Compressed Delta Encoding for LZSS Encoded Files

Shmuel T. Klein
Department of Computer Science
Bar Ilan University
Ramat Gan 52900, Israel
tomi@cs.biu.ac.il

Dana Shapira
Department of Computer Science
Ashkelon Academic College
Ashkelon 78211, Israel
shapird@ash-college.ac.il

Abstract

We explore the Full Compressed Delta Encoding problem in compressed texts, defined as the problem of constructing a delta file directly from the two given compressed files, without decompressing. We concentrate on the case where the given files are compressed using the LZSS compression technique and propose solutions for the special cases involving substitutions only.

1. Introduction

Delta encoding is a way of storing or transmitting data in the form of differences between two given files. This is a widely studied subfield of data compression, as can be seen by the large number of pubications on the topic, see [2, 3, 6, 7, 9, 8, 13], to cite just a few. The *Compressed Differencing* Problem is of constructing a delta file of the two given original files out of their compressed forms in time proportional to the size of the input, i.e., without decompressing the compressed files. That is, let S be the source file and T be the target file (in many applications, these are two versions of the same file), and suppose we are given their compressed forms $E(S)$ and $E(T)$ as input. Our goal is to construct a new file $\Delta(S,T)$ which is the differencing file of S and T, but without decompressing, that is, without using S or T themselves. The problem, called also the *Full Compressed Differencing Problem*, was first introduced in the work of Klein et al. [10], along with another, reduced, variant called the *Semi Compressed Differencing Problem*, in which only one of the files is given in its compressed form. If none of the input files are compressed, we are back to the original problem of differencing.

A motivation for this problem is, first, reducing the number of I/O operations, by transmitting a (shorter) delta file instead of the compressed target file when the compressed source file is already shared by the transmitter and receiver. Another application is detecting resemblance in a set of files when they are all given in their compressed forms. Using techniques for solving the Compressed Differencing Problem, this can be done without decompression, since delta files that are smaller than the underlying compressed files indicate resemblance between the original files.

If both files, S and T, are compressed using static Huffman coding (or any other static method), generating the delta file can be done by applying the same algorithms directly on the compressed files. The size of the output delta file is at least as small as the size of the delta encoding generated on the original files S and T, since the corresponding compressed forms of common substrings of S and T are still common substrings of the compressed versions. The reverse, however, is not necessarily true, as the common substrings of the compressed forms can exceed codeword boundaries. Consider for example a Huffman code in which {`00, 01`} are the codewords assigned to `a` and `b`, respectively. Let S be `bab` and T be `baa`, then $E(S)$ is `010001` and $E(T)$ is `010000`. The substring `ba` is the longest both S and T have in common, and it is encoded by `0100`. This encoded substring, however, can be extended in the compressed form of the files to include also the following bit `0`, since `01000` is still common to both encoded strings.

The problem is harder when using adaptive compression methods such as Lempel-Ziv techniques. The encoding of a substring depends also on its environment in the file rather than on its characters only. Thus a recurring substring is not necessarily encoded identically throughout the text. Our goal is to identify reoccurring substrings in the compressed form so that we can replace them by pointers to previous occurrences. In [10] we explored the compressed differencing problem on LZW compressed files and showed that in order to achieve efficient delta compression, one may need some partial decoding. In the present work, we deal with the Full Compressed Differencing Problem, which is less trivial than the semi compressed variant, and concentrate on the case where the original files are compressed using LZSS [14]. Since basic Delta encoding techniques use generally copy and add commands, there seems to be a closer relation between Delta encoding and LZSS than with LZW.

Section 2 shows the difficulties of solving the problem in its general form. We therefore restrict our attention to the special case of substitutions only. Section 3 presents a first attempt to solve the restricted problem, using partial decoding. In Section 4 any usage of decoding is removed and algorithms for a single and an unknown number of substitutions are suggested, based on the compressed files alone. Experiments then show that the damage to the delta file is reasonable.

2. Difficulties in Solving the LZSS Compressed Delta Encoding

The LZSS compression algorithm represents a given file, T, as a sequence of copies and single characters. The copies are described in the form of ordered pairs (*off*, *len*), meaning 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, and the length of this substring (the number of its characters) is *len*.

