the buffer at the time we want to correct it. Replacing a portion of the add command with a copy command reduces the size of the delta encoding while increasing the number of commands. Our implementation fails to support this, unless we are lucky enough to find a dummy command adjacent to the add command we are modifying. We feel that this limitation is a desirable trade-off, since we achieve superior search time when using a circular queue, as compared to a linked list.

One might consider balanced search trees (see, e.g., Knuth [1973, Sect. 6.2.3]) as a way to implement the buffer, because they support all of our needed operations in logarithmic time per operation. However, a main goal in choosing a data structure for the buffer is to minimize the time required for operations at the head and tail of the buffer, because these occur much more frequently than operations in the middle of the buffer. In particular, tail correction does not need operations in the middle at all. Circular queues (and linked lists) meet this goal better than balanced search trees.

6. *The Correcting One-Pass Algorithm*

We now reconsider our one-pass differencing algorithm in light of the technique for correction. This combination results in an algorithm that provides superior compression without significantly increasing execution time. We term this algorithm the *correcting one-pass algorithm*.

The one-pass algorithm focuses on finding matches between substrings of R and V in the "next match" sense. It does this by:

(1) flushing the hash tables after encoding a copy command; and
(2) discarding multiple instances of offsets having the same footprint in favor of the first offset having that footprint.

However, strictly adhering to the next match policy causes the one-pass algorithm to lose compression when it fails to find transposed data. Since the correcting one-pass algorithm uses correction to recover the lost compression when making a poor encoding, it may relax the strict policies used for enforcing the next match policy and look more optimistically for any matching substrings, since a better match can still be found later even if it is not contained in the current encoding.

In terms of bookkeeping, the correcting one-pass algorithm differs from the one-pass algorithm by:

(1) keeping all existing entries in the hash tables after encoding a copy command; and

(2) discarding a prior offset that has a particular footprint in favor of the current offset having that footprint.

By keeping all existing entries in the hash table, the correcting one-pass algorithm retains information about past substrings in both input strings. Favoring more recent offsets keeps the hash tables' contents current without flushing the hash tables. In effect, the algorithm maintains a "window into the past" for each input string. This window tends to well represent substrings that have been seen recently, but the representation becomes more sparse for substrings seen longer ago. By retaining information about past substrings, the correcting one-pass algorithm can detect "nonsequential" matching substrings, that is, substrings that occur in the version string in an order different from the order in which they occur in the reference string; in particular, the correcting one-pass algorithm can detect transpositions. The problem with nonsequential matches is that it can lead to spurious matches. Correction deals handily with spurious matches, by exchanging the bad encodings of spurious matches for the "correct" encodings that occur later in the input.

Another significant difference between the one-pass and correcting one-pass algorithms is that the correcting one-pass algorithm performs match extension both backwards and forwards from a matching seed. This is needed to take full advantage of correction. The correcting one-pass algorithm outputs all commands through the encoding lookback buffer. Data is written to the delta encoding only by flushing the buffer at the end of the algorithm or by overflowing the buffer, which causes a command to be written to the delta. Pseudocode for the correcting one-pass algorithm is in Figure 7.

The correcting one-pass algorithm differs from the one-pass algorithm in five ways: (i) in Step (4a), the insert-new rather than retain-existing policy is used for entering an offset at a hash table entry that already contains an offset; (ii) in Step (5), the match is extended backwards as well as forwards; (iii) in Step (6), correction is performed on the buffer, and commands are output to the buffer rather than to the delta; (iv) in Step (7), the current pointers v_c and r_c must advance by at least 1; and (v) in Step (7), the hash tables are not flushed.

Looking again at Figure 4, we can see how either tail correction or general correction might correct a suboptimal encoding of the long match, caused by a spurious match. The correcting one-pass algorithm might also first encode the spurious match ABC, but if it later finds a matching seed anywhere in the long match ABCDEFG in R and V, it will find the complete long match by forwards and backwards match extension, and it will correct the spurious match by integrating it into the new single-copy encoding of the long match. The correcting one-pass algorithm can also experience the blocking of footprints (but the offsets lost are ones appearing earlier in the string, rather than later as in the one-pass algorithm). Reverse matching in the correcting one-pass algorithm can find matches starting at offsets having a blocked footprint, and bad encodings caused by blocked footprints can be corrected.

Despite the similarity between the correcting one-pass algorithm and the one-pass algorithm, the correcting one-pass algorithm does not have the same linear running time guarantee. The algorithm potentially spends a large amount of time extending matches backwards at many executions of Step (5), so that the total time spent during backwards matching grows faster than linearly in the length

Correcting One-Pass Algorithm

Given a reference string R and a version string V, generate a delta encoding of V as follows:

(1) Create empty hash tables, H_V and H_R, for V and R.

(2) Start pointers r_c, v_c, and v_s at offset zero. The pointer v_s marks the start of the suffix of V that has not been encoded.

(3) If $v_c + p > |V|$ and $r_c + p > |R|$ go to Step (8). Otherwise, generate footprint $F_V(v_c, v_c + p)$ when $v_c + p \leq |V|$ and footprint $F_R(r_c, r_c + p)$ when $r_c + p \leq |R|$.

(4) For footprints $F_V(v_c, v_c + p)$ and $F_R(r_c, r_c + p)$:
   (a) Place the offset v_c (resp., r_c) into H_V (resp., H_R). That is, assign $H_V[F_V(v_c, v_c + p)] \leftarrow v_c$ and $H_R[F_R(r_c, r_c + p)] \leftarrow r_c$.
   (b) As in the one-pass algorithm, if $H_V[F_R(r_c, r_c + p)] \neq \perp$, set $r_m \leftarrow r_c$ and $v_m \leftarrow H_V[F_R(r_c, r_c + p)]$ to the start offsets of the potential match. If the seeds at offsets r_m and v_m are identical, continue at Step (5) to extend the match (skipping the rest of Step (4b)).
   Symmetrically, if $H_R[F_V(v_c, v_c + p)] \neq \perp$, set $v_m \leftarrow v_c$ and $r_m \leftarrow H_R[F_V(v_c, v_c + p)]$. If the seeds at offsets r_m and v_m are identical, continue at Step (5) to extend the match. At this point, no match starting at v_c or starting at r_c has been found. Increment both r_c and v_c by one, and continue hashing at Step (3).

(5) Having found a matching seed at Step (4b), extend this match forwards and backwards from v_m and r_m as long as possible, reset v_m and r_m to the start offsets of the matching substring (if the match extended backwards), and set l to the length of the matching substring.

(6) Encode the matching substring and attempt to use this substring to correct encodings in the encoding lookback buffer. One of the following three sub-steps is performed:
   (a) If $v_s \leq v_m$, the matching substring overlaps only the previously unencoded suffix V; it cannot be used to correct encodings already in the buffer. If $v_s < v_m$, encode the substring $V[v_s, v_m)$ using an add command. Encode the substring $V[v_m, v_m + l)$ as a copy of the substring of length l starting at offset r_m in R. Output the command(s) to the buffer. Set $v_s \leftarrow v_m + l$.
   (b) If $v_m < v_s < v_m + l$, the matching substring overlaps both the encoded prefix and the unencoded suffix of V. Perform tail correction as described in Section 5.1. That is, attempt to correct encodings from the tail of the buffer by integrating commands into the new copy command. All whole and partial add commands and all whole copy commands that encode the substring $V[v_m, v_s)$ can be integrated into the new copy command, that also encodes the substring $V[v_s, v_m + l)$. Delete from the buffer all commands that were wholly integrated. Output the new copy command to the buffer. Set $v_s \leftarrow v_m + l$.
   (c) If $v_m + l \leq v_s$, the matching substring overlaps only the existing encoded prefix of V. Perform general correction as described in Section 5.1. That is, perform binary search in the buffer to find the commands that encode the substring $V[v_m, v_m + l)$ and correct sub-optimal encodings when possible. (In this case, v_s does not change.)

(7) Set $v_c \leftarrow \max(v_m + l, v_c + 1)$ and $r_c \leftarrow \max(r_m + l, r_c + 1)$; that is, we set the new values of these pointers to the offsets just following the matching substring, but we also require these pointers to advance by at least 1. Return to Step (3).

(8) All of the input has been processed. Flush all commands from the buffer to the delta encoding. If $v_s < |V|$, encode the substring $V[v_s, |V|)$ with an add command. Terminate the algorithm.

FIG. 7. Pseudocode for the correcting one-pass algorithm.

of the input. While this does not occur in our experiments, adversarial inputs do exist that cause the algorithm to exhibit this behavior. Rather than attempt to place a (superlinear) worst-case upper bound on the running time of the algorithm, we choose instead to let the algorithm express its running time experimentally (Section 9), where the correcting one-pass algorithm mimics the time performance of the one-pass algorithm, and compresses data more efficiently in a large majority of cases.

Note that a pointer r_c or v_c can increase by more than one in Step (7), when the algorithm finds a matching substring that extends past the current value of r_c or v_c. A consequence is that footprints are not computed in a region of R from $r_c + 1$ up to the updated value $r_m + l$, and similarly for V. This leaves a hole in the "window into the past." An alternative would be to continue footprinting and modifying the hash tables when extending a match (in identity mode) past the current value of v_c and r_c. (The one-pass algorithm does not continue footprinting in identity mode. This would be useless in the one-pass algorithm, because the hash tables are flushed before the new footprint information can be used.) The reason that we chose not to continue footprinting in identity mode in the correcting one-pass algorithm was to keep running time small. As previously noted, checking equality of symbols is significantly faster than computing a new footprint, and we expect the algorithm to spend a significant fraction of its time in identity mode when the inputs are highly correlated. There is the potential for a loss of compression if, say, an unfootprinted region of R could have been used to copy-encode another part of V (in addition to the copy-encoding that the algorithm found when passing over this region of R in identity mode). In this case, the additional match might not be found. It could still be found, though, if a match originating at a seed outside the region extends into the region. We have not investigated the compression-time trade-off caused by continuing to generate footprints while in identity mode.

