Delta Algorithms: An Empirical Analysis

JAMES J. HUNT
University of Karlsruhe
KIEM-PHONG VO
AT&T Laboratories
and
WALTER F. TICHY
University of Karlsruhe

Delta algorithms compress data by encoding one file in terms of another. This type of compression is useful in a number of situations: storing multiple versions of data, displaying differences, merging changes, distributing updates, storing backups, transmitting video sequences, and others. This article studies the performance parameters of several delta algorithms, using a benchmark of over 1,300 pairs of files taken from two successive releases of GNU software. Results indicate that modern delta compression algorithms based on Ziv-Lempel techniques significantly outperform *diff*, a popular but older delta compressor, in terms of compression ratio. The modern compressors also correlate better with the actual difference between files without sacrificing performance.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution and Maintenance—*version control*; D.2.8 [**Software Engineering**]: Metrics—*performance measures*; E.4 [**Data**]: Coding and Information Theory—*data compaction and compression*; E.5 [**Data**]: Files—*backup/recovery*

General Terms: Algorithms, Experimentation, Measurement, Performance

Additional Key Words and Phrases: Benchmark, delta encoding, differencing

1. INTRODUCTION

Delta algorithms, i.e., algorithms that compute differences between two files or strings, have a number of uses when multiple versions of data objects must be stored, transmitted, or processed. The major early applica-

An earlier version of this article appeared in the Proceedings of the ICSE'96 SCM-6 Workshop (March 25–26, 1996), pp. 49–66.

Authors' addresses: J. J. Hunt and W. F. Tichy, Institut fuer Programmstrukturen und Datenorganisation, University of Karlsruhe, Am Fasanengarten 5, Postfach 6980, D-76128 Karlsruhe Germany; K.-P. Vo, Research, AT&T Laboratories, 180 Park Avenue, Florham Park, NJ 07932 .

tion of delta algorithms occured in revision control systems such as SCCS and RCS [Rochkind 1975; Tichy 1985]. By storing deltas relative to a base revision, these systems save substantial amounts of disk space compared with storing every revision in its entirety. Much less information need be stored for each revision because changes from one revision to the next are typically small. Other well-known applications are the display of differences between files and the merging together of the changes in two different files relative to a common base.

The classic program for generating deltas is Unix *diff* [Hunt et al. 1976; 1977]. Both SCCS and RCS use it for storage and display of differences; RCS also uses it for merging. Since *diff* is limited to text files, so are SCCS and RCS. However, users wish to place binary code under revision control as well, not just source text. A simple technique is to map the binary code into text and then apply *diff*. While this works reliably and is widely used in practice, the deltas produced are typically larger than the originals! Newer algorithms such as *bdiff* [Tichy 1984] and *suff* [Obst 1987] do not exhibit this problem.

Today, binary differencing capability has become mandatory. There are now many binary source formats that users need to manage, such as word processor files, spreadsheet data, electrical and mechanical CAD data, audio, and images. Thus, revision control system must handle binary files in a space efficient way. The display of differences and merging is also more difficult for these complex formats than for simple text.

Compression not only saves space, but also reduces I/O and thus can speed up programs. Compressing I/O is interesting because the speed differential between processors and bulk storage devices is about three orders of magnitude and increasing. Reading a delta from disk can be faster than reading the corresponding full file, even with the decompression time added in.

Deltas have found a number of applications outside revision control. For example, backup programs save space by storing deltas. Checkpoints of large data spaces can be compressed dramatically with deltas and can then be reloaded rapidly. Display updates can be performed efficiently using a delta that moves lines or rectangles around on the display [Gosling 1981]. Deltas are also needed for sequence comparison in molecular biology.

The recent explosive expansion of the World Wide Web has accentuated another use of deltas: they can be used for distributing updates for software and other data. Besides saving time, space, and network bandwidth, distributing deltas has another interesting benefit: a delta is effectively an encrypted form of the new version. It can only be decoded if the original is available. Thus, software updates in the form of deltas could be made available while strongly reducing the threat of piracy. Of course, revision control is also needed for simplifying web maintenance and providing access to old web pages.

Obviously, choosing the best delta algorithm for any given application requires reliable and comparable information about the performance of each algorithm; however, the state of empirical comparisons of delta algorithms is poor. Miller and Myers [1985] compare the runtime of their delta program, *fcomp*, with that of Unix *diff*. Their first test involves two pairs of (highly atypical) files, and *fcomp* fails on one of them. Additional tests were run, but not enough particulars are given to repeat the tests independently. Obst [1987] compares several delta algorithms on programs of about 3MB. No details are given that would permit the repetition of their experiment. In both instances, the claims are quite dubious. The requirements of a proper benchmark—a well-defined metric and a large, accessible, domain-appropriate data set—are not met. Nor are statistical methods brought to bear on the results. The unreliability of the observations is underscored by outliers and irregularities.

The authors' goals in this article are threefold: to provide a metric for comparing delta algorithms objectively, to suggest an appropriate benchmark for the comparison, and to present the results of applying the metric and benchmark to an interesting set of delta algorithms, i.e., Unix *diff* and two modern delta compressors. Results are presented for text, object code, and a mixture of binary code and text and report compression ratio, compression speed, and decompression speed. Enough information is given so that anyone can repeat the experiment.

2. METRIC

Designing a metric for comparing delta algorithms objectively is not as easy as it sounds. While it is straightforward to measure time, measuring compression ratio is more difficult. It is not enough to simply use a set of file pairs and determine the cumulative size of the deltas. Such summary information is not helpful without knowledge about the characteristics of the differences and the content of files being compressed. For instance, how large is a delta if the difference is small? What happens if the difference is (relatively) large, but the file size is small? In other words, how well does the delta size track the actual difference for various file sizes? How do different formats such as text, binary code, image data, etc., affect the delta size? A new metric based on the Longest Common Subsequence (LCS) addresses these concerns.

The performance of a delta algorithm depends on the size of the difference between pairs of files. The authors propose using the LCS as the reference against which to measure compression. This metric applies mainly to one-dimensional data, but it may also apply to higher-dimensional data that can be linearized without fragmenting typical changes.

Given two strings of characters, an LCS is a longest sequence of characters that is contained within both strings. The characters in the LCS need not be contiguous in the two strings. Thus, the length of the LCS is a good measure of the commonality of two strings. The **difference** between two files can then be expressed as the average size of the two files[1] minus the size of the LCS (all in bytes):

[1] The average of the sizes of both files is used to eliminate any bias in the direction in which a delta is computed.

$$\text{difference} = \frac{\textbf{size}(\textbf{file}_1) + \textbf{size}(\textbf{file}_2)}{2} - \textbf{size}(\text{LCS}) \qquad (1)$$

An LCS is computed by an algorithm using dynamic programming [Hunt et al. 1977; Nakatsu et al. 1982]. The algorithm's runtime is $O(nm)$ where n and m are the files sizes, so it is not practical for general use. Practical delta algorithms find approximations of the LCS or use other techniques to find common segments of strings.

3. BENCHMARK

Though a metric is useful for providing a basis of analysis, truly meaningful comparison requires a standard data set or benchmark as well. This benchmark combined with an objective metric makes the comparison repeatable and the results verifiable. Repeatability is one of the essential ingredients of scientific experiments. Results that cannot be reproduced in an independent trial are not trustworthy.[2]

The first problem encountered when defining a benchmark is finding an appropriate data set that is both large enough for the results to be statistically significant and representative of real-world applications. For delta algorithms, the most important quality of any benchmark is that it contain a wide spectrum of change examples. This means that both the size of the changes represented and the size of the files involved should vary considerably. Large changes on small files and small changes on large files should be included as well as small changes on small files and large changes on large files.

