Bidirectional Delta Files

Dana Shapira[1,2] and Michael Kats[2]

[1] Dept. of Computer Science, Ashkelon Academic College, Ashkelon 78211, Israel
[2] Dept. of Computer Science, The Open University of Israel, Raanana 43107, Israel

Abstract. This paper introduces a new method called *Bidirectional Delta file*, which is to construct a two way delta file out of two given files. Previous work focuses on forwards and backwards delta files. Here we suggest efficiently combining them into a single file. Given two files S and T, the paper designs a greedy algorithm, which produces an efficient bidirectional delta file in terms of the memory storage it requires. Given this encoding file and the original file S, one can decode it in order to produce T, and vice versa. Experiments show memory storage savings of at least 25% between the implemented algorithm and the traditional way of using both forwards and backwards delta files.

1. Introduction

Delta compression represents a target file T making use of a source file S. The general approach for differencing algorithms which construct delta files, compress T by finding common substrings between S and T and replacing these substrings by a copy reference. The delta file is then encoded as a sequence of elements which are either pointers to an occurrence of the same substring in S, or individual characters that are not part of any common substring. To improve compression performance, pointers to previously occurring substrings in T are also used. When the delta file is the sequence of differences between a given source file which was chronologically created before the target file we call it a *forward* delta file. If the source file was created after the target file, it is called a *reverse delta file* or a *backwards delta File*. There are several practical applications that benefit from the use of delta compression, in cases where the new information that is received or generated is highly similar to information already present. Such applications include distribution of software revisions, incremental file system backups, and archive systems, where using delta techniques is much more efficient than using regular compression tools. For example, incremental backups can not only avoid storing files that have not changed since the previous back-up and save space by standard file compression, but can also save space by differential compression of a file with respect to a similar but not identical version saved in the previous backup.

A *Bidirectional Delta file* provides concurrent storage and usage of forward and backward delta techniques in a single file. When one wishes to go back and forth between versions, bidirectional delta files should be used, providing flexibility, processing time savings, space storage efficiency, and I/O operations reduction. Instead of storing both source and target versions for future use, the bidirectional delta file together with one of the versions is sufficient. Once the target file is constructed using its previous version and the bidirectional delta file, the source file is no longer needed and can be deleted, thus saving memory storage. A practical application for bidirectional delta files is software distribution. When a new revision is released to license users, the distributer can make use of bidirectional delta techniques, providing both forward and backward deltas in an economical way. When software distribution is done on a remote computer, providing forwards and backwards delta files, the user should transfer these files and perform the upgrade on his personal computer, since memory resources are not always available on the distributer's computer.

This paper suggests that the distributer will provide the corresponding bidirectional delta file instead of the forward and backward deltas, so that a single file could be transferred. Since the bidirectional file is usually smaller than the sizes of the forwards and backwards delta filea, taken together, using it reduces data traffic and storage resources at both ends. Moreover, I/O operations are also saved by transferring a smaller number of bytes. Another connected, yet different application, is in the case a user regrets an *in place* upgrade he just performed, overwriting the source file with the new target file. Undoing this last update, going back to its previous release, can be achieved by using the same bidirectional file.

Generating a delta file of two given files S and T can be done in two typical ways, LCS (Longest Common Subsequence) based algorithms (e.g. [5]), and edit-distance based algorithms (e.g. [1],[15], [17]). An edit distance oriented delta compressor is introduced in the work of Hunt, Vo, and Tichy [6], who compute a delta file by using the reference file as part of the dictionary to LZ-compress the target file. Their results indicate that delta compression algorithms based on LZ techniques significantly outperform LCS based algorithms in terms of compression performance. Based on these papers the constructed delta file in this research uses edit distance techniques including insert and copy commands. The already compressed part of the target file is also referenced for better compression. Shapira and Storer [9] study in-place differential file compression. They present a constant factor approximation algorithm based on a simple sliding window data compressor for the non in-place version of this problem which is known to be NP-Hard. Motivated by the constant bound approximation factor they modify the algorithm so that it is suitable for in-place decoding and present the In-Place Sliding Window Algorithm (IPSW). The advantage of the IPSW approach is simplicity and speed, achieves in-place without additional memory, with compression that compares well with existing methods (both in-place and not in-place). Our bidirectional delta file is not necessarily in-place, but minor changes (such as limiting the offset's size) can result in an in-place bidirectional delta version. Shapira in [10] introduces the problem of merging two delta files, also called *Compressed Transitive Delta Encoding (CTDE)* problem. The problem is to construct a single delta file which has the same effect as the two given delta files, by working directly on the compressed forms, without the use of the source file, and in time proportional to the size of the delta files.