7. *The Correcting 1.5-Pass Algorithm*

In contrast to the correcting one-pass algorithm, which improves the compression of the one-pass algorithm, the correcting 1.5-pass algorithm can be viewed as a reformulation of the greedy algorithm (Section 3.1) that improves the running time of the greedy algorithm. The principal change is that the correcting 1.5-pass algorithm encodes the first matching substrings found, rather than searching exhaustively for the best matching substrings as the greedy algorithm does. Both the greedy algorithm and the correcting 1.5-pass algorithm first make a pass over the reference string computing footprints and storing information in a hash table.[3] But where the greedy algorithm stores, for each footprint, all offsets having that footprint, the correcting 1.5-pass algorithm stores only the first such offset encountered. By not searching through all possible matching substrings, the correcting 1.5-pass algorithm exhibits linear execution time in practice. Since the correcting 1.5-pass algorithm uses correction to repair poor encodings, it can

[3] The name "1.5-pass" was chosen for consistency with the name of the one-pass algorithm, which makes one pass over both halves, R and V, of the input. A 1.5-pass algorithm makes an additional pass over one half of the input (R, V), namely, R.

Correcting 1.5-Pass Algorithm

Given a reference string R and a version string V, generate a delta encoding of V as follows:

(1) For each offset a in input string R in the interval $[0, |R| - p]$, generate the footprint $F_R(a, a + p)$. For each footprint generated, if the entry of H_R indexed by that footprint is empty, store the offset in that entry:

$$\text{for } a = 0, 1, \ldots, |R| - p : \text{if } \; H_R[F_R(a, a + p)] = \perp \; \text{ then } \; H_R[F_R(a, a + p)] \leftarrow a.$$

(2) Start v_c and v_s at offset zero.

(3) If $v_c + p > |V|$ go to Step (8). Otherwise, generate a footprint $F_V(v_c, v_c + p)$ at v_c.

(4) If $H_R[F_V(v_c, v_c + p)] \neq \perp$, check that the seed in R at offset $H_R[F_V(v_c, v_c + p)]$ is identical to the seed in V at offset v_c. If matching seeds are found, continue at Step (5). Otherwise, increment v_c by one and repeat Step (3).

(5) Extend the matching substring forwards and backwards as far as possible from the matching seed. Set v_m and r_m to the start of the matching substring in V and R, respectively, and set l to the length of this substring. Note that $v_s \leq v_c < v_m + l$ because v_c never decreases, and because the match originated at the seed at offset v_c.

(6) Encode the match and attempt to correct bad encodings. The following two sub-steps are identical to sub-steps (6a) and (6b) in the correcting one-pass algorithm (sub-step (6c) cannot occur here, because $v_s < v_m + l$):

  (a) If $v_s \leq v_m$: If $v_s < v_m$, encode the substring $V[v_s, v_m)$ using an add command. Encode the substring $V[v_m, v_m + l)$ as a copy of the substring of length l starting at offset r_m in R. Output the command(s) to the buffer. Set $v_s \leftarrow v_m + l$.

  (b) If $v_m < v_s$: We have noted in Step (5) that $v_s < v_m + l$. Attempt to correct encodings from the tail of the buffer (tail correction described in Section 5.1). Delete from the buffer all commands that were wholly integrated. Output the new copy command to the buffer. Set $v_s \leftarrow v_m + l$.

(7) Set $v_c \leftarrow v_m + l$ and return to Step (3).

(8) All of the input has been processed. Flush all commands from the buffer to the delta encoding. If $v_s < |V|$, encode the substring $V[v_s, |V|)$ with an add command. Terminate the algorithm.

FIG. 8. Pseudocode for the correcting 1.5-pass algorithm.

optimistically encode the first matching substring it finds, and rely on correction to improve the encoding if a better one is found later. Pseudocode for this algorithm is in Figure 8.

The correcting 1.5-pass algorithm differs from the greedy algorithm at Steps (1), (5), and (6). In Step (1), the correcting 1.5-pass algorithm keeps only a single offset for each footprint value. At Step (5), the correcting 1.5-pass algorithm extends the match backwards as well as forwards. (Because the greedy algorithm optimally encodes the encoded prefix of V at every step, extending backwards would not help.) Step (6a) differs only because the correcting 1.5-pass algorithm outputs commands to the buffer rather than directly to the delta. Step (6b) not only outputs a command to the buffer, but can also correct bad encodings from the tail of the buffer. As explained in Step (6) of Figure 8, the general correction Step (6c) of the correcting one-pass algorithm is irrelevant to the correcting 1.5-pass algorithm. Because the correcting 1.5-pass algorithm does only tail correction, the problem of inserting

commands in the middle of the buffer does not occur with this algorithm, and it never has to use dummy commands.

As with the correcting one-pass algorithm, the correcting 1.5-pass algorithm has the potential to exhibit superlinear time behavior in the worst case. This can again be attributed to the potential for the algorithm to spend superlinear time during backwards matching. Again, the bad inputs for the correcting 1.5-pass algorithm are adversarial and not witnessed by the inputs in our experiments. The algorithm exhibits linear asymptotic execution time on all examined inputs and even runs faster than the correcting one-pass algorithm on many inputs.

One can also consider the *1.5-pass algorithm*, which is the correcting 1.5-pass algorithm without correction. The 1.5-pass algorithm can be described by the pseudocode in Figure 8, with a few modifications: in Step (5), the match is extended only forwards, not backwards; Step (6b) does not occur; and there is no encoding lookback buffer, so commands are output directly to the delta. Because the 1.5-pass algorithm does not do backwards match extension, it runs in linear time in the worst case. Based on our experience with the one-pass and correcting one-pass algorithms, we would expect the 1.5-pass algorithm to run a little faster than the correcting 1.5-pass algorithm in practice, but give worse compression.

The correcting 1.5-pass algorithm will likely have poor compression performance when the size of the reference string is much larger than the size of the hash table. In the next section, we explain why this poor performance occurs, and we describe a technique that essentially reduces the length of the inputs to a length that is compatible with the size of the hash table.

8. *Handling Arbitrarily Large Input Data*

The correcting algorithms have compression limitations when operating on very large inputs. These limitations arise as a result of the hash tables these algorithms use becoming overloaded. As increasing the size of the hash table is not a scalable solution, we present a method to reduce the amount of information in an input string to an amount that is compatible with the correcting one-pass and correcting 1.5-pass algorithms.

The compression performance of the correcting 1.5-pass algorithm breaks down when there are many more offsets than footprint values, that is, when the length of the reference string significantly exceeds the size of the hash table. When this occurs, many offsets in the reference string have the same footprint value and must be discarded. As the correcting 1.5-pass algorithm chooses to remember the first offset found for each footprint, the hash table contains better information about the earlier substrings in the string. For long inputs, information at the end of the string can be sparse, as many footprints will have already occurred. As the reference string get larger and larger, the hash table contains less information about the whole string and compression degrades accordingly.

In contrast, the correcting one-pass algorithm loses the ability to detect "distant" matching data when inputs grow. The hash tables in the correcting one-pass algorithm always contain the last offset with a particular footprint. So when inputs are large, the hash table does not cover all previously seen offsets, only the recently seen offsets (the "window into the past"). If matching substrings occur far apart from

synchronized offsets, the algorithm may miss them, as the "distant" offsets will have been replaced. This mode of failure is far less severe than that of the correcting 1.5-pass algorithm. Consequently, without the use of some additional technique to effectively reduce the size of the inputs (such as the technique to be described in this section), the correcting one-pass algorithm is preferable for large inputs.

To address large inputs, we present a technique called *checkpointing*. The checkpointing technique declares a certain subset of all possible footprints to be checkpoints. The algorithm will then only operate on footprints that are in this checkpoint subset. An algorithm modified with checkpointing will still compute a footprint every time that the original algorithm would, but only those footprints that are in the checkpoint subset participate in finding matches. This reduces the number of entries in the hash table(s), and allows algorithms to accept longer inputs without the hash table(s) becoming overloaded. For the correcting 1.5-pass algorithm, the expected frequency of checkpoints is chosen so that the hash table contains a good representation of all sufficiently large regions of the reference string, although not necessarily of all offsets in the reference string. Because the correcting one-pass algorithm does not need to remember checkpoints over the entire reference string, but only those in some "window into the past", the frequency of checkpoints can be larger for the correcting one-pass algorithm than for the correcting 1.5-pass algorithm; this is discussed in Section 8.2.2.

Checkpointing allows an algorithm to reduce the input size by an arbitrary factor chosen so that the algorithm exhibits its best performance. We then need to address the issues of selecting checkpoints and integrating checkpointing into the existing algorithms.

8.1. SELECTING CHECKPOINTS. Let $\mathcal{C}$ be the set of checkpoint values and $\mathcal{F}$ be the set of footprint values, so $\mathcal{C} \subseteq \mathcal{F}$. Say that a seed is a *checkpoint seed* if the footprint of the seed belongs to the set $\mathcal{C}$ of checkpoints. Below, we describe a method for defining the set $\mathcal{C}$, testing whether a given footprint belongs to $\mathcal{C}$, and integrating checkpointing into differencing algorithms; the only details that the method needs are the number $|\mathcal{C}|$ of checkpoints and the number $|\mathcal{F}|$ of footprints. Because there is a one-to-one correspondence between checkpoint values and entries in the hash table, we choose $|\mathcal{C}|$ based on the amount of memory that we want to use for the hash table. The choice of $|\mathcal{F}|$ can depend on the differencing algorithm and the length of the input. For definiteness, we next describe a heuristic for choosing $|\mathcal{F}|$ that suits the needs of the correcting 1.5-pass algorithm.

The heuristic is to arrange things so that we expect the number of checkpoint seeds in R to be some chosen fraction of the number of entries in the hash table. For definiteness, we take this fraction to be 1/2 (the modifications needed for an arbitrary fraction will be obvious). This fraction is chosen to balance two competing goals: (i) to make it unlikely that two different checkpoint seeds that appear in R will collide (have the same footprint), so that the hash table will contain an entry for (almost) all of the checkpoint seeds that actually appear; and (ii) to utilize a significant fraction of the hash table.

Let L be the length of the string that is to be represented by entries in the hash table; in the present case, $L = |R|$. In this string, we expect there to be about $L \cdot |\mathcal{C}|/|\mathcal{F}|$ checkpoint seeds. Using the heuristic that this number should be about half the number of hash table entries, we want

$$L \cdot |\mathcal{C}|/|\mathcal{F}| \approx |\mathcal{C}|/2, \quad \text{that is,} \quad |\mathcal{F}| \approx 2L.$$

For example, since we are using a modular hash function to generate footprints, $|\mathcal{F}|$ equals the modulus q, so $q \approx 2L$ is used to select q.