Furthermore, the benchmark should contain a variety of formats, particularly, pure text, pure object code, and pseudotext. Pseudotext is what word processors produce: stretches of text with interspersed binary data and infrequent line breaks. Pseudotext and binary formats have become more important as word processors, spread sheets, and multimedia data have become prevalent.

Data sets that meet these requirements can be found in the GNU software. Thanks to the Free Software Foundation, quite a number of software projects are available in successive versions. The authors chose two versions of GNU *emacs*—19.28 and 19.29—and two versions of GNU *gcc*—2.7.0 and 2.7.1—as their test suite. These versions provide a broad spectrum of variation between one revision of any given file and the next. Files range from 0 to over 200KB in size with differences from 0 to 90%. The benchmark contains 810 text files (C programs, Lisp programs, documentation) and 300 files with Lisp byte code (a pseudotext format). The authors also compiled the 201 C program files present in both versions and included the resulting object code in the study. The total comes to 1344 file pairs.

[2] The history of benchmarking teaches that benchmarks cannot be completely static; they must be extended from time to time to prevent overfitting of algorithms to their benchmark.

Another important advantage of GNU software is that it is freely available. Thus, it is easy to confirm or reject our results independently. It also allows the developers of new delta algorithms to compare their algorithms with those presented below without duplicating this entire study.

A potential problem with GNU software is that it was produced under unique circumstances. However, we are not claiming that the GNU software is representative for all development environments. Since we report results dependent on the variables format, change size, and file size, it is unimportant what mix of these variables is actually present in the GNU software. Thus, our results permit easy determination of an appropriate algorithm for given circumstances: simply select the ranges of the variables appropriate for a given environment and read-off the performance of the algorithms.

4. ALGORITHMS

Three delta algorithms are examined in this study: Unix *diff*, *bdiff*, and *vdelta*.[3] Unix *diff* is still the most widely used delta algorithm in revision control and configuration management systems. It finds an approximation of the LCS by considering whole lines instead of characters as indivisible units [Hunt et al. 1976]. It is much faster than computing a byte-level LCS because it does not examine all possible combinations of byte positions. However, only common lines can be found with *diff*.

As stated above, revision control for binary data has become essential, but *diff* can only handle text files. A simple solution to this problem is to map binary files to text before handing them to *diff*. The most widely available program used for this conversion is *uuencode*. Since *uuencode* recodes data with fixed line lengths, it is not hard to convince oneself that it is a poor encoding choice. One could come up with a better solution, but that is not the purpose of this study. Instead the *uuencode* + *diff* combination is examined because it is frequently used: it is used in at least one commercial product and the widely available RCS. It is important to include the *uuencode* + *diff* results to show how poor a choice *uuencode* actually is. Folding binary files into the text range is not necessary for any of the other algorithms, so *uuencode* is only used with *diff*.

The other two algorithms—*bdiff* and *vdelta*—piece together the second file out of blocks from the first file. Unlike *diff*, these algorithms are applicable to any byte stream. They exploit reordering of blocks to produce short differences. Further details about *bdiff* and *vdelta* are given in the appendices.

Bdiff and *vdelta* offer additional compression on the resultant delta. For this reason, *diff* coupled with *gzip* postprocessing is included in the study as well. All algorithms run enough faster than a byte-level LCS computation to have practical applications.

[3] Vdelta is described in Appendix B.

Table I. Software System Examples: GNU Emacs

	Files	Size 19.28	Size 19.29	LCS	Changed	Ratio (%)
Text	640	16,640,810	17,538,744	14,966,051	2,572,693	15.5
ELC	305	4,287,642	4,674,901	3,519,338	1,155,563	27.0
Object	60	3,035,984	3,441,120	2,598,887	842,233	27.7
Totals	1,005	23,964,436	25,654,765	21,084,276	4,570,489	19.1

Table II. Software System Examples: GNU GCC

	Files	Size 2.7.0	Size 2.7.1	LCS	Changed	Ratio (%)
Text	196	12,319,992	12,422,167	12,198,290	223,877	1.8
Object	143	10,818,472	10,888,376	10,158,623	729,753	6.7
Totals	339	23,138,464	23,310,543	22,356,913	953,630	4.1

Both *Bdiff* and *vdelta* are comparable in utility to *diff*. Not only do they produce deltas suitable for compression and three-way file merging, but their output can also be used to display differences. Since *bdiff* and *vdelta* break lines apart, minor postprocessing of the deltas is needed to produce output identical to *diff*'s. Other human-readable output that takes advantage of the finer granularity of the output of these algorithms can be produced using color-coding techniques. In addition, both *bdiff* and *vdelta* can compress a single file with itself.

5. SUMMARY RESULTS

Before delving into the details of the experiment, one can get a reasonable sense of performance from a simple overview. A delta is often one to two orders of magnitude smaller than the original and significantly smaller than a direct compression of the original. For example, Tables I and II summarize the differences between two versions of the two different GNU projects mentioned above: GNU *emacs*, releases 19.28 and 19.29, and of GNU *gcc*, releases 2.7.0 and 2.7.1.

The data are divided into text and objects code files. For *emacs* there is an additional row for byte-compiled Lisp (ELC) files. The column headed with *Changed* in Tables I and II is the size of what was changed from or added to the first release to obtain the second. One can see that quite large space savings can be obtained simply by storing a reference release and a sequence of changes instead of a full copy of each release.

One would expect that the better the delta algorithm is, the closer the delta produced would be to the LCS size. Tables III and IV give summary figures for all the delta algorithms. The *ratio* is computed as a percentage of the resultant delta and the size of the second version. The output from *diff* is always longer than the LCS-based delta. Furthermore, binary files, which are preprocessed with *uuencode* before *diff* is applied, are significantly longer than the originals in two of the three cases in Tables III and IV. Even with *gzip* postcompression, the resultant deltas from *diff* are still

Table III. Results Summary: GNU Emacs

		Changed	diff	diff+gzip	bdiff	vdelta
Text	Size	2,572,693	4,509,529	1,530,684	1,528,763	1,277,287
	Ratio	15.5%	27.1%	9.2%	9.2%	7.7%
ELC	Size	1,155,563	6,359,727	2,697,410	574,216	542,577
	Ratio	27.0%	148.3%	62.9%	13.4%	12.7%
Object	Size	842,233	4,313,537	1,579,490	962,562	1,000,759
	Ratio	27.7%	142.1%	52.0%	31.7%	33.0%
Totals	Size	4,570,489	15,182,793	5,807,584	3,065,541	2,820,623
	Ratio	19.1%	63.4%	24.2%	12.8%	11.8%

Table IV. Results Summary: GNU GCC

		Changed	diff	diff + gzip	bdiff	vdelta
Text	Size	223,877	647,079	175,300	140,549	146,204
	Ratio	1.8%	5.3%	1.4%	1.1%	1.2%
Object	Size	729,753	9,746,789	3,346,493	1,359,273	1,536,779
	Ratio	6.7%	90.1%	30.9%	12.6%	14.2%
Totals	Size	953,630	10,393,868	3,521,793	1,499,822	1,682,983
	Ratio	4.1%	44.9%	15.2%	6.5%	7.3%