Rochkind [8] introduces the *Source Code Control System* (SCCS), which is a model where each change made to the software module is stored as a discrete delta file. To produce the latest version of the source code module, SCCS follows the forward delta files from the beginning, applying them as it goes. Revision Control System (RCS) described by Tichy in [14, 16] was first to use reverse delta files. A reverse delta file describes how to go backwards in the developed history: it produces the desired revision if applied to the successor of that revision. Sobel's invention [12] is used for generating a stored back-patch to undo the effect of forward patching. A back-update file (reverse delta file) is created, in order to allow future access to the previous version of a file, by providing the information necessary to construct the previous version out of the current version. In Forster [4] a system with concurrent usage of forward and backward delta techniques is presented, which provides an improved method for updating an archive of a computer file to substantially reduce or eliminate problems and disadvantages associated with archive resources like data storage and data transfer speed. In this paper we take this idea one step forward by merging the forward and backward deltas into a single file, in a non trivial way.

Our paper is constructed as follows. Section 2 introduces the problem of generating an efficient bidirectional delta file with an optimal sub-quadratic time algorithm to solve it, using dynamic programming. To save processing time and reduce memory complexity we suggest a streaming greedy algorithm which is gradually developed as follows. The first

attempt algorithm named *Aligned_BD* is presented in section 3, and its drawbacks are overcome in the algorithm named *Non_aligned_BD* presented in Section 4. The final algorithm applies ad hoc heuristics to achieve better compression results. Section 5 presents experimental results showing compression savings of at least 25% over the traditional approach.

2. Problem Presentation and Optimal Solution

Given a source file S and a target file T, let $\Delta(S,T)$ denote the forward delta file of T with respect to S, and $\Delta(T,S)$ denote the backward delta file of S with respect to T. The delta file (forward and backward) can be formed out of three types of items: pointers into the source file, self pointers (pointers copying substrings to the current position in the file from the already scanned portion of the same file), and raw characters. The copies are initially described in the form of ordered pairs, (pos, len) and (off, len) for source-file-pointers and self-pointers, respectively. The second component, *len*, in both types of pointers, describes the length of the reoccurring substring, which is the number of its characters. The position component, *pos*, of a source-file-pointer refers to a copy of a substring starting at position *pos* in the source file. The *off* component of a self-pointer means that the substring starting at the position corresponding to the current ordered pair can be copied from *off* characters before the current position in the decompressed file. A copy from the base file is denoted by a flag bit BP (Base Pointer flag bit), and a self copy from the target file is denoted by OP (Offset Pointer flag bit), while raw characters are given explicitly. Thus (BP, pos, len) refer to copies of substrings from the source file, and (OP, off, len) indicates that the copy is from the target file. For example, suppose S is the string $abcdxxxdyyz$ and T is the string $yyzzzabcdyyzzz$, both starting at index 0. The delta file representing T with respect to S is $\Delta(S,T) = (BP,8,3)zz(BP,0,4)(OP,9,5)$, where $(BP,8,3)$ is used to copy the substring yyz from address 8 of S, $(BP,0,4)$ is used to copy the substring $abcd$ from address 0 of S, and $(OP,9,5)$ is used to copy the substring $yyzzz$ from the beginning of T, which occurs 9 characters before the current position.