We now turn to the process of deciding if a given footprint is a checkpoint. With the goals of efficiency and simplicity, we take every $(|\mathcal{F}|/|\mathcal{C}|)$th footprint to be a checkpoint. Let m be an integer close to $|\mathcal{F}|/|\mathcal{C}|$, either $m = \lceil |\mathcal{F}|/|\mathcal{C}| \rceil$ or $m = \lfloor |\mathcal{F}|/|\mathcal{C}| \rfloor$. Fix an integer k with $0 \leq k < m$. A given footprint $f \in F$ is a checkpoint if

$$f \equiv k \pmod{m}. \tag{3}$$

(The number of checkpoint values might be slightly smaller or larger than the original goal $|\mathcal{C}|$, depending on the choice of k and how $|\mathcal{F}|/|\mathcal{C}|$ is rounded to obtain m.)

To complete the determination of the checkpoints, k must be determined. One possibility is to choose k at random. A better method is to bias the random choice in favor of checkpoints that appear more often in V, because this gives a better expected coverage of V by checkpoints. To do this, choose a random offset a in V, compute the footprint f_a of a, and set $k = f_a \bmod m$. Consider, as a simple example, that $m = 2$, there are a large number ℓ of offsets a in V with $f_a \equiv 1 \bmod m$, and there are a relatively small number s with $f_a \equiv 0 \bmod m$. If k is chosen at random, the expected coverage is $E_1 = (\ell + s)/2$. If k is chosen by randomly choosing a as above, then the expected coverage is $E_2 = (\ell^2 + s^2)/(\ell + s)$. Now $E_2 - E_1 = (\ell - s)^2/(\ell + s)$. Because $\ell + s$ is a constant ($|V| - p + 1$), it follows that $E_2 - E_1$ increases as the square of $\ell - s$.

If a footprint f passes the checkpoint test (3), another computation must be done to map f uniquely to an integer i with $0 \leq i \leq |\mathcal{F}|/m$, which will be used to index the hash table. This is easily done by taking $i = \lfloor f/m \rfloor$.

Methods other than (3) are possible for defining the set of checkpoints. Assuming that the footprinting (hash) function produces a uniform distribution of footprints over its range $[0, |\mathcal{F}|)$, then once we have decided on the number of footprints and the number $|\mathcal{C}|$ of checkpoints, it should not matter what subset of (approximately) $|\mathcal{C}|$ footprints is defined to be the set of checkpoints. For example, we could divide the interval $[0, |\mathcal{F}|)$ into disjoint subintervals, each containing approximately $|\mathcal{C}|$ footprints, and use the biased random method to randomly choose one of these sub-intervals to be the set of checkpoints. This alternative requires one or two comparisons to test if a footprint is a checkpoint, and a checkpoint is mapped to an index by one subtraction.

Of course, nothing comes for free: checkpointing has a negative effect on the ability of the algorithm to detect short matching substrings between versions. If an algorithm is to detect and encode a matching substring, one of the seeds of this substring must be a checkpoint seed. Shorter matching substrings will have fewer checkpoint seeds. If the length of a "short" matching substring is small when compared to the distance between checkpoint seeds, $|\mathcal{F}|/|\mathcal{C}|$ on average, the short matching substring will likely be missed, since none of its seeds are checkpoint seeds. On the other hand, for versioned data, we expect highly correlated input strings and can expect long matching substrings that contain checkpoint seeds.

8.2. INTEGRATING CHECKPOINTS WITH DIFFERENCING ALGORITHMS. We perform checkpointing in an on-line fashion by testing for checkpoints at the same time that we generate footprints. For all of our algorithms described above, whenever a footprint f of an offset in string X is generated: (i) the algorithm might add

or replace an entry at index f in X's hash table, and (ii) the algorithm checks if f has a nonnull entry in the other string's hash table. In the algorithm modified with checkpointing, it first tests f (by (3)) to see if it is a checkpoint. If f is not a checkpoint, the algorithm does not perform (i), and it assumes that the test (ii) is false. If f is a checkpoint, f is converted to a hash table index i (by $i = \lfloor f/m \rfloor$), and (i) and (ii) are performed using i in place of f.

We also note that the checkpointing technique performs better with algorithms that employ correction. Since an algorithm using checkpointing does not remember offsets whose footprint is not a checkpoint, the algorithm is likely to miss the *starting* offsets of matching substrings. With correction, however, the algorithm can still find a matching substring if a checkpoint seed appears *anywhere* in the substring. With correction, missed starting offsets are handled transparently by backwards matching, and the algorithm finds the true starts of matching substrings without additional modifications to the algorithm. An algorithm that uses checkpointing but not correction would fail to copy-encode any portion of a matching substring prior to its first checkpoint seed, as it has no way to encode data prior to its current offset.

8.2.1. *Checkpointing and the Correcting 1.5-Pass Algorithm.* Checkpointing alleviates the breakdown of the correcting 1.5 pass algorithm operating on large inputs. Using the heuristic described above for choosing $|\mathcal{F}|$, the algorithm can fit an approximation of the contents of any reference string into its hash table. The argument behind the heuristic is probabilistic, and it is possible that some of our checkpoints will be very popular (appear at many offsets of R) or not appear at all. A checkpoint that does not appear is a loss, because an entry in the hash table is wasted. A frequently appearing checkpoint does not affect the time performance of the algorithm, because we store at most one offset for each checkpoint value. For the same reason, this could be detrimental to compression performance, because the offsets appearing in the hash table will be biased toward those near the beginning of R.

8.2.2. *Checkpointing and the Correcting One-Pass Algorithm.* The correcting one-pass algorithm has problems detecting distant matches when its hash table becomes over-utilized. This is not so much a mode of failure as a property of the algorithm. Applying checkpointing as we did in the correcting 1.5-pass algorithm allows such distant matches to be detected. In effect, checkpointing gives a "window into the past" that covers a larger portion of the past, but with a more sparse representation. Yet, if the version of the data does not exhibit transpositions, then checkpointing sacrifices the ability to detect short matches and gains no additional benefit.

With the correcting one-pass algorithm, the appropriate frequency of checkpoint seeds depends on the nature of the input data. For data without transpositions, checkpointing can be disregarded. Any policy decision as to the frequency of checkpoints is subject to differing performance, and the nature of the input data needs to be considered to formulate such a policy. In our opinion, it can rarely be correct to choose the frequency as small as was done in the heuristic for the correcting 1.5-pass algorithm, because the correcting one-pass algorithm will then never fill its hash tables and never use its full substring matching capabilities. Frequently occurring checkpoints are not as much of a problem for the correcting one-pass algorithm, because it constantly updates its hash tables (the window into the past).

Perhaps a more appropriate heuristic would be to choose enough checkpoints so that the window into the past covers some fraction of the input strings.

9. *Experimental Results*

We implemented the greedy, one-pass, correcting one-pass, and correcting 1.5-pass algorithms to evaluate their compression and running time performance experimentally. Checkpointing was used with the two correcting algorithms, but not with the one-pass or greedy algorithms. Experiments were performed on two types of input pairs (reference file, version file).[4] The first type consisted of two versions of actual files, such as two versions of software. This type generally consisted of pairs where the version file is highly correlated with the reference file (an earlier version). On inputs of this type, the two correcting algorithms achieve better compression than the one-pass algorithm, which confirms our intuition that correction (with checkpointing) helps compression. Another result is that the compression performance of the correcting algorithms is close to that of the greedy algorithm that achieves optimal compression. The results also indicate that the running times of the new algorithms grow linearly in the size of the files. Although the greedy algorithm uses more time than the new algorithms, it does not exhibit its worst-case quadratic growth on these highly correlated pairs. To evaluate the time performance of the algorithms on uncorrelated pairs, the algorithms were run on pairs of randomly generated files. Here, the running times of the new algorithms are linear, while the greedy algorithm does exhibit quadratic time. The experiments verify theoretical results about the algorithms. In particular, the one-pass algorithm runs in linear time, and the greedy algorithm uses quadratic time in the worst case. The experiments also support our assertion that the correcting one-pass algorithm and correcting 1.5-pass algorithms run in linear time, both on actual (correlated) versioned files and on uncorrelated files.

9.1. Experimental Setup.

All algorithms were implemented in the C programming language and were executed on an IBM PowerPC workstation running the AIX operating system. The workstation was an IBM model 43-P with a 133 MHz processor and 128 MB of RAM. Files were used as the input strings and all file I/O was performed through the local file system. Experiments were conducted out of a warm cache; that is, the input files were already resident in the computer's memory before measurements were taken. This eliminates the effect of caching on the measurement of running time. In particular, the first algorithm to run is not penalized for the extra time it spends conducting I/O from disk drives, rather than a cached copy of the data.

Our experiments measure both the time it takes an algorithm to execute and the compression that the algorithm achieves. Compression is measured as the ratio of the size of the output delta file to the size of the input version file (when defined this way, smaller compression is better). Running time is a measurement of the total execution time of the algorithm. This time includes time the algorithms spend waiting for file system I/O operations to complete.

[4] In this section, we use the term "file" rather than "string." A file is given as input to a differencing algorithm by regarding it as a string of bytes.

We gave all algorithms 64 kilobytes (KB) of memory for each of their hash tables. However, each entry of the hash table of the greedy algorithm is the head of a linked list containing all offsets with a given footprint. Thus, the total space used by the greedy algorithm's hash table structure is at least $|R|$, whereas that of the new algorithms is at most 128 KB. The significance of the amount of memory we chose is not in the total memory, but the ratio of the size of the memory to the size of the input. We chose 64 KB of memory for each hash table so that our input data included files that were more than ten times larger than the algorithm's fixed memory for the hash table used to store information about the seeds appearing in the file. The two correcting algorithms used a relatively small amount, 4 KB, of additional memory for the buffer (Section 5), divided into 256 entries of 16 bytes each. Each entry stored the information associated with one command; thus, the buffer could hold up to 256 commands at the same time. The length p of seeds was held fixed at $p = 16$. (Thus, the delta encodings found by the greedy algorithm are optimal given that matching substrings of length less than $p = 16$ are not found. To obtain a fair comparison of the new algorithms with the greedy algorithm, it is reasonable to use the same p for each.)