The Delta file $\Delta(S,T)$ can be formed using a similar format of individual characters and copy items, but copies can now refer either to S or to T itself. This can be implemented using a flag bit, for example, (1, *off*, *len*) could refer to copies of substrings in S, while (0, *off*, *len*) would indicate that the copy is from T. The value of *off* is taken relative to the current position in T, thus an additional difference is that unlike the offset component in T that refers to characters appearing before the current position, the offset component in S can also refer to characters ahead; an extra flag-bit may encode this choice.

The following notation is used:

$$E(S) = s_1 s_2 \cdots s_u \qquad \text{and} \qquad E(T) = t_1 t_2 \cdots t_v,$$

where each of the s_i and t_i is either a codeword of the form (o_i, ℓ_i) for a copy codeword, or it is an individual character. For consistency, if s_i or t_i is an individual character, we define $\ell_i = 1$. When necessary, the ℓ_i component will appear with a superscript as ℓ_i^s or ℓ_i^t. Greedy LZSS compression is used, i.e., for each position in the file, the longest possible match between the string starting at the current position and the already scanned part of S or T is located.

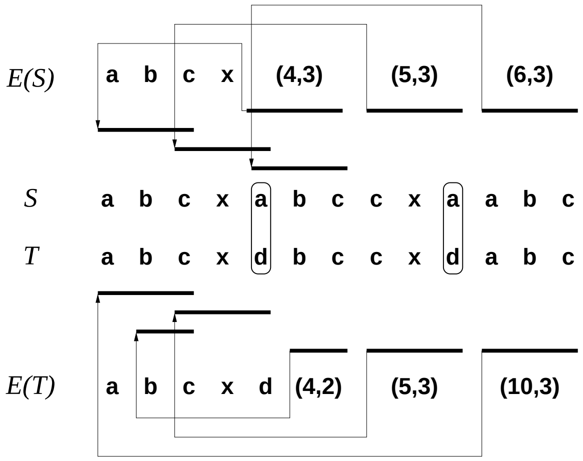

FIGURE 1: *example*

The difficulty of detecting matches between two files S and T which are given in their LZSS compressed form is illustrated in the example in Figure 1. Let the source file be $S =$ abcxabccxaabc and suppose the target file $T =$ abcxdbccxdabc is obtained by substituting a by d in positions 5 and 10, as indicated by the boxes in the Figure. The LZSS compressed forms are $E(S) =$ abcx(4,3)(5,3)(6,3) and $E(T) =$ abcxd(4,2)(5,3)(10,3) and are shown above, resp. below, S and T. Recurring strings are visualized by connected bars.

The first difference between $E(S)$ and $E(T)$ occurs at the fifth codeword, where $s_5 = (4,3)$ and $t_5 =$ d are referring to the substrings abc and d, respectively. After regaining synchronization, the following pair of codewords is (s_6, t_7), referring, respectively, to the substrings cxa in S and cxd in T. However, the codewords are identical, $s_6 = t_7 = (5,3)$, which shows that an identical codeword at corresponding locations of S and T does not necessarily imply underlying identical substrings. Moreover, different ordered pairs do not necessarily refer to different substrings, as can be seen in the example with the following codewords, where the codewords $s_7 = (6,3)$ and $t_8 = (10,3)$ both refer to the same substring abc. The expected output would be a compact delta file, which for our example could be (1,0,4)d(1,0,4)d(1,0,3). Note that in this example only substitutions occur, therefore all *off* parameters are 0, as they are taken relative to the current position in T.

These problems suggest that it may be difficult to solve the Compressed Delta Encoding problem in its most general form, where any sequence of edit operations, i.e., character insertions, deletions or substitutions, is allowed when passing from one file to the other. We therefore relax the problem to take care of a file T that was generated from another file S using only a limited set of edit operations, and shall deal in the sequel with **substitutions only**. The restriction to substitutions can be justified in certain applications, for example when dealing with updates of web pages, where only a small number of fixed length fields, such as the current date or time, are updated. Another suitable application is successive versions of backups in databases which are constraint to a rigid layout, so that only a small number of fixed length changes are present. Restricting a problem to specific special cases has also been done for pattern matching applications, such as variants dealing with a known or unknown number of don't cares, swaps or mismatches [11, 1, 4, 5]. In the next section, we try, in a first attempt, to overcome the difficulties by using some degree of partial decoding.