A bidirectional delta file is denoted by $BD\Delta(S,T)$, which is a two way differencing file. The objective is to construct a single compact file in linear time in the sizes of the input files. The fundamental approach of storage savings in the bidirectional delta file represents a common substring of S and T using a single copy reference, unlike two independent copies in the forward and backward deltas. A first attempt for solving the problem of choosing the "best" set of common substrings of two given files is by looking at the corresponding delta files. Selecting the same common substrings chosen by the differencing algorithm raises difficulties which stem from the fact that the corresponding delta files are not symmetric. Not only do the forward and backward deltas choose different substrings for pointer references, but even if the same substring is represented by a reference, it does not necessarily use an identical pointer to represent such copy. Using the previous example, the forward delta file used to construct T with respect to S is $\Delta(S,T) = (BP,8,3)zz(BP,0,4)(OP,9,5)$, as explained above. The reverse delta file for representing S with respect to T is $\Delta(T,S) = (BP,5,4)xxx(BP,8,4)$. Although the substring $abcd$ is represented by pointer references in both deltas, the corresponding triplets are different ($(BP,5,4)$ in $\Delta(T,S)$ and $(BP,0,4)$ in $\Delta(S,T)$). Moreover, the common substring $dyyz$ is represented by $(BP,8,4)$ in $\Delta(T,S)$ and overlaps the decoding of $(BP,0,4)(OP,9,5)$ in $\Delta(S,T)$, showing that common substrings are not necessarily copied from the other file. Note that the prefix d of $dyyz$ is copied from S in $\Delta(S,T)$ using the triplet $(BP,0,4)$, but this triplet refers to the occurrence of d at position 3 of S and not the one which occurs in the common substring $dyyz$ at position 7. This example illustrates

that choosing the set of common substrings based on an independent left to right parsing of S and T might result in a small number of short reoccurring substrings, which would impose a non efficient bidirectional delta file. Rather, what is required is to find regions of the two files that are identical by a parallel scan of the files.

An alignment of given strings S and T is a parsing of both of them according to their common substrings so that the common substrings occur in the same relative order. The common substrings (contiguous matching characters) of an alignment are called blocks. In other words, a set of substrings is aligned if by writing T below S and drawing straight lines between corresponding matches, no lines cross. The following figure gives a schematic view of aligned blocks of S and T. Given two strings S and T, the *(global) sequence alignment problem* is finding an alignment of maximal length. Formally, an alignment with k ordered blocks $\{\beta_1, \beta_2, \cdots, \beta_k\}$ is said to be of maximal length if $\Sigma_{i=1}^{k}|\beta_i|$ has the maximum value of all alignments of S and T. Note that not all alignments necessarily have the same number of blocks. Instead of referring to the aligned blocks, β_i $(1 \leq i \leq k)$, one can refer to the contiguous characters between the blocks. These are also known as *gaps*. Thus, another way to formalize the alignment problem is minimizing the accumulated length of the gaps.

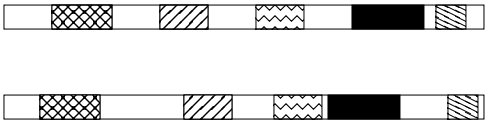


FIGURE 1: *Schematic view of aligned blocks of S and T*

The edit distance problem is another way to measure similarity between two given strings. The original problem was defined as finding the minimum number of insertions, deletions and substitutions in order to transform one string to another. In this paper we focus on uniform costs of the operations. Otherwise, the costs are specified in a given *scoring matrix*. An optimal alignment is an alignment that yields the best edit distance. A gap is the result of the deletion of one or more consecutive characters in one of the strings.

The similarity of two strings of size n, and the associated optimal alignment, can be computed using dynamic programming in $O(n^2)$ time and space [11], and is given in the full version of this paper. Masek and Paterson present a sub-quadratic global alignment string comparison algorithm based on the Four Russians paradigm, which divides the dynamic programming table into uniform sized $(\log n \times \log n)$ blocks. Under the conditions of a constant alphabet and *discreteness* (rational weights for the operations) the time complexity of the algorithm they present is $O(\frac{n^2}{\log n})$. Crochemore et al. [3] describe a $O(\frac{hn^2}{\log n})$ algorithm, where h denotes the entropy of the strings, being faster than Masek and Paterson's algorithm for compressible strings. In practice, even though encoding is done only once, we suggest to apply a streaming greedy linear time heuristic algorithm rather than a dynamic programming approach even when adapting a sub-quadratic time performance.

3. Basic Bidirectional Delta Encoding Algorithm

Given $S = s_1 \cdot s_2 \cdots s_n$ and $T = t_1 \cdot t_2 \cdots t_m$, we adapt the notation $s[i, j]$ for representing the substring $s_i \cdot s_{i+1} \cdots s_j$ of S, and symmetrically for T. A streaming greedy algorithm for constructing a bidirectional delta file is presented in Figure 2. The aligned blocks are found by a synchronized parsing of the strings from left to right. The gaps in both files are encoded using self pointers, i.e., pointers copying substrings to the current position in the file from the already scanned portion of the same file. It uses the well known algorithm of Lempel-Ziv-Storer and Syzmanski, LZSS, [13] for compressing a single file against itself, using a sliding window.