9.2. EXPERIMENTAL DATA. For experimental data, we used multiple versions of application and operating system software distributions that are commonly downloaded from the Internet. These distributions include multiple versions of the GNU tools and the BSD operating system distributions, among other data. The files consist of binary and executable files, source files, and documentation. It does not include image files, multimedia data, or databases. We were restricted to distributions of software for which multiple versions were available so that we could difference old files against new. In total, our data set consists of over 30,000 files. The average file size is approximately 180,000 bytes and the maximum file size is 900,000 bytes.

The files we ran experiments against were acquired from various Web sites that mirrored GNU and BSD distributions at various dates during 1996 and 1997. When we found a software distribution for which we already had an older copy, we would download and build that distribution. Then, we ran scripts that would find files having same path and name in the distribution and would compare them. We discarded all files with identical contents and all files not matched, leaving only files that were non-trivially modified between versions.[5]

9.3. COMPARISONS AGAINST SUFFIX TREES. We do not include a direct comparison against algorithms based on suffix-trees [Weiner 1973], because their space bounds are unreasonable for our intended applications. However, we do draw some conclusions based on results presented by Kurtz [1999]. A major goal of our differencing algorithms is to compress versions of data much larger than the amount of memory available. This cannot be realized using suffix trees. The most space-efficient implementation of a suffix tree considered in Kurtz [1999] builds a tree 10 times the size of the input, determined experimentally, and has an analytical bound of 16 times this input size.

When looking at other performance measures, suffix trees are still unattractive. All data in Kurtz [1999] indicate that suffix trees are substantially less time efficient

[5] These data will be made available by request from `randal@cs.jhu.edu`.

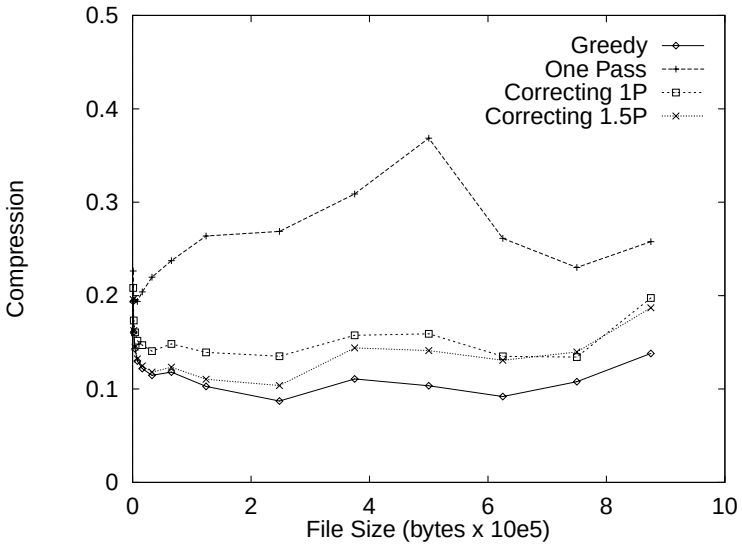


FIG. 9. Compression results on experimental data.

than our methods. On a machine over twice as fast (300 MHz vs. 133 MHz) and with more memory (192 MB vs. 128 MB), suffix trees can be constructed at an average rate of 0.4 MB/sec [Kurtz 1999] as compared to our algorithms, which produce output at an average rate of 0.5 MB/sec or better in our experiments. A delta algorithm based on suffix trees would use even more time, because it has to take a pass over the version file and output a delta after the suffix tree is constructed. Suffix trees compress data optimally, so the compression performance is equal to or slightly better than the greedy algorithm (recall that the greedy algorithm cannot detect matches smaller than the length of a seed). Most compression arises from long matching strings, so this difference is very small.[6]

While suffix trees are noteworthy for being the only known linear-time, optimally compressing delta algorithms, a small amount of compression loss can be traded for significant time and space performance improvements.

9.4. COMPRESSION RESULTS ON FILE DATA. Compression results from our experiments show the effectiveness of our algorithms in creating a compact differential encoding of a version file in terms of a previous reference file. Figure 9 compares the relative compression performance of the four implemented algorithms. This plot compares input file size on the X-axis and compression (size of the delta file divided by the size of the version file) on the Y-axis.

As shown in Section 3, the greedy algorithm provides optimally compressed deltas (under the simple cost measure). So, the curve for the greedy algorithm represents a lower bound on how much compression can be achieved. All other algorithms compress less well. The difference between the compression curve of an algorithm and the curve for the greedy algorithm describes how much compression the algorithm sacrifices to achieve its better time performance.

[6] We ran the greedy algorithm with seeds as small as 2 bytes and there was no detectable compression difference.

In general, the one-pass algorithm compresses input data significantly less well than the correcting algorithms and the greedy algorithm.

For smaller input files, the correcting 1.5-pass algorithm closely matches the (optimal) compression performance of the greedy algorithm. In this case, "small" means that file size is a small multiple of the amount of memory in which the algorithm operates. For small files, the checkpoints in the correcting 1.5-pass algorithm cover the input files densely. However, as the size of the input files increase, the checkpoints become more sparse, which adversely affects the compression. Recalling that the correcting 1.5-pass algorithm is a time-efficient variation of the greedy algorithm, we observe the similarities between the curves of the greedy and correcting 1.5-pass algorithms and conclude that a small amount of compression is lost to improve the time performance.

The correcting one-pass algorithm addresses the compression performance shortcomings of the one-pass algorithm. Its compression performance resembles that of the greedy and correcting 1.5-pass algorithms, more than that of the one-pass algorithm from which it is derived. The improved compression of the correcting one-pass algorithm is a testament to the effectiveness of correction and checkpointing. For small files, the correcting one-pass algorithm is outperformed by the correcting 1.5-pass algorithm, which collects more complete information about the reference file before encoding—recall that the one-pass algorithm goes through both input files simultaneously. Interestingly, the correcting one-pass algorithm performs almost identically to the correcting 1.5-pass algorithm on large files. The most plausible explanation for this is that the checkpoints in the one-pass algorithm can be taken more densely than for the 1.5-pass algorithm, because the one-pass algorithm does not need to process the whole reference file at one time. The more dense checkpoints find matching substrings that the relatively sparse checkpoints in the correcting 1.5-pass algorithm miss. It is the authors' intuition that this effect will be dependent upon the input data, but should be prevalent on large inputs.

9.5. RUNNING TIME RESULTS ON FILE DATA. From the compression experiments, we also extract running time information to compare the relative performance of all algorithms. The graph in Figure 10 describes our results. Again, the X-axis represents file size. The Y-axis represents the average amount of time that it takes for the algorithm to process a byte of data (seconds/byte). The seconds/byte metric is chosen because it normalizes the amount of time the algorithm uses with respect to file size, so that results from different file sizes may be compared. We also considered the inverse of our metric—data rate (bytes/second), but chose seconds/byte, as it allows us to easily correlate experimental results with asymptotic time bounds for algorithms.

The relative compression performance of the algorithms, from best to worse (see Figure 9), is: (1) greedy; (2) correcting 1.5-pass; (3) correcting one-pass; and (4) one-pass. The relative time performance of the algorithms, from best to worse (see Figure 10) is: (1) correcting one-pass; (2) correcting 1.5-pass; (3) one-pass; and (4) greedy. Thus, if we ignore the one-pass algorithm, we see the expected inverse correlation between compression and running time. (The one-pass algorithm suffers from not doing checkpointing.)

For all algorithms, on small files, the seconds/byte is significantly larger than that on larger files, due to certain start-up costs such as program load, initializing

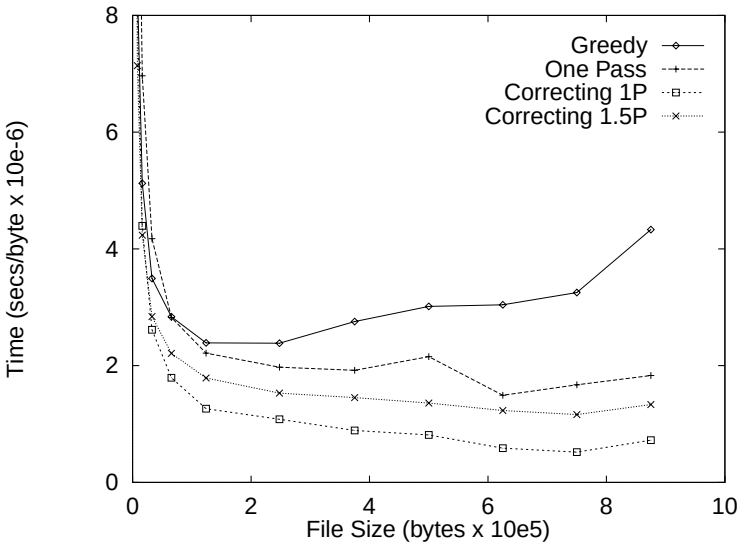


FIG. 10. Running time results on experimental data.

data structures, and memory allocation. For larger inputs, the total time over which an algorithm runs increases, so the start-up cost is amortized over more bytes.

The running time results show that the one-pass algorithm and the two correcting algorithms proceed through data at roughly a constant rate for files of all sizes. This represents linear asymptotic growth in the algorithms' times. Small variations in seconds/byte arise due to differences in the experimental data. The one-pass algorithm proceeds through data at different rates depending upon whether it is searching for matching strings or encoding matching data between versions (Section 4). So the actual time performance varies depending upon the compressibility of data and type of changes that occur.

We first consider the effect of correction by comparing the one-pass algorithm with the correcting algorithms. For larger files, both correcting algorithms use less time than the one-pass algorithm. Because the correcting algorithms use the checkpointing technique to reduce the number of offsets that they examine to find and encode matches, the seconds/byte decreases as files grow. That is, the larger the file, the larger the checkpoint interval, and the less dense the checkpoints in the file. For large files, checkpointing dominates performance, and the correcting algorithms outperform the one-pass algorithm. For small files, where checkpointing is not a factor, the performance of the one-pass algorithm is inferior to the performance of other algorithms because it produces significantly larger delta files. Poor compression performance causes the algorithm to perform significantly more I/O, increasing the running time.