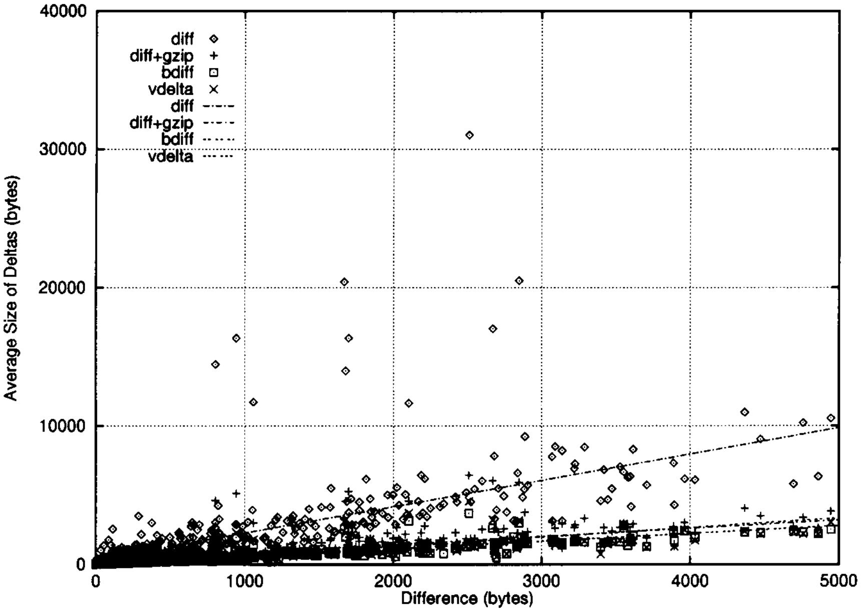
significantly worse than *bdiff* and *vdelta*. The deltas from *bdiff* and *vdelta* are very close in compression ratio overall. *Vdelta* seems to compress text data better, and *bdiff* appears to compress object code better. One can see that in some cases, *bdiff* and *vdelta* outperform LCS. That is because the algorithms postcompress the resultant delta. Tables III and IV just give a broad overview of the Emacs and GCC results. How the compression rate depends on the actual difference between the files being compared is not apparent. For this reason, it is important to investigate how the delta of each algorithm compares with the LCS size as well as how fast deltas can be computed and what effects additional compression might have.

6. METHOD

In order to obtain a more detailed view of the performance of the various delta algorithms, a comparison was done based on the difference metric defined in Section 2. As above, the Longest Common Subsequence was computed for each pair. Then each algorithm was run on each pair of files. This procedure was repeated for the compiled C code of both versions of GNU *emacs* and GNU *gcc*. Files that existed in one version and not the other and files that did not differ at all were eliminated. All computation was carried out on a DEC Alpha system.

The precise combination of algorithms chosen were Unix *diff -n* (as used by RCS),[4] Unix *diff* followed by compressing the results with *gzip*, *bdiff*, and *vdelta*. The files were broken up into three types: text files (mostly C

[4] Unlike the default and -c output, the -n output of *diff* is quite dense, as it is not designed for human consumption, but rather as a script for automatic processing.



Correlations: diff 0.952; diff+gzip 0.976; bdiff 0.962; vdelta 0.976

Fig. 1. Plot of delta size for text files.

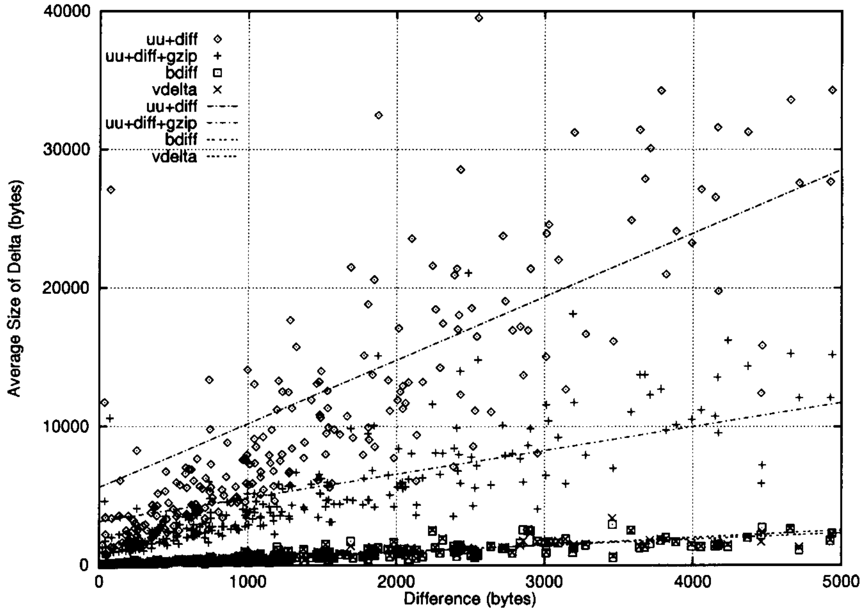
and Elisp code), byte-compiled Elisp code (ELC files), and object files. As stated above, since *diff* was designed to work only with text files, *uuencode* was used to convert ELC and binary files to text.

Each algorithm ran with each file pair both forward and reverse, e.g., revision 19.28 then 19.29 and revision 19.29 then 19.28. Each forward and reverse pair was then averaged together to give a single value for comparison. This procedure averages out the effect on differences where one file is much smaller than the other. Removing large sections from one file results in a small delta, and adding large sections results in a large delta. In practice, this phenomena is also averaged out in revisioning systems, since one revision must be stored in its entirety.

Two types of data were collected: the size of the delta file and the time needed to encode and decode each pair. Since no dedicated decoder is available for Unix *diff -n* files, RCS was used to time *diff -n* encoding and decoding. (The authors tried using *diff -e* and *ed*, but that proved to be ridiculously slow.) Unix *wc* measured file sizes and Unix time measure duration. The results report byte count and user plus system times.

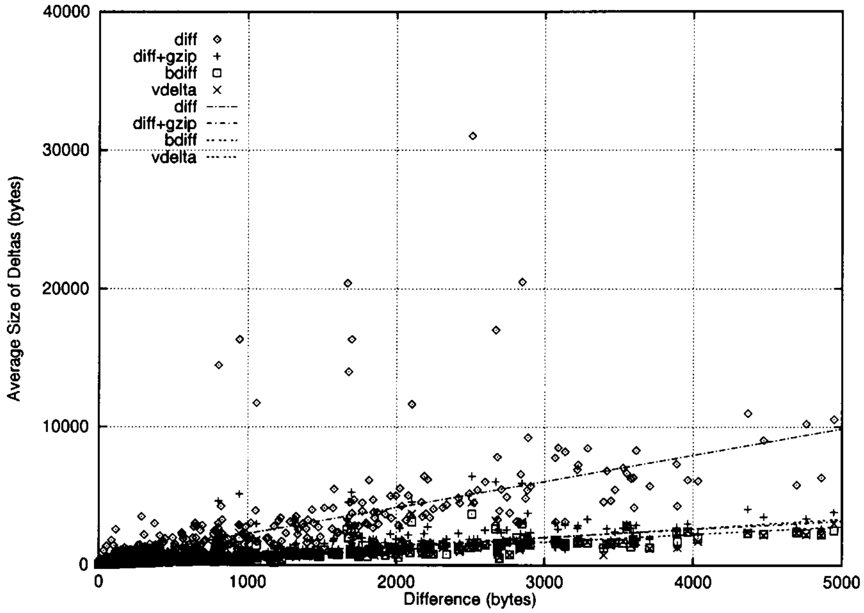
7. DETAILED RESULTS

There are two important measures for compression algorithms: the resultant compression ratio and the execution speed. The authors present the results of their study in graphic form below. Results are given separately