The algorithm parses the strings trying to keep the pointers to both files synchronized. S and T are scanned in parallel by either checking whether the position of S precedes the position of T, or by checking whether the remaining part of S is bigger than the remaining part of T. Using the first alternative, if the position of S precedes the position of T the next common substring is found by searching S for the longest substring that matches the substring of T which starts at the current position j of T. Otherwise, the next common substring is found by searching T for the longest substring that matches the substring of S which starts at the current position i of S. The algorithm uses the $CS()$ method which is applied on two strings, X and Y, and returns an ordered pair where the first component is the index of the starting position of a substring in Y which matches the longest prefix of X, and the second component is its length. For example, $CS(abcdxxx, xyzabcdyyabcdx) = (9, 5)$, since the longest occurrence of a prefix of X in the second component string is at its ninth position, and refers to the string $abcdx$, which consists of 5 characters. Note that this method is not symmetric, and $CS(X, Y)$ is not necessarily equal to $CS(Y, X)$.

The *Aligned_BD* algorithm can be implemented in linear time in the sizes of S and T by using a suffix trie for the string $S \cdot T\$$, where \$ is a character not belonging to the original alphabet of S and T. Every node v of a regular trie is associated with a string which is obtained by concatenating, top down, the labels on the edges forming the path from the root to node v. The suffix trie is a *compact trie*, i.e., each path of single child nodes is collapsed to its starting and ending node, with an edge labeled with a string that is a concatenation of all labels on the original path so that each non-leaf node (except the root that might be a single child node) has at least two children. The set of strings associated to its leaves is the set of the suffixes of $S \cdot T\$$. Since the \$ character does not occur elsewhere in S or T, each suffix corresponds to a unique leaf. Therefore, a node with descendant nodes that refer to substrings with prefixes from S and T correspond to common substrings, and the deeper such node is the longest common substring. This implies that the $CS()$ method can be performed in time proportional to the length of the longest common substring of the the two input strings which is also a prefix of its first component string. In practice, $CS()$ can be implemented using hashing, having better processing time, at the price of not necessarily locating the longest match.

The algorithm uses indices i and j to point to the current location in S and T, respectively. Assistant indices, i_{old} and j_{old}, are used for saving the starting position of the next portion of the file to be encoded, and i_{new} and j_{new} are used for saving the starting position in S and T files, respectively, of the common substring found by the $CS()$ method. The length of the common substring found by the $CS()$ method is compared against a supplied parameter, to justify the use of this aligned block by checking whether the common substring is long enough. If the length is less than a given parameter MinLen, the method $CS()$ is applied on the successive position in S or T. Otherwise, the gaps in both files are encoded using self pointers and raw characters, using LZSS, followed by the encoding of the common substring itself. The indices i and j are then advanced together with the assistant indices, i_{old} and j_{old}, and the search continues right after the common substring. When the scan of one of the files end, the remaining part of the other file is compressed using LZSS and outputted to the bidirectional delta file.

The format of the bidirectional file is composed out of flag bits, aligned pointers, and LZSS items of S and T, which, in turn, include flag bits, self pointers and raw characters. Thus, 3 flag bits are needed to distinguish between such items in $BD\Delta(S, T)$, for which LZSS items require 2 additional inner flag bits to differentiate pointers from raw characters. For simplicity we refer to flag bits $1, 2$ and 3, for aligned blocks, LZSS S-items, and LZSS

```
Aligned_BD(S, T, MinLen) {
    BDΔ(S,T) ⟵ ϵ; //initialize with empty string
    i ⟵ 0; j ⟵ 0; //current position in S and T
    i_old ⟵ 0; j_old ⟵ 0; //the next starting position to be encoded
    while S and T not empty {
        if i ≤ j
            (i_new, len) ⟵ CS(t[j,m],s[i,n])
            if(len ≤ MinLen) // not long enough
                j++
            else
                BDΔ(S,T) ⟵ BDΔ(S,T) · 2 · LZSS(s[i_old, i_new]) // S LZSS item
                BDΔ(S,T) ⟵ BDΔ(S,T) · 3 · LZSS(t[j_old, j]) // T LZSS item
                BDΔ(S,T) ⟵ BDΔ(S,T) · 1 · (i_new, j, len) // aligned block
                i ⟵ i_new + len; j ⟵ j + len; i_old ⟵ i ; j_old ⟵ j
        else // j < i
            (j_new, len) ⟵ CS(s[i,n],t[j,m])
            if(len ≤ MinLen) // not long enough
                i++
            else
                BDΔ(S,T) ⟵ BDΔ(S,T) · 2 · LZSS(s[i_old, i]) // S LZSS item
                BDΔ(S,T) ⟵ BDΔ(S,T) · 3 · LZSS(t[j_old, j_new]) // T LZSS item
                BDΔ(S,T) ⟵ BDΔ(S,T) · 1 · (i, j_new, len) // aligned block
                i ⟵ i + len;j ⟵ j_new + len; i_old ⟵ i;j_old ⟵ j
    }
    if S empty // concatenate the encoding of suffix of S or T
        BDΔ(S,T) ⟵ BDΔ(S,T) · 3 · LZSS(t[j_old, m]) // T LZSS item
    else
        BDΔ(S,T) ⟵ BDΔ(S,T) · 2 · LZSS(s[i_old, n]) // S LZSS item
    return BDΔ(S,T)
}
```