Comparing now the correcting one-pass and the correcting 1.5-pass algorithms, we see that their running times are very similar, with the correcting one-pass algorithm having slightly faster running time than the correcting 1.5-pass algorithm at all points. The correcting one-pass algorithm has the advantage of being able to go through both files at the same time, as opposed to examining the whole reference file before encoding the version file. The slight time advantages of the correcting one-pass algorithm should be compared against the

slight compression advantages of the correcting 1.5-pass algorithm. Neither algorithm stands out as clearly superior; rather, the correcting one-pass algorithm has advantages for large inputs where its running time is superior, while the correcting 1.5-pass algorithm has compression advantages for small and moderate sized inputs.

The greedy algorithm has running time comparable to that of other algorithms for very small files, but for larger file sizes its running time is significantly worse. The greedy algorithm displays a moderate increase in seconds/byte running time as file size increases. However, the algorithm does not display a linear increase in seconds/byte running time that would result from quadratic growth of time versus file size. The greedy algorithm does not display quadratic time behavior here, because the experimental data is so compressible. For illustration, consider the greedy algorithm differencing two files that are identical. The algorithm builds its data structures in linear time,[7] finds a matching seed at the first offset of each file, extends this match to the end of both files, and encodes the match with a single copy command. This all occurs in $O(n)$ time.

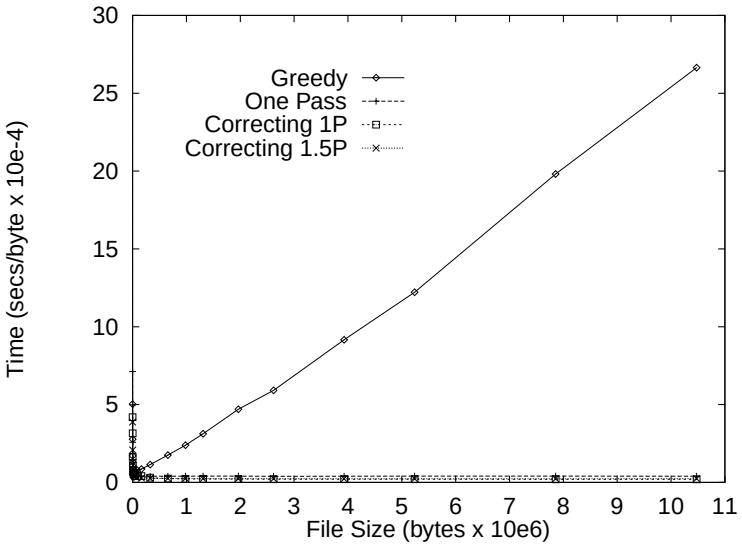
9.6. RUNNING TIME RESULTS ON UNCORRELATED VERSIONS. Almost all the file data used in our experiments shared the trait of similarity between versions. The version files were highly compressible using differencing algorithms, which indicates that they share many common substrings with their corresponding reference versions. To better understand the time performance of the four algorithms, and explore worst case behavior, we also ran the algorithms against uncorrelated files, two files that are likely to share only very short common substrings. We tested differencing on pairs of files that were roughly the same length, but whose bytes were random with respect to each other. To create uncorrelated files, we took a data string (any string would do), and encoded it using the PGP cryptography [Zimmerman 1995] package with two different keys. The output of this process is two strings of roughly the same length that are random with respect to each other and the input string. We created random files for many file sizes and ran our algorithms against them.

Since the strings were random, all four differencing algorithms were unable to achieve any compression. The size of the delta file was, without exception, the size of the version file plus a small encoding overhead.

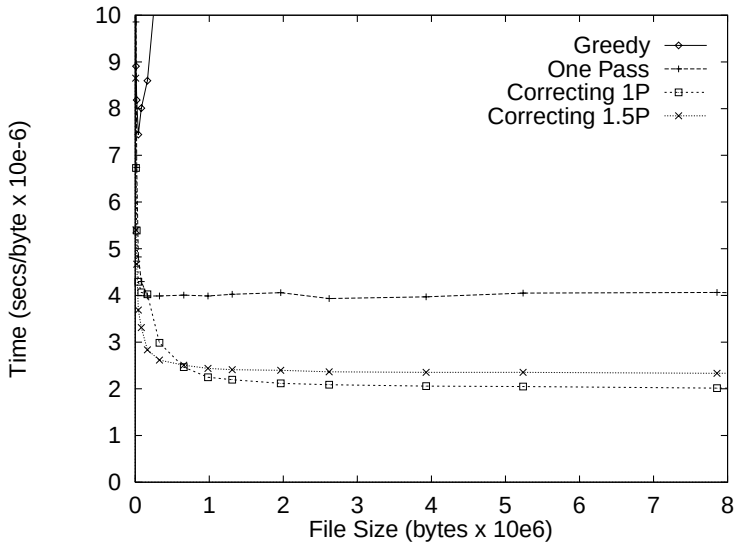
The running time results appear in Figure 11. In these graphs, the X-axis is file size and the Y-axis is seconds/byte. Since, we were generating data, we could choose the file size. However, each point does represent many samples at that size.

The one-pass algorithm proceeds through inputs of any size using the same seconds/byte, that is, at a constant rate. This is the expected result, since the one-pass algorithm examines every byte of the two files and, finding no matches, hashes the seed at that offset and places it in the hash table if that table entry is empty. File size has no effect on the seconds/byte performance.

[7] Our implementation of the greedy algorithm is based on an algorithm in the literature [Reichenberger 1991] that constructs its hash table using time in $\Omega(n^2)$. We modified the algorithm to construct the hash table in $O(n)$ time. This modification in no way changes the manner in which the algorithm finds and encodes matching substrings, so it does not affect the proof that it compresses optimally. This change makes for a more equitable comparison of a greedy algorithm with other algorithms.



(a) Full view of file size and time.



(b) Partial or zoom-in view of data.

FIG. 11. Running time results on random data.

The correcting algorithms both show a moderate decrease in seconds/byte as file size increases. This, just like the results on actual file data, can be attributed to checkpointing. As files grow larger, checkpoints are less dense, and the algorithm looks at fewer substrings in each region of the file.

The seconds/byte used by the greedy algorithm grows roughly linearly as file size increases. Linear growth in seconds/byte is equivalent to quadratic growth in time as file size increases, that is, $\Omega(n^2)$. Unlike the experiments on correlated files, on uncorrelated inputs the greedy algorithm shows its worst time behavior.

10. *A Limitation of Single-Pass Differencing Algorithms*

We had previously (Section 4) indicated that single-pass differencing algorithms have limited differencing power. We now present a lower bound on the length of the delta string $\Delta_{(R:V)}$ produced by single-pass differencing algorithms when the reference string has the form $R = XY$ and the version string has the form $V = YX$, where X and Y have the same length. This limits the differencing power of single-pass algorithms in the presence of transposed data and illuminates the limitations of any algorithm operating in a single pass over the input.

The definition of "single-pass" in this section is strict: each string is scanned by one pointer that can only move from the beginning of the string to the end; no random-access or reversal is allowed. This is a considerably more stringent restriction than the one used in the one-pass algorithm, because the one-pass algorithm can use random-access in its attempts to find matching substrings. The inputs R and V and the output delta are all binary strings. If n is the length of R (this is a different definition of n than the one used previously) and the algorithm has m bits of memory, the lower bound is $n/2 - m - O(\log n)$. As shown below, this lower bound is optimal to within an additive $O(\log n)$ term. In the strongest version of the result, the algorithm can be probabilistic, X and Y are random strings of length $n/2$, and the lower bound is on the expected length of $\Delta_{(R:V)}$. The lower bound holds for any encoding method having the property that V can be recovered from R and $\Delta_{(R:V)}$. One such method is the one described in Section 2.1.1, where V is encoded as a sequence of added substrings and substrings copied from R.

10.1. ENCODING METHODS. Let $\{0, 1\}^*$ denote the set of all binary strings, and let $\{0, 1\}^n$ denote the set of binary strings of length n. All logarithms are to the base 2.

An *encoding method* is a function δ that maps each pair R and V of binary strings to a nonempty set $\delta(R, V)$ of binary strings. Each string in $\delta(R, V)$ is an encoding of V with respect to R. In order that V be recoverable from R and every string in $\delta(R, V)$, we require that, for all R, V, V', if $V \neq V'$ then $\delta(R, V) \cap \delta(R, V') = \emptyset$.

10.2. MODEL OF COMPUTATION. A single-pass differencing algorithm is performed by an automaton, that we call a *single-pass automaton*, having two input tapes, a finite-state control, and a mechanism for producing an output string (described below). A few words should be said about the assumption of a finite-state control, even though the input strings could be arbitrarily large. In the definitions given next and in the statements of the results, we assume that the length of each input is some fixed (but arbitrary) even number n. The lower bound on the length of the output is stated in terms of n and the number of bits of memory in the finite state control, that is, the log of the number of states. Alternatively, the results could be stated in an "asymptotic" way, where the size of memory increases as some function of n. Stating the results as we do has the advantage that the results hold in a *nonuniform* model, where we can use a different differencing algorithm for each length n of the inputs; again the size of memory can increase as n increases. Because we are proving *lower bounds* on the length of the output, our results can only become stronger by using a more general model.

We first describe this type of automaton informally, and then give a more precise definition. One input tape (tape 1) holds the reference string R, and the other (tape 2) holds the version string V; each string is followed by an end-of-string

character denoted \$. Each input tape is scanned by a single reading head, which is initially positioned on the leftmost bit of the string. Each head on an input tape can move only from left to right. Typically, the output string of a finite-state machine is produced incrementally, by issuing the symbols of the string from left to right. We can use a more general type of output mechanism, which permits incremental changes other than appending a symbol at the end. For example, this mechanism allows a certain number of symbols to be erased from the end. In more detail, the finite-state control can issue *output actions*. Each output action is a function from binary strings to binary strings. Initially, the output string is the empty string. If the output action α is issued when the output string is w, the output string is changed to $\alpha(w)$. Without some restriction on the output actions, this model can compute any function $f(R, V)$ (e.g., the function computed by a perfect differencing algorithm): the automaton first copies the inputs R and V to the output string and then issues an output action that has the effect of applying f to R and V. A sufficient restriction for our purposes is that, for every output action α and every two strings u and w, if $|u| = |w|$, then $|\alpha(u)| = |\alpha(w)|$. Examples of output actions meeting this restriction are: for each positive integer i, the action α_i that erases i bits from the end of the string; and for each binary string y, the action α_y that appends y to the end of the string, that is, $\alpha_y(w) = wy$. To make our proof work using this general type of output mechanism, we must also assume an upper bound on the length of the output string at every point during execution of the algorithm. (In general, the output string at some intermediate point could be longer than the final output.) To simplify the statements of the results, we assume that the length of the output string never exceeds the length n of the inputs. As remarked below, we still get meaningful results if this upper bound is weakened to n^c for any constant c.