3. Compressed Delta Encoding with Partial Decoding

Assuming substitutions only, one can try and keep the pointers into the compressed files in some synchronization. First we note that, as shown in the example above, the *off* component in pointers to S will always be zero and can thus be omitted. Further, one can control synchronization by keeping track of the lengths of the copied items in both S and T and assuring that $\sum_i \ell_i^s = \sum_j \ell_j^t$, by advancing the pointer (into S or T) that is lagging behind.

In order to detect all the differences between the two given files, we consider pairs of corresponding codewords from $E(S)$ and $E(T)$, and issue a copy command for substrings pointed to by differing codewords, even if the substrings are identical. Once the first difference is located at position p, each ordered pair (o, ℓ) is examined to see whether it refers to a string including position p. If c is the current location in T, then we must check whether $c-o \leq p < c-o+\ell$. We shall refer to this operation as an *intersection* between the ordered pair (o, ℓ) and the list of changes, consisting, after the first difference, only of $\{p\}$.

The general algorithm is given in Figure 2. It partially decodes selected parts of the compressed inputs using the decoding procedure D, which is the inverse of the encoding E. The differences between the decoded strings are recorded in a list called *Changes*, and the remaining substrings of T are output to the delta file as copies of the corresponding substrings of S. The pointers into S and T, denoted by Sp and Tp, respectively, are then advanced. The algorithm uses a variable *length* to accumulate the length of the matching string that is currently expanded. The function *update*, given in Figure 3, compares two corresponding decoded substrings of S and T from position *start* to position *end*.

$i \longleftarrow 1 \qquad j \longleftarrow 1$ // $E(S)$ and $E(T)$ indices
$Sp \longleftarrow 1 \qquad Tp \longleftarrow 1$
$Changes \longleftarrow \emptyset$
while not EOF $E(T)$
{
    $length \longleftarrow 0$
    if $s_i = t_j$ and $t_j \cap Changes = \emptyset$
        $length \longleftarrow length + \ell^t_j$
    elseif $D(s_i) \neq D(t_j)$ // partial decoding
        $update\Big(Sp, Sp + \min(|D(s_j)|, |D(t_j)|) - 1\Big)$
    $Sp \longleftarrow Sp + \ell^s_i \qquad Tp \longleftarrow Tp + \ell^t_j$
    $i++ \qquad j++$
    while $Sp \neq Tp$ // search synchronization point
    {
        if $Sp < Tp$ // advance in $E(S)$
            $update(Sp, Tp)$
            $Sp \longleftarrow Sp + \ell^s_i$
            $i++$
        else // $Tp < Sp$, advance in $E(T)$
            $update(Tp, Sp)$
            $Tp \longleftarrow Tp + \ell^t_j$
            $j++$
    }
}

FIGURE 2: *Compressed Differencing algorithm with partial decoding*

$update(start, end)$
{
    for $k \longleftarrow start$ to end
        if $S[k] \neq T[k]$
            concatenate k to $Changes$
            if $length > 0$
                output $(1, length)$ // copy from S
                output $T[k]$
                $length \leftarrow 0$
        else
            $length \longleftarrow length + 1$
}

FIGURE 3: *Comparing corresponding decompressed substrings of S and T*

If the number of changes is n and the number of elements in the compressed file $E(T)$ is v, then the total time spent on the intersections is $O(v \log n)$: the indices of the substitutions are discovered in order and can be kept in a balanced search tree, and for each element t_j, two searches in this tree are sufficient to verify whether one of the indices falls into the corresponding range in T. In practice, however, the influence of a small number of substitutions will be locally restricted, as can be seen in Figure 4, which shows the number of changes in the LZSS file as function of the number of substitutions in the original for several test files.