FIGURE 2: *Basic algorithm for constructing a bidirectional delta file of two given files S and T*

T-items, respectively, and ignore the inner implementation of the LZSS components (since pointers are given as ordered pairs and raw characters are written explicitly) . Note that, for example, when a raw character is individually output to the bidirectional file, the appropriate inner flag bit of LZSS is concatenated to the corresponding $BD\Delta(S,T)$ flag bit to differentiate a S raw character from a T raw character. An aligned block is represented by a **1** flag bit, and followed by a triple (S_{add},T_{add},len) for referring to the common substring that occurs in S at address S_{add} and in T at address T_{add}, and the number of characters is len. The items of the encoded gaps can be put in between the encodings of the corresponding common substring in any order (e.g. alternating LZSS S-items and LZSS T-items), as long as they occur in the same order as in LZSS for S and T. For simplicity, LZSS S-items are put before LZSS T-items in each gap. Consider for example $S = xxxabcdefxablmn$ and $T = abcdxyzlmnxxx$. Using flag bits BP and OP as defined in the beginning of Section 2, the forward and backward delta files are $\Delta(S,T) = (BP,3,4)xyz(BP,12,3)(BP,0,3)$ and $\Delta(T,S) = (BP,10,3)(BP,0,4)ef(OP,7,3)(BP,7,3)$. By applying the algorithm of Figure 2 for constructing the bidirectional file, two aligned blocks are found by the $CS()$ method, *abcd* and *lmn* and are encoded by $(\mathbf{1},3,0,4)$ and $(\mathbf{1},12,7,3)$, respectively (flag bits are highlighted in bold). The gaps between these common substrings (including the gaps at both ends of the strings) are encoded using LZSS. The gaps of S are encoded by three times $(\mathbf{2},x)$ for the first gap xxx, and $(\mathbf{2},e)(\mathbf{2},f)(\mathbf{2},7,3)$ for the second gap *efxab*. The first gap of T is encoded $(\mathbf{3},x)(\mathbf{3},y)(\mathbf{3},z)$ for xyz and three times $(\mathbf{3},x)$ for the last gap of T, xxx. The output bidirectional delta file is therefore:

$$BD\Delta(S,T) = (\mathbf{2},x)(\mathbf{2},x)(\mathbf{2},x)(\mathbf{1},3,0,4)(\mathbf{2},e)(\mathbf{2},f)(\mathbf{2},7,3)$$
$$(\mathbf{3},x)(\mathbf{3},y)(\mathbf{3},z)(\mathbf{1},12,7,3)(\mathbf{3},x)(\mathbf{3},x)(\mathbf{3},x).$$

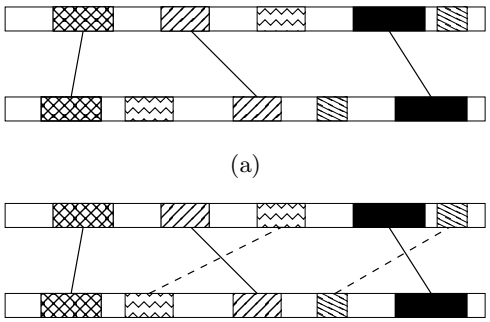
4. Non-Aligned Bidirectional Delta Encoding Algorithm

In the previous section a first attempt to construct the bidirectional delta file was presented. Empirical experiments given in the following section show poor compression performance as compared to the traditional approach of using both forward and backward deltas. In this section we suggest two improved versions to the basic algorithm, which have significant preference on the traditional one.