The definition of a single-pass automaton is now made more precise. A (*deterministic*) *single-pass automaton* D consists of a finite set Q of states, a finite set of output actions, and a transition function. Each output action is a function $\alpha : \{0,1\}^* \to \{0,1\}^*$ such that, for all $u, w \in \{0,1\}^*$, if $|u| = |w|$ then $|\alpha(u)| = |\alpha(w)|$. Recall that the two input tapes are numbered 1 and 2. The transition function maps each "condition" to a "move". A condition is of the form (q, s_1, s_2) where $q \in Q$ and $s_1, s_2 \in \{0, 1, \$\}$. This condition means that the automaton is in state q, and the head on tape i is reading the symbol s_i. The transition function maps each such condition to a move of the form (q', d_1, d_2, α) where $q' \in Q$, $d_1, d_2 \in \{0, 1\}$, and α is an output action. This move means to change the state to q', move the head on tape i one symbol to the right if $d_i = 1$ or leave it stationary if $d_i = 0$, and issue the output action α. (If a head is reading \$, it must remain stationary.) The set of states contains an initial state and a halting state. It is technically convenient to assume that exactly one of the heads moves at each step, unless both heads are reading \$ and the step causes the machine to enter the halting state without moving either head. It is easy to modify an automaton to one meeting this restriction while tripling the number of states, that is, adding at most 2 bits of memory (which can be absorbed in the $O(\log n)$ term): Whenever the original automaton would move both heads at the same step, the modified automaton moves the heads in two separate steps. Then steps that move neither of the heads can easily be eliminated.

The automaton is started on inputs R, V with the input head on tape 1 (respectively, tape 2) scanning the leftmost bit of R (respectively, V), and with the control in the initial state. Let $\alpha_1, \ldots, \alpha_t$ be the sequence of output actions

issued during the computation until the halting state is entered. The output, denoted $D(R, V)$, is $\alpha_t(\alpha_{t-1}(\cdots\alpha_1(\lambda))\cdots)$ where λ is the empty string. (The set of output actions can contain the identity function, which has the effect of a null action, so the automaton can make moves without changing the output string.)

Let n, m be positive integers. A single-pass automaton D is a *single-pass differencing algorithm for n-bit strings with m bits of memory* if

(1) there is an encoding method δ such that $D(R, V) \in \delta(R, V)$ for all $R, V \in \{0, 1\}^n$; equivalently, for all $R, V, V' \in \{0, 1\}^n$, if $V \neq V'$, then $D(R, V) \neq D(R, V')$;
(2) D has 2^m states; and
(3) for all $R, V \in \{0, 1\}^n$, if $\alpha_1, \ldots, \alpha_t$ is the sequence of output actions issued in the computation of D on inputs R and V, then for all i with $1 \leq i \leq t$, we have $|\alpha_i(\alpha_{i-1}(\cdots\alpha_1(\lambda))\cdots)| \leq n$.

The definitions of a *probabilistic* single-pass automaton and differencing algorithm are similar to those above. The difference in the definition of the automaton is that the transition function maps each condition to a probability distribution on the set of moves. It is useful to assume that all probabilities in these distributions are rational. Whenever a certain condition holds, the next move is chosen according to its associated probability distribution. Therefore, $D(R, V)$ is a random variable. The definition of a probabilistic single-pass differencing algorithm is identical to the definition above for the deterministic case, except that items (1) and (3) must hold with probability 1. For example, in item (1), the algorithm might produce different outputs depending on what random choices it makes, but every output that could possibly be produced must belong to $\delta(R, V)$.

10.3. LOWER BOUNDS. We prove lower bounds on $|D(XY, YX)|$ for all single-pass differencing algorithms D having sufficiently small memory. To introduce the proof method, a lower bound is first proved for the case that the algorithm is deterministic, and the lower bound is shown to hold for some input of the form $R = XY$ and $V = YX$ where X and Y are both of length $n/2$. By modifications to the first proof, we then prove a lower bound on the average length of the output when X and Y are chosen randomly and independently from the set of strings of length $n/2$, where the algorithm is still deterministic. This average-case result for deterministic algorithms is then used to obtain a lower bound on the average length of the output where the algorithm is probabilistic and X and Y are chosen at random.

10.3.1. *Worst-Case Bound for Deterministic Algorithms.*

THEOREM 10.1. *Let n, m be positive integers with n even, and let D be a single-pass differencing algorithm for n-bit strings with m bits of memory. There exist $X, Y \in \{0, 1\}^{n/2}$ such that*

$$|D(XY, YX)| \geq \frac{n}{2} - m - 2\log n - 2.$$

PROOF. For each pair (X, Y) with $X, Y \in \{0, 1\}^{n/2}$, perform the following simulation. Run D on input (X, Y) but do not place end-of-string symbols at the

ends of these strings. Continue the simulation until either (1) the head reading X moves off the right end of X, or (2) the head reading Y moves off the right end of Y. Exactly one of (1) or (2) must occur, because we have assumed that exactly one head moves at each step, and if D halts before either (1) or (2) occurs then $D(XW, YW) = D(XW, YW')$ for all $W, W' \in \{0,1\}^{n/2}$, contradicting that D is a differencing algorithm. Let F_1 (respectively, F_2) be the set of pairs (X, Y) such that (1) occurs before (2) (respectively, (2) occurs before (1)). The proof has two cases depending on whether F_1 or F_2 contains at least half of the pairs. (There are a total of 2^n pairs.)

Case 1. $|F_1| \geq 2^{n-1}$.

For each Y, let

$$S(Y) = \{ X \mid (X, Y) \in F_1 \}.$$

Since there are $2^{n/2}$ different Y's, there must be some Y with $|S(Y)| \geq |F_1|/2^{n/2} \geq 2^{n/2-1}$. For the remainder of this case, fix Y to be some string with this property. For each $X \in S(Y)$, let *div*(X) (for "dividing point") be the step at which the head reading X moves off the right end of X in the simulation on input (X, Y). For each $X \in S(Y)$, define *type*$(X) = (q, i, j)$ where, in the configuration of the automaton just after step *div*(X) in the simulation on input (X, Y), the state is q, the head on tape 2 is reading the ith symbol of Y, and the length of the current output string is j. Define an equivalence relation on $S(Y)$ by $X \equiv X'$ iff *type*$(X) =$ *type*(X'). Because a type is specified by a triple (q, i, j) where $1 \leq i \leq n/2$ and $0 \leq j \leq n$, and because there are 2^m states q, an upper bound on the number of equivalence classes is

$$k = 2^m \left(\frac{n}{2}\right)(n+1). \tag{4}$$

Note that if $X \equiv X'$, then the behavior of D on input (XY, YX) after step *div*(X) is identical to the behavior of D on input $(X'Y, YX)$ after step *div*(X'), because *type*(X) and *type*(X') are identical in their first component (the state), identical in their second component (the position of the head on the second input YX), and in both cases the head on tape 1 is scanning the first bit of Y in the first input, XY or $X'Y$. In particular, the two sequences of output actions are the same. Since *type*(X) and *type*(X') are identical in their third component (the length of the current output string) and since we have assumed that each output action maps strings of equal length to strings of equal length, we have the key fact that

$$X \equiv X' \text{ implies } |D(XY, YX)| = |D(X'Y, YX)|. \tag{5}$$

Because the equivalence classes partition $S(Y)$, a set of size at least $2^{n/2-1}$, there is some equivalence class C such that

$$|C| \geq \frac{2^{n/2-1}}{k}. \tag{6}$$

Fix an arbitrary $X' \in C$. Because D is a differencing algorithm, we know that $D(X'Y, YX_1) \neq D(X'Y, YX_2)$ for all $X_1, X_2 \in C$ with $X_1 \neq X_2$. Therefore, as X ranges over C, the output $D(X'Y, YX)$ ranges over $|C|$ different binary strings.

So there is some $X \in C$ with

$$|D(X'Y, YX)| \geq \lfloor \log |C| \rfloor \geq \log |C| - 1. \tag{7}$$

Because $X \equiv X'$, it follows from (5) that $|D(XY, YX)| = |D(X'Y, YX)|$. The lower bound on $|D(XY, YX)|$ stated in the theorem now follows because

$$\begin{aligned} |D(X'Y, YX)| &\geq \log |C| - 1 && \text{by (7)} \\ &\geq n/2 - \log k - 2 && \text{by (6)} \\ &\geq n/2 - m - 2\log n - 2 && \text{by (4).} \end{aligned}$$

In the third inequality, we use that $(n/2)(n+1) \leq n^2$ for all $n \geq 1$.

Case 2. $|F_2| > 2^{n-1}$.

This case is essentially symmetric to Case 1, interchanging the roles of X and Y. There is a small deviation from symmetry at the end of the argument, since the differencing problem is not completely symmetric between the two input strings.

Symmetrically, we find an X such that $|S(X)| \geq 2^{n/2-1}$, where $S(X) = \{ Y \mid (X, Y) \in F_2 \}$, and we define an equivalence relation on $S(X)$ such that $Y \equiv Y'$ implies $|D(XY, YX)| = |D(XY, Y'X)|$. Let C be an equivalence class satisfying (6). Fix an arbitrary $Y \in C$, and find $Y' \in C$ such that $|D(XY, Y'X)| \geq \lfloor \log |C| \rfloor$. The rest of the proof is the same as above. □

If we assume only a polynomial upper bound n^c on the length of the intermediate output string at all steps, then the upper bound on the number of equivalence classes becomes $k = 2^m(n/2)(n^c + 1)$, and the term $-2\log n$ in the lower bound becomes $-(c+1)\log n$ as can be seen from the three-line calculation above.