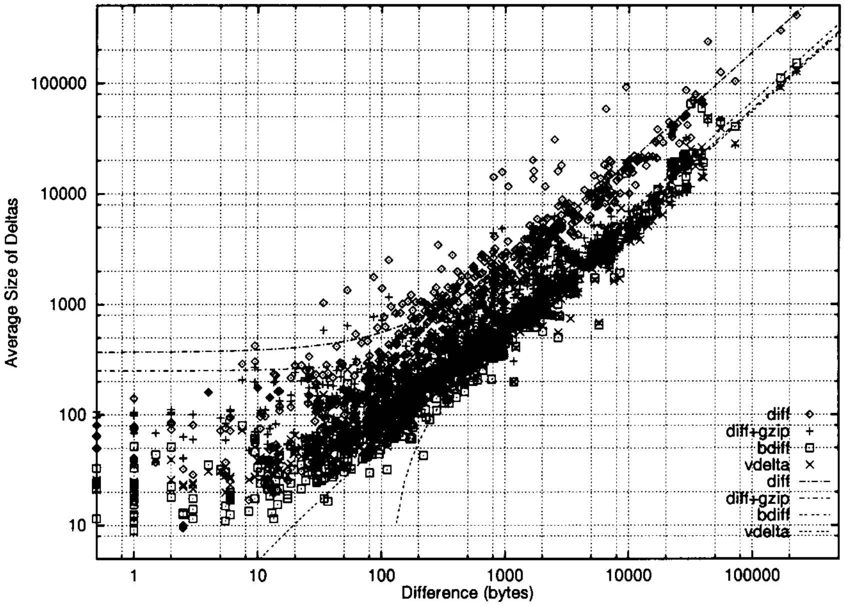
Correlations: uu+diff 0.805; uu+diff+gzip 0.804; bdiff 0.949; vdelta 0.959

Fig. 2. Plot of delta size for ELC files.



Correlations: uu+diff 0.847; uu+diff+gzip 0.857; bdiff 0.959; vdelta 0.937

Fig. 3. Plot of delta size for object files.



Correlations: diff 0.952; diff+gzip 0.976; bdiff 0.962; vdelta 0.976

Fig. 4. Log plot of delta sie for text files.

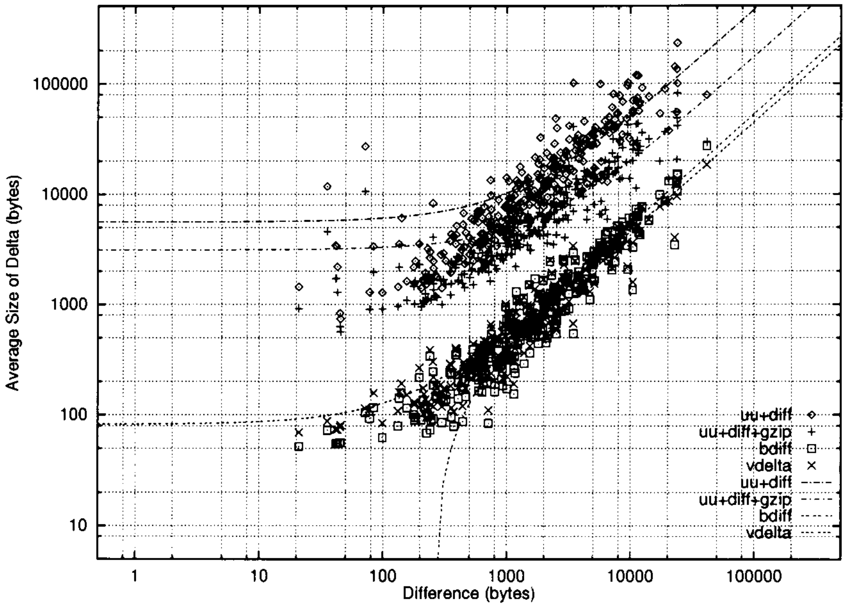
for text files, ELC files, and object files, since the behavior of some of the algorithms differ for these classes. ELC files were not combined with object files because their content is mostly text and thus similar to many common word processor formats.

7.1 Compression

The metric in Section 2 is the basis for measuring differences here. An alternative would have been to just examine file sizes and compare them to the sum of the delta sizes. Although this number is also interesting, the LCS comparison used here sheds more light on the correlation between the actual changes and the size of the delta. Each graph contains a point for each file pair and each algorithm. The overall trend of the data is illustrated with a linear regression line for each algorithm. The correlation coefficients provide the reader with an indication of how linear each of the depicted compression relationships are.[5]

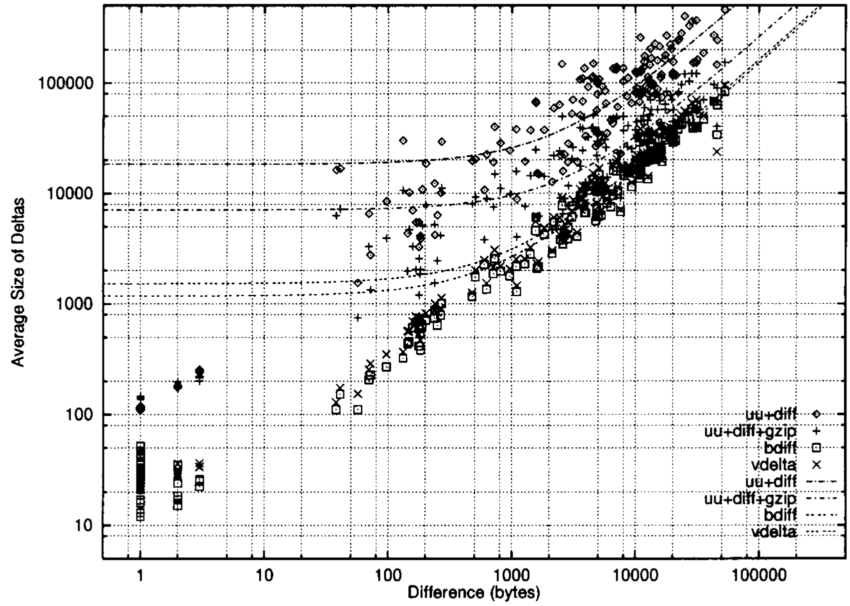
In the first set of three graphs—Figures 1, 2, and 3—the average size of the forward delta and reverse delta is plotted against the difference. The x-axis is simply the average size of the two files in each pair minus the LCS size. The y-axis is the average size of the forward and reverse delta for each

[5] The correlation coefficient does not indicate how good the algorithm is, rather how consistent the compression is with the actual difference between versions.



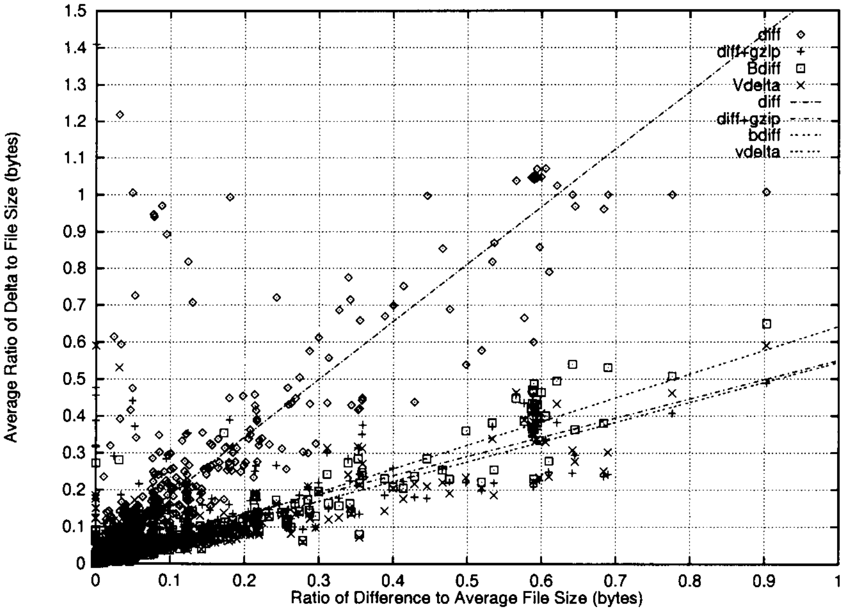
Correlations: uu+diff 0.805; uu+diff+gzip 0.804; bdiff 0.949; vdelta 0.959

Fig. 5. Log plot of delta size for ELC files.