In our last example we discover that the substring xxx, even though being a common substring of S and T, is encoded as individual characters in both LZSS encodings of the bidirectional delta file. This loss in compression is due to the fact that only aligned common substrings are efficiently encoded. An improved version of the basic bidirectional delta encoding algorithm, named *non_aligned_BD*, suggests using a regular delta encoding instead of the LZSS encoding used in the *non_aligned_BD* algorithm presented in Figure 2. This comes at the price of having 3 different formats of items in the delta encoding (pointers to the source file, self pointers, and raw characters) as opposed to only 2 in the LZSS encoding (self pointers and raw characters).

Returning to our previous example, substring xxx of the first gap of S is therefore encoded as $(\mathbf{2}, BP, 10, 3)$ for copying it from the 10^{th} position of T, and the suffix xxx of T is replaced by $(\mathbf{3}, BP, 0, 3)$, for copying it from the beginning of S. The output bidirectional delta file is therefore:

$BD\Delta(S,T) = (\mathbf{2}, BP, 10, 3)(\mathbf{1}, 3, 0, 4)(\mathbf{2}, e)$
$(\mathbf{2}, f)(\mathbf{2}, OP, 7, 3)(\mathbf{3}, x)(\mathbf{3}, y)(\mathbf{3}, z)(\mathbf{1}, 12, 7, 3)(\mathbf{3}, BP, 0, 3).$



(a)

(b)

FIGURE 3: *Schematic view of the proposed algorithms*

Figures 3(a) and 3(b) are schematic illustrations, which visually represent differences between *Aligned_BD* and *Non_aligned_BD* algorithms. The source file S and target file T are presented such that S is drawn above T. Common substrings of S and T have the same texture. In the *Aligned_BD* algorithm, only aligned blocks are used as pointers to the other file. The gaps between the aligned blocks are encoded by pointing backward to previous occurring substrings in the same file. On the other hand, in the *Non_aligned_BD* algorithm non-aligned blocks are also used as pointers to the other file The remaining portions of the source and target files are encoded by pointing backward to previous occurring substrings in the same file.

The following example shows that there are many examples for which the $CS()$ method used in Figure 2 is inefficient. Let $S = abcdxyzlmnxxx$ and $T = xxxabcdefxablmn$, which is in fact swapping the roles of S and T in our previous example. In this example, the first common block selected by the $CS()$ algorithm, is xxx, and occurs in opposing ends of the files. As a result, the resulting set of aligned block consists of a single common

substring. Since the compression savings of a bidirectional delta file over conventional delta files (backward and forward) is due to using a single copy of the aligned blocks, the resulting bidirectional file may be relatively inefficient. In fact, other aligned blocks could have been selected to better utilize the similarity of the two given files. The advantage of this bidirectional delta file over the corresponding forward and backward delta files, is referring only once to the single aligned block. However, the remaining items in the bidirectional delta file use more flag bits, than the items in the forward and backward delta files (the bidirectional delta file uses 3 flag bits to differentiate aligned blocks, S-items and T-items in addition to the flag bits, which are also used in a regular delta file). Although such an example is quite rare, it emphasizes the loss of aligned blocks which are too far apart. Our final bidirectional delta encoding algorithm, named *improved_BD*, suggests using heuristics for selecting the aligned blocks by controlling the distance between the corresponding positions in the source and target files and checking whether its proportional to its length. More formally, if the factor between the length of the candidate block, and the distance between the last chosen block and the candidate block is less than a supplied constant (empirically chosen as 0.1, 0.2,...,1.0) the candidate block is ignored, and the search continues with the following position. Otherwise, the block is chosen, and both positions to the files are advanced.