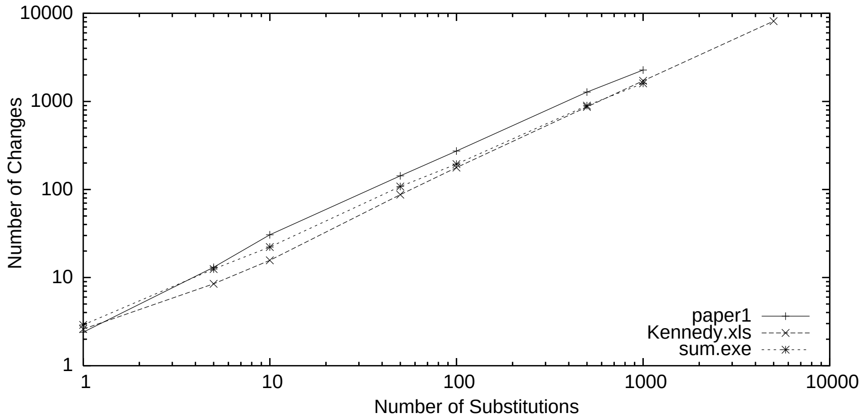

FIGURE 4: *Substitutions as function of changes*

For the experiment we considered different file types: text, spreadsheet and executable, choosing `paper1` of the Calgary Corpus and `Kennedy.xls` and `sum.exe` of the Canterbury corpus. We then randomly created substitutions on the file. A single substitute was created by first choosing two positions in the file and overwriting the character that appears at the first position with that appearing at the second position. Note that in realistic applications, changes tend to be more clustered, so assuming the substitutions are uniformly distributed only worsens the results. This process was repeated for the desired number of changes. We then constructed the two LZSS compressed forms of these files. The number of changes in the compressed files was computed as follows: the two compressed versions were scanned in parallel keeping them synchronized. That is, once a mismatch was detected, the number of changes was increased and the pointers of the two files were moved to the same position in the original files, before continuing with the count. We considered different numbers of changes, repeated each experiment 10 times and averaged the results. The graphs show almost straight lines, implying that a small number of substitutions, which we assume when talking about similar files for which delta encoding is suitable, cannot, generally, cause a large number of changes.

The main problem in the partial decoding algorithm stems from the fact that LZSS is a left to right decoding algorithm while decoding here proceeds from right to left. That is, suppose we are decoding an ordered pair $t_j = (o, \ell)$ which corresponds to positions $c - o$ to $c - o + \ell - 1$ in the original file; in the compressed file, the corresponding positions may again contain some (o, ℓ) codewords, which must be recursively decoded (right to left) as part of the decoding of t_j. For example, consider the LZSS encoding abcde(5,5)fg(4,3)(14,3)(7,3). To decode $t_{10} = (14, 3)$, one has to go backwards and sum up the number of individual characters and the length components of the copy items. In this example t_{10} points directly to the substring bcd. In contrast, the range pointed to by $t_9 = (4, 3)$ covers both a part of $t_6 = (5, 5)$ and $t_7 =$ f, so t_6 has to be decoded recursively. Although decoding here proceeds "on demand", in a worst case scenario, the partial decoding algorithm may become full decoding. Moreover, if no additional space is used, the same codeword might be decoded more than once, as can be seen when decoding $t_{11} = (7, 3)$, pointing to $t_9 = (4, 3)$, which might have already been decoded. This can be solved by storing each decoded substring in a buffer, thus saving redundant decoding operations and recursive calls.

In the following section we propose an alternative algorithm which does not necessarily produce an optimal delta file, and the size of the delta file can be larger than the size of that produced by the partial decoding algorithm, but on the other hand, it works without any decompression.