Correlations: uu+diff 0.847; uu+diff+gzip 0.857; bdiff 0.959; vdelta 0.937

Fig. 6. Log plot of delta size for object files.



Correlations: diff 0.877; diff+gzip 0.545; bdiff 0.947; vdelta 0.870

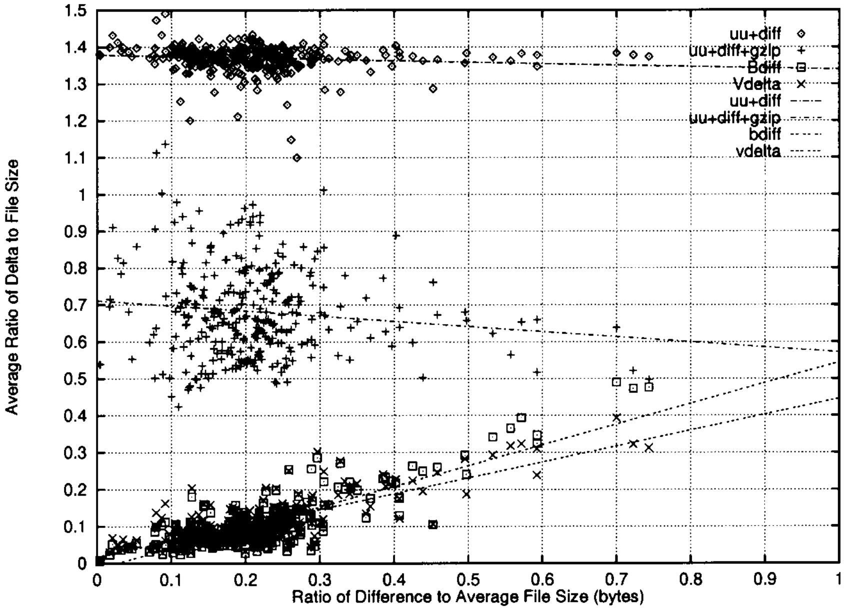
Fig. 7. Plot of compression ratio for text files.

algorithm. Here, one can see how much better *bdiff* and *vdelta* correlate[6] to the difference than *diff* and *diff* with *gzip*. Though *diff* with *gzip* performs as well as *bdiff* and *vdelta* for text files (Figure 1), it performs much more poorly for ELC and object files. The outlying points correspond to files where the difference is much smaller than the file size. All algorithms performed more poorly on the object file set (Figure 3) than on the other sets.

The next set of three graphs—Figures 4, 5, and 6—presents the same data as a log-log plot. The logarithmic scales permit the presentation of a much greater range of differences and delta sizes. As expected, the linear regression lines are not straight due to their nonzero y-intercepts. This is caused by a constant overhead factor in the delta format. Here one can see the low end of the graph in more detail. The separation between the data points for the different algorithms is clearly visible, and the band for *diff* with *gzip* is much closer for those of *bdiff* and *vdelta* for text files than for the other categories.

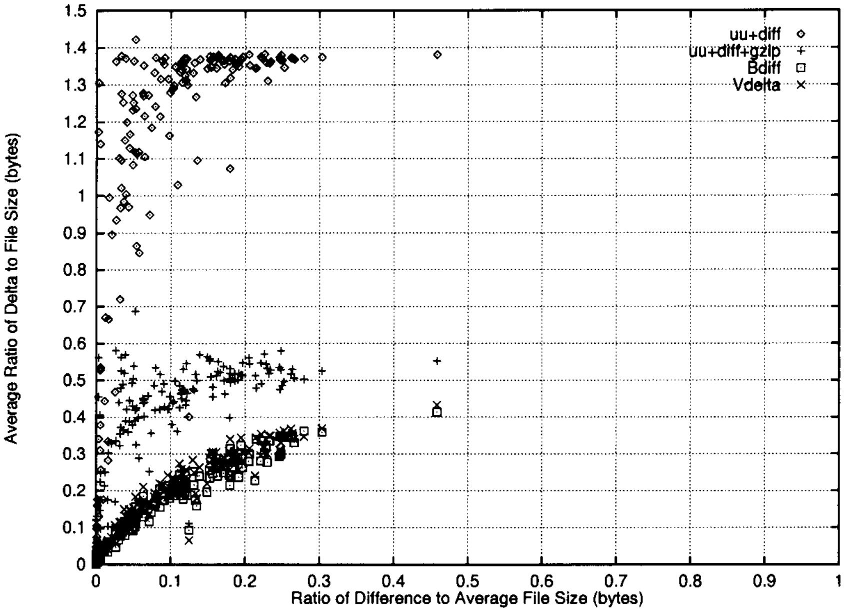
The final set of three graphs—Figures 7, 8, and 9—presents the average compression ratio against the ratio of difference to average file size. Here, the x-axis is given as one minus the LCS size divided by the average file size in each pair, thus expressing how much the two files differ as a ratio. The y-axis is the size of the delta produced by a given algorithm divided by

[6] Correlation coefficients are provided for each graph.



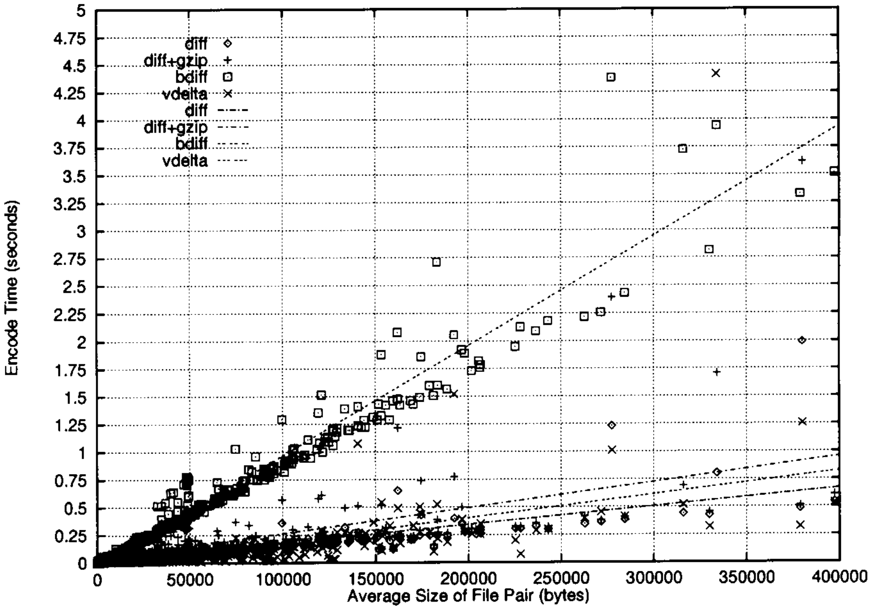
Correlations: uu+diff -0.111; uu+diff+gzip -0.118; bdiff 0.832; vdelta 0.778

Fig. 8. Plot of compression ratio for ELC files.



Correlations: uu+diff 0.742; uu+diff+gzip 0.736; bdiff 0.958; vdelta 0.945

Fig. 9. Plot of compression ration for object files.