5. Experimental Results

This section presents experiments which evaluates the compression savings and processing time performance of a bidirectional delta file constructed as described above. At first, a simulation LZ based delta encoder was implemented, using pointers to the source file, pointers to the already scanned portion of the target file, and raw characters. The format of a delta file was constructed around key parameters of the UNIX *gzip* utility that uses a window of size 32K and maximum match length 256. For our experiments we consider three typical series of consecutive versions of software releases. First, source codes of the GNU GCC compiler collection which includes front ends for C, C++, Objective-C, Fortran, Java, and Ada, as well as libraries for these languages (libstdc++, libgcj,...), versions 4.3.3 and 4.4.0, gcc.c , combine.c and parser.c files. Second, the GNU source code of MySQL server, which is a portable multiuser relational database management system, versions 5.0.82 and 5.0.83, sql_select.cc and sql_yacc.cc files. Third, the GNU Emacs text editor source code, versions 22.3 and 23.1, calc.texi file. Table 1 presents the set of software releases that were used in our experiments and their corresponding sizes in bytes.

Source File	Size	Target File	Size
gcc.c 4.3.3	227,793	gcc.c 4.4.0	230,932
combine.c 4.3.3	430,988	combine.c 4.4.0	429,853
parser.c 4.3.3	629,211	parser.c 4.4.0	678,333
sql_select.cc 5.0.82	515,311	sql_select.cc 5.0.83	515,525
sql_yacc.cc 5.0.82	1,285,116	sql_yacc.cc 5.0.85	1,336,001
calc.texi 22.3	1,471,104	calc.texi 23.1	1,484,655

TABLE 1: *Experimental data files*

Our first experiment was used to empirically evaluate the compression performance of our delta simulation against *xdelta* and *zdelta* LZ based UNIX utilities, which use the zlib compression library. Table 1 presents the size in bytes (Y-axis) of the output deltas applied to the data files (X-axis) of Table 1, where the Y-axis are logarithmic rather than linear axis providing better appearance. As can be seen, the compression performance of

our delta algorithm, denoted by $Simulated\Delta(S,T)$, consistently outperforms *xdelta*, but is slightly less efficient as compared to *zdelta*. As the format of our bidirectional delta file was constructed around the key parameters of this delta simulation, the compression results given in Figure 4, are encouraging, since one can only expect that the compression we achieve here will be improved by using a gzip or zdelta implementation.

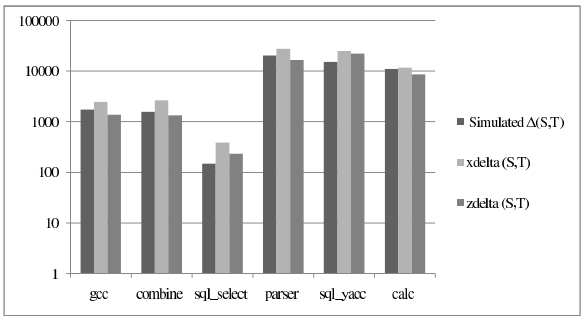


FIGURE 4: *Compression Performance of Regular Delta Simulation, xdelta and zdelta*

The following experiment compares the processing time of encoding and decoding of bidirectional files as opposed to the traditional approach of encoding and decoding two separate files. The top half of table 2 presents the encoding processing time performance, and the bottom half gives the results for decoding, where all figures are given in seconds. Each column corresponds to a different set of source and target file as in Table 1. The forward+backward rows refer to the time taken to construct the forward delta file, in addition to the time it takes to generate the backward delta file. The *improved_BD* row corresponds to the time taken to construct the bidirectional delta file using the improved version, which is the most time consuming version of all bidirectional delta algorithms. The encoding running time for regular deltas is superior as compared to *improved_BD* in most cases. However, encoding is done only once, while decoding can be done over and over. Surprisingly, our time performance for decoding are comparable to the general approach despite the fact of dealing with a bigger number of different formats.

	gcc.c	combine.c	parser.c	sql_select.cc	sql_yacc.cc	calc.texi
Encoding time						
forward+backward	0.284	0.530	2.711	0.496	2.941	2.570
improved_BD	0.324	0.475	8.296	0.271	17.310	4.853
Decoding time						
forward+backward	0.052	0.071	0.194	0.088	0.210	0.247
improved_BD	0.048	0.075	0.112	0.084	0.211	0.240

TABLE 2: *Encoding and Decoding Processing Time Performance*

	gcc.c	combine.c	parser.c	sql_select.cc	sql_yacc.cc	calc.texi
aligned_BD	120	109	123	45	177	123
non_aligned_BD	94	89	93	38	152	90
improved_BD	82	82	83	30	91	77