The lower bound of Theorem 10.1 is optimal to within an additive $O(\log n)$ term, as we now show by describing a strict single pass differencing algorithm that outputs an encoding of cost at most $n/2 - m + O(\log n)$ when $R = XY$, $V = YX$, $|X| = |Y|$, and $m \leq n/2$. We assume that the algorithm knows n and m; this is allowed by our formal model and typically holds in practice. Write $R = R_1R_2$ and $V = V_1V_2$, where $|R_1| = |R_2|$ and $|V_1| = |V_2|$. For a suitable $j = m - O(\log n)$, the algorithm starts by moving the R-pointer forward, while remembering the first j bits of R_1 and counting (using about $\log n$ bits of memory) to stop this pointer on the first symbol of R_2. It then moves both the R-pointer and the V-pointer forward, while comparing bits in V_1 with those in R_2. When the first mismatch is found, it outputs a copy command for the longest prefix of V_1 that matches a prefix of R_2, and encodes the rest of V_1 with an add command. It then compares the first j bits of V_2 with the (stored) first j bits of R_1, and encodes V_2 similarly, except that the prefix of V_2 that is encoded by a copy command can have length at most j. If $R = XY$, $V = YX$, and $|X| = |Y|$, the algorithm outputs an encoding of cost at most $n/2 - j + O(\log n) = n/2 - m + O(\log n)$.

10.3.2. *Average-Case Bound for Deterministic Algorithms.* For U a real-valued random variable, let $\mathbf{E}(U)$ denote its expected value.

THEOREM 10.2. *Let n, m be positive integers with n even, and let D be a single-pass differencing algorithm for n-bit strings with m bits of memory. If the random variables* $\mathbf{X}$ *and* $\mathbf{Y}$ *are uniformly and independently distributed over* $\{0,1\}^{n/2}$*, then*

$$\mathbf{E}(|D(\mathbf{XY}, \mathbf{YX})|) \geq \frac{n}{2} - m - 2\log n - 4.$$

PROOF. The proof is similar in many ways to the previous one, and we use the notation defined there. One difference is the following. In the previous proof, we used that if I is a set of t distinct binary strings, then some string in I must have length at least $\lfloor \log t \rfloor$. In the present proof, we need a lower bound on the sum of the lengths of the strings in I. A lower bound is

$$\sum_{w \in I} |w| \geq \sum_{i=1}^{t} \lfloor \log i \rfloor \geq \log(t!) - t \geq t(\log t - 3).$$

(The inequality $t! \geq t^t 2^{-2t}$ can be proved easily by induction on t.)

The other principal difference is the following. In the previous proof, we used several times that if a number N is partitioned into t parts, that is, $N = p_1 + p_2 + \cdots + p_t$, then some part p_i must be at least N/t. In the present proof, we use the following lemma to argue that there is no loss of generality in choosing *all* parts equal N/t. When we take the size of all parts to be the same, the algebraic expressions look very much like the ones in the previous proof.

LEMMA 10.3. *Let N, t be positive integers and let $p_1, \ldots, p_t$ be positive real numbers such that $p_1 + \cdots + p_t = N$. Then*

$$\sum_{i=1}^{t} p_i \log p_i \geq N \log (N/t).$$

PROOF. Because $x \log x$ is convex in $(0, \infty)$ (which is true because its second derivative exists and is nonnegative in $(0, \infty)$), the lemma follows easily from the basic fact that if ϕ is convex then $(\sum_{i=1}^{t} \phi(p_i))/t \geq \phi((\sum_{i=1}^{t} p_i)/t)$ (see, e.g., Hardy et al. [1964, Sect. 3.6]). By viewing each p_i/N as a probability, it also follows easily from the fundamental fact of information theory that the entropy of a discrete probability space having t sample points is at most $\log t$ [Gallager 1968, Theorem 2.3.1]. □

Let $P = \{ (X, Y) \mid X, Y \in \{0,1\}^{n/2} \}$. The quantity of interest is

$$\mathbf{E}(|D(\mathbf{XY}, \mathbf{YX})|) = 2^{-n} \cdot \sum_{(X,Y) \in P} |D(XY, YX)|. \tag{8}$$

To place a lower bound on the sum (8), the sum is broken into pieces. At the highest level, P is broken into F_1 and F_2. Consider the part of the sum that is over $(X, Y) \in F_1$. For each Y, define $S(Y) = \{ X \mid (X, Y) \in F_1 \}$ as before, and each $S(Y)$ is divided into equivalence classes as before. We restrict attention to those Y's such that $S(Y)$ is nonempty. Let $\mathcal{E}(Y)$ be the set of equivalence classes into which $S(Y)$ is divided. For each Y and each $C \in \mathcal{E}(Y)$, let X'_{YC} be some (arbitrary) member of C. As in the previous proof (see (5)), if $X \in C$, then $|D(XY, YX)| = |D(X'_{YC}Y, YX)|$. The part of the summation in (8) over $(X, Y) \in F_1$ equals (where the first summation is over those Y such that $S(Y)$ is nonempty)

$$\begin{aligned} \sum_{Y} \sum_{C \in \mathcal{E}(Y)} \sum_{X \in C} |D(XY, YX)| &= \sum_{Y} \sum_{C \in \mathcal{E}(Y)} \sum_{X \in C} |D(X'_{YC}Y, YX)| \\ &\geq \sum_{Y} \sum_{C \in \mathcal{E}(Y)} |C|(\log |C| - 3)\,. \end{aligned} \tag{9}$$

The inequality follows from the bound (shown above) on the sum of the lengths of $|C|$ distinct binary strings, since the strings $D(X'_{YC}Y, YX)$ for $X \in C$ are distinct. As before, an upper bound on the number of equivalence classes is $k = 2^m(n/2)(n+1)$. It is an easy fact that the sum of $|C|$ over $C \in \mathcal{E}(Y)$ equals $|S(Y)|$. Using Lemma 10.3, a lower bound on (9) is obtained by setting $|C|$ to $|S(Y)|/k$ for all $C \in \mathcal{E}(Y)$, giving the lower bound

$$\sum_Y |S(Y)| \left(\log \left(\frac{|S(Y)|}{k} \right) - 3 \right). \tag{10}$$

The sum of $|S(Y)|$ over $Y \in \{0,1\}^{n/2}$ equals $|F_1|$, and there are at most $2^{n/2}$ sets $S(Y)$. Again using Lemma 10.3, a lower bound on (10) is obtained by setting $|S(Y)|$ to $|F_1|/2^{n/2}$ for all Y, giving the lower bound

$$|F_1| \left(\log \left(\frac{|F_1|}{k2^{n/2}} \right) - 3 \right). \tag{11}$$

A symmetric argument is used for F_2, giving a lower bound (11) with F_1 replaced by F_2. Therefore,

$$\mathbf{E}(|D(\mathbf{XY}, \mathbf{YX})|) \geq 2^{-n} \left(\sum_{i=1}^{2} |F_i| \left(\log \left(\frac{|F_i|}{k2^{n/2}} \right) - 3 \right) \right). \tag{12}$$

Finally, again using Lemma 10.3, a lower bound on $\mathbf{E}(|D(XY, YX)|)$ is obtained by setting $|F_i| = 2^{n-1}$ $(i = 1, 2)$ in (12).

10.3.3. *Average-Case Bound for Probabilistic Algorithms.* The lower bound for probabilistic algorithms follows from the average-case lower bound of Theorem 10.2, using a method of Yao [1977]. To use Theorem 10.2, which was proved only for deterministic algorithms, a probabilistic algorithm is viewed as random choice of a deterministic algorithm.

LEMMA 10.4. *Let n, m be positive integers, and let P be a probabilistic single-pass differencing algorithm for n-bit strings with m bits of memory. There is a set $\{D_1, D_2, \ldots, D_z\}$ such that each D_i is a deterministic single-pass differencing algorithm for n-bit strings with $m + \lceil \log(n+1) \rceil + 1$ bits of memory, and for all $R, V \in \{0,1\}^n$, the random variable $P(R, V)$ is identical to the random variable $D_i(R, V)$ where i is chosen uniformly at random from $\{1, \ldots, z\}$.*

PROOF. Because we have assumed that exactly one of the heads advances by one at each step, except possibly the last step, it follows that P halts after taking at most $T = 2n + 1$ steps. We have defined a probabilistic single-pass automaton to have only rational transition probabilities. Let L be the least common multiple of the denominators of all the transition probabilities of P. It is useful to imagine that each deterministic algorithm in the set $\{D_1, \ldots, D_z\}$ is named by a sequence of T integers where each integer in the sequence lies between 1 and L. So $z = L^T$. Fixing some such sequence $\sigma = (r_0, \ldots, r_{T-1})$, we describe the algorithm D_σ named by it. The states of D_σ are of the form (q, t) where q is a state of P and $0 \leq t \leq T$. So D_σ has $2^m(2n+2)$ states. The second component t indicates how many steps have been taken in the computation. The algorithm D_σ uses r_t to determine its transitions from

states (q, t). For example, suppose that when seeing condition (q, s_1, s_2), algorithm P takes move $(q', d_1', d_2', \alpha')$ with probability $1/4$, or takes move $(q'', d_1'', d_2'', \alpha'')$ with probability $3/4$. Then when seeing condition $((q, t), s_1, s_2)$, algorithm D_σ takes move $((q', t+1), d_1', d_2', \alpha')$ if $1 \leq r_t \leq L/4$, or takes move $((q'', t+1), d_1'', d_2'', \alpha'')$ if $L/4 < r_t \leq L$. □

The next theorem follows easily from Theorem 10.2 and Lemma 10.4.