Correlations: diff 0.813; diff+gzip 0.672; bdiff 0.971; vdelta 0.599

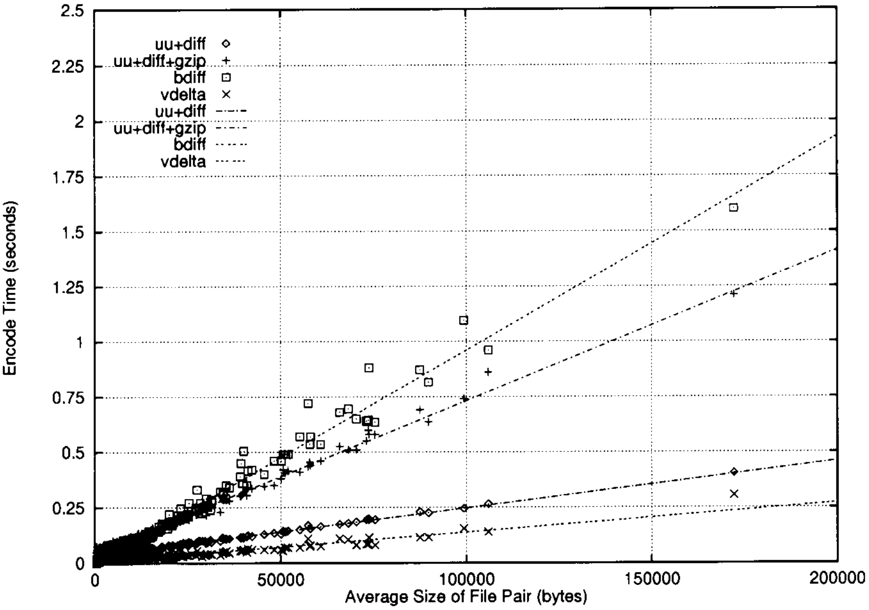
Fig. 10. Plot of encoding time for text files.

the average file size for the pair, i.e., the size of the delta as a ratio to the file size. Here it becomes clear how badly *uuencode* disrupts the Unix *diff* algorithm. Good correlation is obtained for *bdiff* and *vdelta*, whereas *diff* and *diff* with *gzip* appear to be independent of the difference ratio.

All these graphs show clear trends in the performance of *diff*, *diff* + *gzip*, *bdiff*, and *vdelta*.

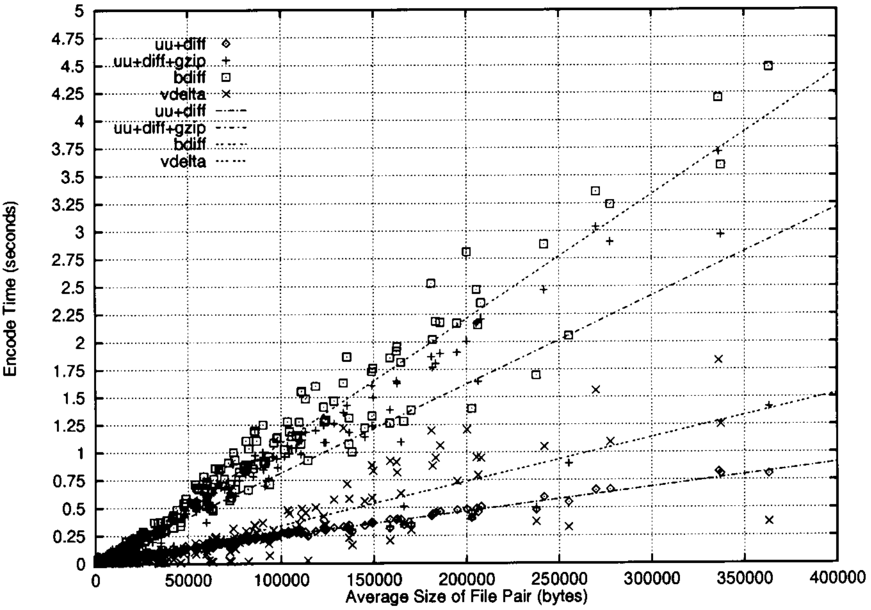
7.2 Efficiency

Time performance for both compression and decompression is also important in evaluating delta algorithms. Time, given as a sum of system and user time as given by the Unix *time* utility, is plotted against the average file size. The first three plots—Figures 10, 11, and 12—present encoding times in seconds and the remaining three plots—Figures 13, 14, and 15—show decoding times in seconds. The scale is not the same for all plots, but the aspect ratio is held constant in each group. The reader should note the change in aspect ratio between the encode plots and the decode plots. Decoding is much faster for all algorithms. Though, the relative performance of the algorithms vary between encoding and decoding, *vdelta* is almost as fast as *diff* (the fastest) for encoding and much faster than all algorithms for decoding.



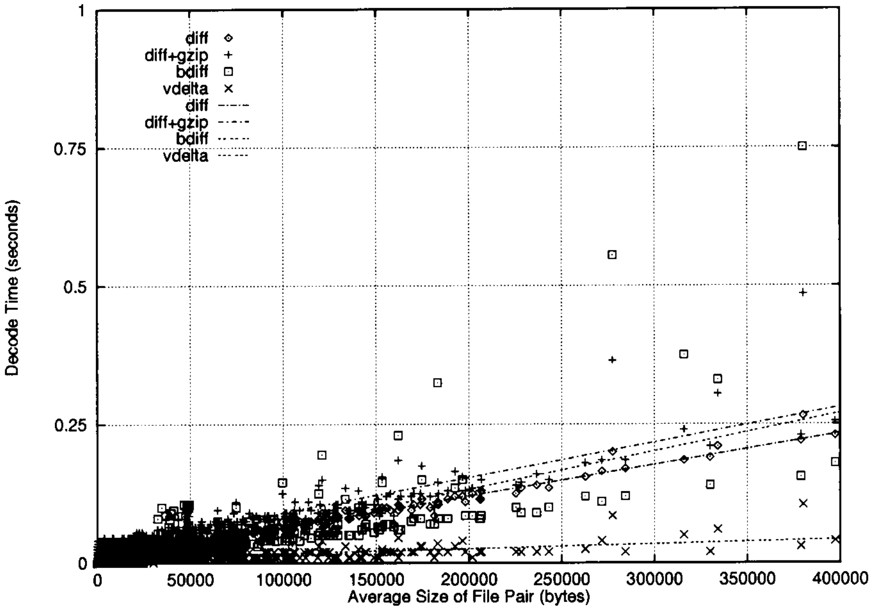
Correlations: uu+diff 0.993; uu+diff+gzip 0.996; bdiff 0.992; vdelta 0.965

Fig. 11. Plot of encoding time for ELC files.



Correlations: uu+diff 0.994; uu+diff+gzip 0.875; bdiff 0.973; vdelta 0.855

Fig. 12. Plot of encoding time for object files.