TABLE 3: *Compression performance*

Our last and most important experiment was applied in order to show the gain in compression performance of our suggested algorithms as compared to the sizes of the corresponding forward and backward deltas. A comparison of *aligned_BD*, *non_aligned_BD*, and *improved_BD* with the traditional method can be found in Table 3. The figures are

given as a percentage of the sum of the sizes of the forward and backward delta files, corresponding to 100%. Thus, results less than 100 indicate an improvement over the traditional approach, while results more than 100 show inferior performance. As can be seen, *improved_BD* has compression capability of about 25%-30% better than the traditional way of using two separate forward and backward deltas. As expected, the two preceding constructions of bidirectional delta file are not as good as the improved version, although even the *non_aligned_BD* version is in most cases better than forward+backward deltas.

6. Conclusion

In this paper the term of *bidirectional delta files* is introduced, providing flexibility when going back and forth between versions. Optimal and practical algorithms are suggested and compared to the general approach of using both forward and backward delta files. Experiments show a consistent advantage of our bidirectional delta compression that does not only improve storage savings, but imply I/O operations reduction in case the deltas should be transformed over a network (e.g. file system backups or software distribution). Surprisingly, the time processing is comparable to that of the traditional approach even with the more complicated structure of the file. Moreover, one can expect that the compression and processing time performance of our algorithm can only be improved by using a gzip or zdelta implementation.

References

[1] Agarwal, R. C., Amalapurapu, S., and Jain,S.: *An approximation to the greedy algorithm for differential compression of very large files*, in Tech. Report, IBM Alamaden Res. Center, 2003.

[2] Burns, R. C.and Long, D. D. E.: *In-place reconstruction of delta compressed files*, in Proceedings of the ACM Conference on the Principles of Distributed Computing, ACM, 1998.

[3] Crochemore, M.,Landau, G. M. and Ziv-Ukelson, M., *A Sub-quadratic Sequence Alignment Algorithm for Generalized Cost Metrics* SIAM Journal of Computing, 32(6), 2003, 1654–1673.

[4] Forster, K.D.: *Method and System for Updating an Archive of a computer file*, European Patent Specification, EP 1259883 B1, 2005, United States Patent, Patent No.: WO/2001/06537, 2001.

[5] P. Heckel: *A technique for isolating differences between files.* CACM, 21(4) 1978, 264–268.

[6] Hunt, J. J., Vo, K. P., and Tichy, W.: *Delta algorithms: An empirical analysis.* ACM Trans. on Software Engineering and Methodology 7, 1998, 192–214.

[7] Masek, W.J. and Paterson, M.S., *A faster algorithm for computing string edit distances.* J. Comput. Syst. Sci., 20, 1980, 18–31.
V. L. Arlazarov, E. A. Dinic, M. A. Kronod, and I. A. Faradzev, "On Economical Construction of the Transitive Closure of an Oriented Graph", Soviet Math. Dokl. Vol. 11 (1970), pp. 1209–1210.

[8] Rochkind, M.J.: *The Source Code Control System*, IEEE Transactions on Software Engineering, *Volume 1(4)*, 1975, 364–370.

[9] Shapira, D. and Storer, J. A.: *In place differential file compression.* The Computer Journal, 48 2005, 677–691.

[10] Shapira, D.: *Compressed Transitive Delta Encoding.* Proc. Data Compression Conference, DCC-2009, 2009, 203–212.

[11] Smith, T.F. and Waterman M.S.: *Identification of common molecular subsequences*, Journal of Molecular Biology, 147, 1981, 195–197.

[12] Sobel E.W. and Nachenberg C.S.: *Storage of Reverse Delta Updates.* United States Patent, Patent No.: US006349311B1, 2002.

[13] Storer J.A. and Szymanski T.G. *Data Compression Via Textual Substitution.* JACM, 29:4, 1982, 928–951.

[14] W. F. Tichy: *Design, Implementation, and Evaluation of a Revision Control System.* , in Proceedings of the 6th International Conference on Software Engineering, 1982, 58–67.

[15] W. F. Tichy: *The string to string correction problem with block moves.* ACM Transactions on Computer Systems, 2(4) 1984, 309–321.

[16] W. F. Tichy: *RCSa system for version control.* Software-Practice & Experience, 15(7), 1985, 637–654.

[17] P. Weiner: *Linear pattern matching algorithms*, in Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory (FOCS), 1973, 1–11.