THEOREM 10.5. *Let n, m be positive integers with n even, and let P be a probabilistic single-pass differencing algorithm for n-bit strings with m bits of memory. If the random variables* **X** *and* **Y** *are uniformly and independently distributed over $\{0, 1\}^{n/2}$, then*

$$\mathbf{E}(|P(\mathbf{XY}, \mathbf{YX})|) \geq \frac{n}{2} - m - 3\log n - 6$$

where the expected value is with respect to **X**, **Y**, *and the random choices made by P.*

PROOF. Given P, find $\{D_1, D_2, \ldots, D_z\}$ from Lemma 10.4. Because Theorem 10.2 applies to each D_i, and each D_i has $m + \lceil \log(n+1) \rceil + 1 \leq m + \log n + 2$ bits of memory, it follows that for all i,

$$\mathbf{E}(|D_i(\mathbf{XY}, \mathbf{YX})|) \geq \frac{n}{2} - m - 3\log n - 6.$$

The lower bound on $\mathbf{E}(|P(\mathbf{XY}, \mathbf{YX})|)$ is now immediate, because $\mathbf{E}(|P(\mathbf{XY}, \mathbf{YX})|)$ equals the average of $\mathbf{E}(|D_i(\mathbf{XY}, \mathbf{YX})|)$ over all i. □

11. *Conclusions and Further Directions*

In Sections 4 through 8, we presented new algorithms and new techniques for the differencing problem. The new algorithms are the one-pass, correcting one-pass, and correcting 1.5-pass algorithms. The new techniques are correction and checkpointing. The main goal in these sections was to present algorithms that (i) operate at fine granularity and make no assumptions about the internal structure, such as alignment, of the data; (ii) scale well to large inputs; and (iii) give good compression in practice. All of these algorithms use constant space, and the one-pass algorithm uses linear time in the worst case. In experiments using actual versioned data, these new algorithms run in linear time and have compression performance that is close to optimal. In Section 10, we showed that strict single-pass algorithms with limited memory must have very poor compression performance on transposed data.

A basic theoretical question is the time-space complexity of perfect differencing, that is, the problem of finding a delta encoding (consisting of copy and add commands) of optimally small cost. For definiteness, and to make the question more tractable, say that the cost is given by the simple cost measure (Section 3). It is known that perfect differencing can be solved in time $O(n^2)$ and space $O(1)$ simultaneously (by the simple constant-space implementation of the greedy method (Section 3.2)), and in time $O(n)$ and space $O(n)$ simultaneously (by the suffix-tree implementation of the greedy method). This suggests that a time–space trade-off might hold. As a step toward establishing a trade-off, one could try to show that

perfect differencing cannot be solved in time $O(n)$ and space $o(n)$ simultaneously. The theorems of Section 10 imply that no strict single-pass (necessarily linear-time) algorithm with $o(n)$ memory can do perfect differencing. It would be interesting to show that no linear-time algorithm operating on a random-access register machine with $o(n)$ memory can do perfect differencing. Ajtai [1999] has recently proved results of this type for other problems.

Appendix

We describe the method used to encode a sequence of add and copy commands as a sequence of bytes. We use the term *codeword* for a sequence of bytes that describes a command; these codewords are variable length, and they are uniquely decodable (in fact, they form a prefix code). We separate codewords into three types: copy and add codewords, corresponding to copy and add commands, and the end codeword, which is a signal to stop reconstruction. The first byte of a codeword specifies the type of the command, and the rest contain the data need to complete that command. In some cases, the type also specifies the number and meaning of the remaining bytes in the codeword. A type has one of 256 values; these are summarized in Figure 12.

An *add codeword* contains an add type that, in some cases, specifies the length of the substring to be added. For types 247 and 248, the add type is followed by 16 and 32 bits, respectively, specifying the length of data in the added substring. The following bytes of the add codeword, up to the length specified, are interpreted as the substring that the add encodes.

A *copy codeword* contains a copy type, which specifies the number and form of the extra bytes used to describe the offset and length of the substring in the reference

END: $k = 0$. No more input.

ADD: $k \in [1, 248]$

$k \in [1, 246]$: The following k bytes are the bytes of a substring to be added at this point to the reconstructed string.

$k \in [247, 248]$: If $k = 247$ (resp., $k = 248$), the next 2 bytes (resp., 4 bytes) are an unsigned 16 bit short integer (an unsigned 32 bit integer) that specifies the number of following bytes that are the bytes of the substring to be added.

COPY: $k \in [249, 255]$. Copy codewords use unsigned bytes, shorts (2 bytes), integers (4 bytes) and longs (8 bytes) to represent the copy offset and length. A copy codeword indicates that the substring of the given length starting at the given offset in the reference string is to be copied to the reconstructed string.

k	offset	length
249	short	byte
250	short	short
251	short	int
252	int	byte
253	int	short
254	int	int
255	long	int

FIG. 12. The codeword type k specifies a codeword as an *add*, *copy* or *end* codeword. For copy and add codewords, the type specifies the number and meaning of the extension bytes that follow.

string that should be copied to the reconstructed string. A copy codeword requires no additional data, because the substring is contained within the reference string.

The *end codeword* is simply the end type 0; it has no additional data. It is basically a "halt" command to the reconstruction algorithm.

Let us provide a simple example. Assume that a delta string contains the bytes

$$3, 100, 101, 102, 250, 3, 120, 1, 232, 0.$$

The algorithm that reconstructs from this string will parse it as three codewords, namely

$$\{3, 100, 101, 102\}, \{250, 3, 120, 1, 232\}, \{0\}:$$

an add codeword of type 3, a copy codeword of type 250 that follows this add, and the end codeword that terminates the string. The add codeword specifies that the first three bytes of the reconstructed string consist of the substring 100, 101, 102. The copy codeword (type 250) specifies the use of 2 bytes to specify the offset (3, 120), and 2 bytes for the length (1, 232). It states that the next bytes in the reconstructed string are the same as the substring starting at offset 888 ($= 3 \cdot 256 + 120$) with length 488 ($= 1 \cdot 256 + 232$) bytes in the reference string. The end codeword 0 halts the reconstruction.

ACKNOWLEDGMENTS. We are grateful to Robert Morris, Norm Pass, and David Pease for their encouragement and advice. In particular, Norm Pass posed the problem to us of devising an efficient differential backup/restore scheme.

REFERENCES

AJTAI, M. 1999. Determinism versus non-determinism for linear time RAMs with memory restrictions. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing*. ACM, New York, 632–641.

BANGA, G., DOUGLIS, F., AND RABINOVICH, M. 1997. Optimistic deltas for WWW latency reduction. In *Proceedings of the 1997 USENIX Annual Technical Conference*. USENIX Association, Berkeley, Calif., 289–303.

BURNS, R. C., AND LONG, D. D. E. 1997. Efficient distributed backup and restore with delta compression. In *Proceedings of the 5th Workshop on I/O in Parallel and Distributed Systems*. ACM, New York.

BURNS, R. C., AND LONG, D. D. E. 1998. In-place reconstruction of delta compressed files. In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing*. ACM, New York.

CHAN, M., AND WOO, T. 1999. Cache-based compaction: A new technique for optimizing web transfer. In *Proceedings of the IEEE Infocom '99 Conference*. IEEE Computer Society Press, Los Alamitos, Calif.

CHAWATHE, S. S., AND GARCIA-MOLINA, H. 1997. Meaningful change detection in structured data. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*. ACM, New York.

DE JONG, S. P. 1972. Combining of changes to a source file. *IBM Tech. Discl. Bull. 15*, 4 (Sept.), 1186–1188.

GALLAGER, R. G. 1968. *Information Theory and Reliable Communication*. Wiley, New York.

GROSSI, R., AND VITTER, J. S. 2000. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing*. ACM, New York, 397–406.

GUSFIELD, D. 1997. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, New York.

HARDY, G. H., LITTELWOOD, J. E., AND PÓLYA, G. 1964. *Inequalities*. Cambridge University Press, London.

HUNT, J. J., VO, K.-P., AND TICHY, W. F. 1998. Delta algorithms: An empirical analysis. *ACM Trans. Softw. Eng. Method. 7*, 2, 192–214.

KARP, R. M., AND RABIN, M. O. 1987. Efficient randomized pattern-matching algorithms. *IBM J. Res. Devel. 31*, 2, 249–260.

KNUTH, D. E. 1973. *The Art of Computer Programming*, Volume 3, *Sorting and Searching*. Addison-Wesley, Reading, Mass.

KORN, D. G., AND VO, K.-P. 1999. The VCDIFF generic differencing and compression format. Tech. Rep. Internet-Draft draft-vo-vcdiff-00.

KURTZ, S. 1999. Reducing the space requirements of suffix trees. *Softw. Pract. Exper. 29*, 13, 1149–1171.

MACDONALD, J. P. 2000. File system support for delta compression. Masters thesis. Department of Electrical Engineering and Computer Science, University of California at Berkeley, Berkeley, Calif.

MILLER, W., AND MYERS, E. W. 1985. A file comparison program. *Softw. Pract. Exper. 15*, 11 (Nov.), 1025–1040.

MOGUL, J. C., DOUGLIS, F., FELDMAN, A., AND KRISHNAMURTHY, B. 1997. Potential benefits of delta encoding and data compression for HTTP. In *Proceedings of ACM SIGCOMM '97*. ACM, New York.

REICHENBERGER, C. 1991. Delta storage for arbitrary non-text files. In *Proceedings of the 3rd International Workshop on Software Configuration Management*. ACM, New York, 144–152.

ROCHKIND, M. J. 1975. The source code control system. *IEEE Trans. Softw. Eng. SE-1*, 4 (Dec.), 364–370.

TICHY, W. F. 1984. The string-to-string correction problem with block move. *ACM Trans. Comput. 2*, 4 (Nov.), 309–321.

TICHY, W. F. 1985. RCS—A system for version control. *Softw. Pract. Exper. 15*, 7 (July), 637–654.

TUDOR, P. N. 1995. MPEG-2 video compression. *Elect. Commun. Eng. J. 7*, 6 (Dec.), 257–264.

WAGNER, R. A., AND FISCHER, M. J. 1973. The string-to-string correction problem. *J. ACM 21*, 1 (Jan.), 168–173.

WEINER, P. 1973. Linear pattern matching algorithms. In *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory*. IEEE Computer Society Press, Los Alamitos, Calif., 1–11.

YAO, A. C. 1977. Probabilistic computation: towards a unified measure of complexity. In *Proceedings of the 18th Annual IEEE Symposium on Foundations of Computer Science*. IEEE Computer Society Press, Los Alamitos, Calif., 222–227.

ZIMMERMAN, P. 1995. *PGP Source Code and Internals*. MIT Press, Cambridge, Mass.

ZIV, J., AND LEMPEL, A. 1977. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory 23*, 3 (May), 337–343.

ZIV, J., AND LEMPEL, A. 1978. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inf. Theory 24*, 5 (Sept.), 530–536.

RECEIVED APRIL 2000; REVISED MAY 2002; ACCEPTED MAY 2002