Correlations: diff 0.987; diff+gzip 0.927; bdiff 0.775; vdelta 0.612

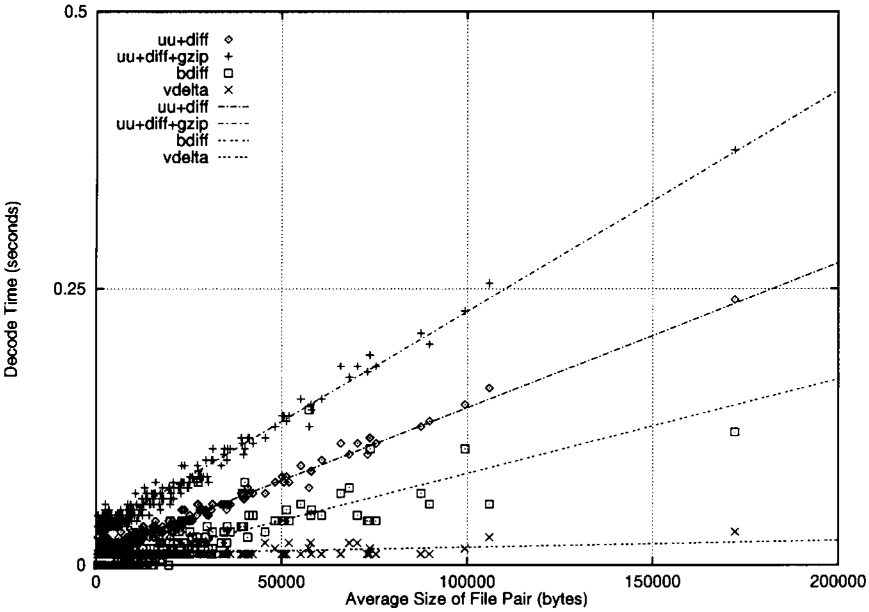
Fig. 13. Plot of decoding time for text files.

8. CONCLUSION

Vdelta is the best algorithm overall. Its coding and decoding performance is high enough to be used for interactive applications. For example, it could be used to improve performance of raster display updates over relatively slow network links. Though *bdiff* generates output that is comparable in size to *vdelta*, *vdelta* is much faster. Both *vdelta* and *bdiff* result in delta sizes that correlate well with the difference. This is not true for *diff*. In the best case—text files—*diff* only reaches the effectiveness of *vdelta* and *bdiff* when it is combined with *gzip*. As expected, using *uuencode* is not a good idea for binary files, since it breaks *diff*'s algorithm for detecting unchanged sequences. This property was anticipated, because *uuencode* essentially removes all natural newlines and adds new ones at constant intervals. This means that only changes that do not modify the positions of unmodified characters or change the file length by an exact multiple of the constant interval can be effectively processed by *diff*.

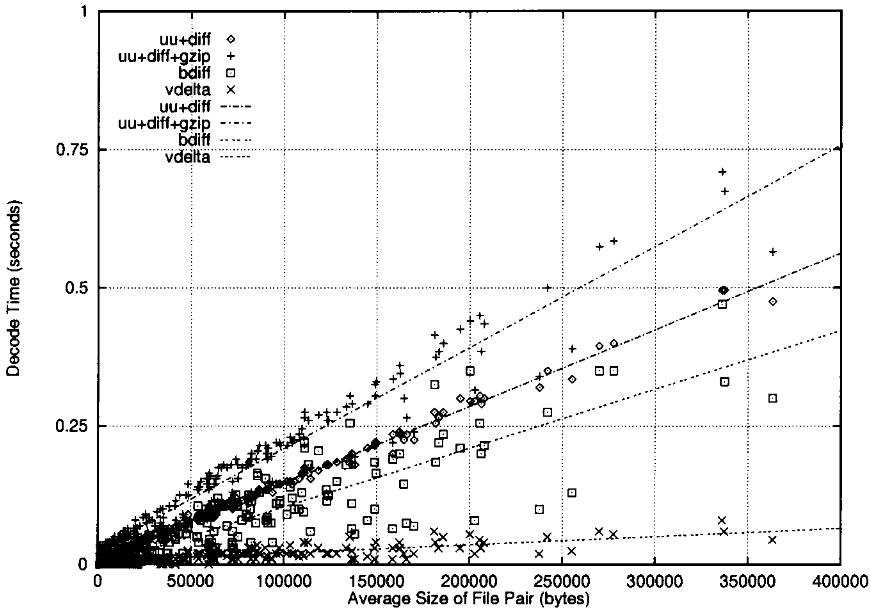
9. FUTURE WORK

The result of this article apply only to one-dimensional data. Other studies should consider two- and higher-dimensional data such as images and video data. In addition, this test suite could be used to fine tune both *bdiff* and *vdelta* and to determine what effect postcompression has on run time and delta size.



Correlations: uu+diff 0.985; uu+diff+gzip 0.987; bdiff 0.868; vdelta 0.395

Fig. 14. Plot of decoding time for ELC files.



Correlations: uu+diff 0.997; uu+diff+gzip 0.976; bdiff 0.883; vdelta 0.787

Fig. 15. Plot of decoding time for object files.

APPENDIX

A. BDIFF

Bdiff is a modification of W. F. Tichy's block-move algorithm [Tichy 1984]. It uses a two-stage approach. First it computes the difference between the two files. Then it uses a second step to compress the resulting difference description. These two parts run concurrently in that the first stage calls the second each time it generates output.

In the first phase, *bdiff* builds an index, called a suffix tree, for the first file. This tree is used to look up blocks, i.e., substrings, of the second file to find matches in the first file. A greedy strategy is used, i.e., every possible match is examined to ensure that the longest possible match is found. The output from this phase is a sequence of copy blocks and character insertions that encode the second file in terms of the first. It can be shown that the algorithm produces the smallest number of blocks and runs in linear time. It also discovers crossing blocks, i.e., blocks whose order was permuted in the second file.

The second phase efficiently encodes the output of the first. A block is represented as a length and an offset into the first file. Characters and block lengths are encoded in the same space by adding 253 (256 minus the three unused lengths) to lengths before encoding. Blocks of lengths less than four bytes are converted to character insertions. Characters and lengths are then encoded using a common splay tree [Jones 1988]. The splay tree is used to generate a character encoding that ensures that frequently encoded characters are shorter than uncommon characters. Splay trees dynamically adapt to the statistics of the source without requiring an extra pass. A separate splay tree encodes the offsets.

Bdiff actually uses a sliding window of 64KB on the first file, moving it in 16KB increments. This means that the first phase actually builds four suffix trees that index 16KB each of the first file. The window is shifted forward whenever the encoding of the second file crosses a 16KB boundary, but in such a fashion that the top window position in the first file is always at least 16KB ahead of the current encoding position in the second file. Whenever the window is shifted, the oldest of the four suffix trees is discarded and a new one built in its space. The decoder has to track the window shifts, but does not need to build the suffix trees. Position information is given as an offset from the beginning of the window.

B. VDELTA

Vdelta is a new technique that combines both data compression and data differencing. It is a refinement of W. F. Tichy's block-move algorithm [Tichy 1984], in that, instead of a suffix tree, *vdelta* uses a hash table approach inspired by the data parsing scheme in the 1978 Ziv-Lempel compression technique [Ziv and Lempel 1978]. Like block-move, the Ziv-Lempel technique is also based on a greedy approach in which the input string is parsed by longest matches to previously seen data. Both Ziv-Lempel and block-move techniques have linear-time implementations [McCreight 1976]. However, implementations of both of these algorithms can be memory intensive, and without careful consideration, they can also be slow because the work required at each iteration is large. *Vdelta* generalizes Ziv-Lempel and block-move by allowing for string matching to be done both within the target data and between a source data and a target data. For efficiency, *vdelta* relaxes the greedy parsing rule so that matching prefixes are not always maximally long. This modification allows the construction of a simple string-matching technique that runs efficiently and requires minimal main memory.

B.1 Building Difference

For encoding, data differencing can be thought of as compression, where the compression algorithm is run over both sequences but output is only generated for the second sequence. The idea is to construct a hash table with enough indexes into the sequence for fast string matching. Each index is a position which is keyed by the four bytes starting at that position. In order to break a sequence into fragments and construct the necessary hash table, the sequence is processed from start to end; at each step the hash table is searched to find a match. Processing continues at each step as follows:

(1) if there is no match,
  (a) insert an index for the current position into the hash table,
  (b) move the current position forward by 1, and
  (c) generate an insert when in output mode; or

(2) if there is a match,
  (a) insert into the hash table indexes for the last 3 positions of the matched portion,
  (b) move the current position forward by the length of the match, and
  (c) generate a copy block when in output mode.

Each comparison is done by looking at the last three bytes of the current match plus one unmatched byte and checking to see if there is an index in the hash table that corresponds to a match. The new match candidate is checked backward to make sure that it is a real match before matching forward to extend the matched sequence. If there is no current match, i.e., just starting a new match, use the four bytes starting at the current position.

As an example, assume the sequence below with the beginning state as indicated (the ⇓ indicates the current position):

⇓																			
0	1	2	3	3	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
b	c	d	e	a	b	c	d	a	b	c	d	a	b	c	d	e	f	g	h

The algorithm starts at position 0. At this point the rest of the sequence is the entire sequence, so there is no possible match to the left. Case 1 requires position 0 to be entered into the hash table (indicated with an asterisk under it) then to advance the current position by 1.

	⇓																		
0	1	2	3	3	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
b	c	d	e	a	b	c	d	a	b	c	d	a	b	c	d	e	f	g	h
*																			

This process continues until position 8 is reached. At that time, we have this configuration:

								⇓											
0	1	2	3	3	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
b	c	d	e	a	b	c	d	a	b	c	d	a	b	c	d	e	f	g	h
*	*	*	*	*	*	*	*												

Now the rest of the sequence is "abcdabcdedfg." The longest possible match to some part previously processed is "abcdabcd," which starts at location 4. Case 1 dictates entering the last 3 positions of the match (i.e., 13, 14, 15) into the hash table, then moving the current position forward by the length of the match. Thus the current position becomes 16 in this example.

																⇓			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
b	c	d	e	a	b	c	d	a	b	c	d	a	b	c	d	e	f	g	h
*	*	*	*	*	*	*	*						*	*	*				

The final step is to match "efgh" (which fails), so the last mark is on position 16. The current position moves to position 17 which now does not have enough data left for the next hash code, so the algorithm stops after outputting the last three characters.

																	⇓		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
b	c	d	e	a	b	c	d	a	b	c	d	a	b	c	d	e	f	g	h
*	*	*	*	*	*	*	*						*	*	*	*			

Note that the matching algorithm above will actually find the longest match if indexes are kept for every location in the string. The skip in step

(2b) prevents the algorithm from being able to always find the longest prefix; however, this rule saves considerable processing time and memory space. In fact, it is easy to see from the above construction rules that the space requirement is directly proportional to the output. The more compressible a target data set is, the faster it is to compress it.

B.2 Difference Encoding

In order to minimize the output generated, the block-move list generated above must be encoded. The output of *vdelta* consists of two types of instructions: **add** and **copy**. The **add** instruction has the length of the data followed by the data itself. The **copy** instruction has the size of the data followed by its address. Two caches are maintained as references to minimize the space required to store this address information.

Each instruction is coded starting with a control byte. Eight bits of the control byte are divided into two parts. The first 4 bits represent numbers from 0 to 15, each of which defines a type of instruction and a coding of some auxiliary information. Below is an enumeration of the first 10 values of the first 4 bits:

0: an **add** instruction,

1,2,3: a **copy** instruction with position in the **QUICK** cache,

4: a **copy** instruction with position coded as an absolute offset from the beginning of the file,

5: a **copy** instruction with position coded as an offset from current location, and

6,7,8,9: a **copy** instruction with position in the **RECENT** cache.

For the **add** instruction and the **copy** instructions above, the second 4 bits of the control byte, if not zero, code the size of the data involved. If these bits are 0, the respective size is coded as a subsequent sequence of bytes.

The above-mentioned caches—**QUICK** and **RECENT**—enable more compact coding of file positions. The **QUICK** cache is an array of size 768 (3*256). Each index of this array contains the value p of the position of a recent **copy** instruction such that p modulo 768 is the array index. This cache is updated after each **copy** instruction is output (during coding) or processed (during decoding). A **copy** instruction of type 1, 2, or 3 will be immediately followed by a byte whose value is from 0 to 255 that must be added to 0, 256, or 512 respectively to compute the array index where the actual position is stored. The **RECENT** cache is an array with 4 indices storing the most recent 4 copying positions. Whenever a **copy** instruction is output (during coding) or processed (during decoding), its copying position replaces the oldest position in the cache. A **copy** instruction of type 6, 7, 8, or 9 corresponds to cache index 1, 2, 3, or 4, respectively. Its copying position is guaranteed to be larger than the position stored in the corresponding cache index, and only the difference is coded.

It is a result of this encoding method that an **add** instruction is never followed by another **add** instruction. Frequently, an **add** instruction has data size less than or equal to 4, and the following **copy** instruction is also small. In such cases, it is advantageous to merge the two instructions into a single control byte. The values from 10 to 15 of the first 4 bits code such merged pairs of instructions. In such a case, the first 2 bits of the second 4 bits in the control byte code the size of the **add** instruction, and the remaining 2 bits code the size of the **copy** instruction. Below is an enumeration of the values from 10 to 15 of the first 4 bits:

10: a merged **add/copy** instruction with copy position coded as itself,

11: a merged **add/copy** instruction with copy position coded as difference from the current position,

12,13,14,15: a merge **add/copy** instruction with copy position coded from a **RECENT** cache.

In order to elucidate the overall encoding scheme, consider the following files:

Version1: a b c d a b c d a b c d e f g h

Version2: a b c d x y x y x y x y b c d e f

The block-move output would be

1.	**copy**	4	0		01000100	0
2.	**add**	2	“xy”	which encodes to	00000010	“xy”
3.	**copy**	6	20	(instruction	01000110	20
4.	**copy**	5	9	in binary)	01000101	9

Note that the third instruction copies from Version2. The address 20 for this instruction is 16 + 4 where 16 is the length of Version1. Note also that the data to be copied are also being reconstructed, i.e., *vdelta* knows about periodic sequences.

This output encoding is independent of the way the block-move lists are calculated; thus *bdiff* could be modified to use this encoding, and *vdelta* could be modified to use splay coding.

REFERENCES

Gosling, J. A. 1981. A redisplay algorithm. In *Proceedings of the ACM SIGPLAN/SIGOA Symposium on Text Manipulation*. ACM, New York, NY, 123–129.

Hunt, J. W. and McIllroy, M. D. 1976. An algorithm for differential file comparison. Tech. Rep. 41, AT&T Bell Laboratories, Inc., Murray Hill, NJ.

Hunt, J. W. and Szymanski, T. G. 1977. A fast algorithm for computing longest common subsequences. *Commun. ACM 20*, 5 (May), 350–353.

JONES, D. W. 1988. Application of splay trees to data compression. *Commun. ACM 31*, 8 (Aug.), 996–1007.

MCCREIGHT, E. M. 1976. A space-economical suffix tree construction algorithm. *J. ACM 23*, 2 (Apr.), 262–272.

MILLER, W. AND MEYERS, E. W. 1985. A file comparison program. *Softw. Pract. Exper. 15*, 11 (Nov.), 1025–1039.

NAKATSU, N., KAMBAYASHI, Y., AND YAJIMA, S. 1982. A longest common subsequence algorithm for similar text strings. *Acta Inf. 18*, 171–179.

OBST, W. 1987. Delta technique and string-to-string correction. In *Proceedings of the 1st European Software Engineering Conference (ESEC '87)* (Strasbourg, France, Sept. 9-11, 1987), H. Nichols and D. Simpson, Eds. Springer-Verlag, Berlin, Germany, 69–73.

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 moves. *ACM Trans. Comput. Syst. 2*, 4 (Nov.), 309–321.

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

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

Received: September 1996; revised: January 1997 and October 1997; accepted: November 1997