4. Compressed Delta Encoding without Decoding

4.1 Compressed Delta for a Single Substitutions

Given are two files S and T, where T differs from S only by a few substitutions, and we here assume first a single substitution only. Our goal is to construct the Delta file of T with respect to S. Let us assume that the first $k-1$ codewords $s_1, s_2 \cdots s_{k-1}$ of $E(S)$ are identical to the first $k-1$ codewords $t_1 t_2 \cdots t_{k-1}$ of $E(T)$, that $s_k \neq t_k$, and define $\ell = \sum_{i=1}^{k-1} \ell_i$. Using the fact that both files were compressed using the same LZSS compression algorithm, we conclude that the substitution occurs at the kth codeword. If t_k is a single character, then $\Delta(S, T)$, the delta file of T with respect to S, is $(1, \ell) t_k (1, |E(T)| - \ell - 1)$, meaning that the delta file consists of two (possibly empty) copies of substrings of S, inserting the character t_k between them. In fact, even if t_k is a copy item, we can use the same delta file. In this case t_k refers to a substring that occurred earlier in T itself. The fact that the first part of $\Delta(S, T)$, the pair $(1, \ell)$, refers a substring in S is not confusing, since on the corresponding parts, the files S and T are matching. Notice that the construction of the delta file is the same even if we allow self references in the LZSS compressed file. The algorithm is given in Figure 5.

4.2 Compressed Delta Encoding for an unknown number of Substitutions

To generalize the algorithm above, we consider k substitutions ($k \geq 1$) and refer to the second one. It can either be detected when two corresponding codewords are different, or when a copy item t_j in $E(T)$, though being equal to s_j in $E(S)$, refers to a substring affected by the first substitution. Consider then the case when different copies refer to identical substrings: this can be detected only by decoding.

$i \longleftarrow 1$
$length \longleftarrow 1$
while not EOF $E(T)$ and $s_i = t_i$
    $i{+}{+}$
    $length \longleftarrow length + \ell_i$
if $i > 1$
    output$(1, length)$
output $(0, t_i)$ // output the change preceded by a 0 bit
if $i \neq |E(T)|$
    output$(1, |E(T)| - length - \ell_i)$

FIGURE 5: *Compressed Differencing algorithm for a single substitution*

$i \longleftarrow 1$ $j \longleftarrow 1$ // $E(S)$ and $E(T)$ indices
$Sp \longleftarrow 1$ $Tp \longleftarrow 1$
$Changes \longleftarrow \emptyset$
while not EOF $E(T)$
{
    $length \longleftarrow 0$
    while $s_i = t_j$ and $t_j \cap Changes = \emptyset$
    {
        $i{+}{+}$ $j{+}{+}$
        $Sp \longleftarrow Sp + \ell_i^s$ $Tp \longleftarrow Tp + \ell_j^t$
        $length \longleftarrow length + \ell_j^t$
    }
    if $length > 0$
        output $(1, length)$
    output $(0, t_j)$ // output the change preceded with a 0 bit
    concatenate t_j to $Changes$
    while $Sp \neq Tp$ // search synchronization point
    {
        if $Sp > Tp$ // advance in $E(S)$
            $Sp \longleftarrow Sp + \ell_i^s$
            $i{+}{+}$
        else // $Tp > Sp$, advance in $E(T)$
            $Tp \longleftarrow Tp + \ell_j^t$
            $j{+}{+}$
            output $(0, t_j)$
    }
}

FIGURE 6: *Compressed Differencing for an unknown number of substitutions*

The decoding operations can be avoided at the price of a larger than necessary delta file. In fact, since the substrings are identical,the currently constructed copy from S could be extended as in the case $s_i = t_j$, but instead, we keep the item t_j and simply output this redundant copy to the delta file.

The other case when two identical ordered pairs refer to different substrings should be detected for the correctness of the constructed delta file. This only happens when ordered pairs point to previous substitutions. Although such ordered pairs do not necessarily refer to actual substitutions, we nevertheless append them to the list of detected substitution candidates and output them as copies to the delta file, as can be seen in the algorithm in Figure 6.

5. Experiments

Figure 7 presents the compression performance of the algorithm proposed in Figure 6. We considered the same set of files as above, with various numbers of substitutions. Beside the graphs for the Compressed Delta Algorithm (CDA), we included also graphs for the file size obtained by Unix' XDelta and for LZSS(T). The following encoding was used for CDA: characters occupied a single byte plus a flag bit; for copies from T, ordered pairs were encoded using a static format — 4 bits for the length component and 12 bits for the offset component (based on the constants of Nelson [12]); the length component in copies from S were encoded, depending on their size, by 5, 8 or 15 bits, using, respectively, the flags 0, 10 and 11 to identify the different cases. The figures are given in bytes.

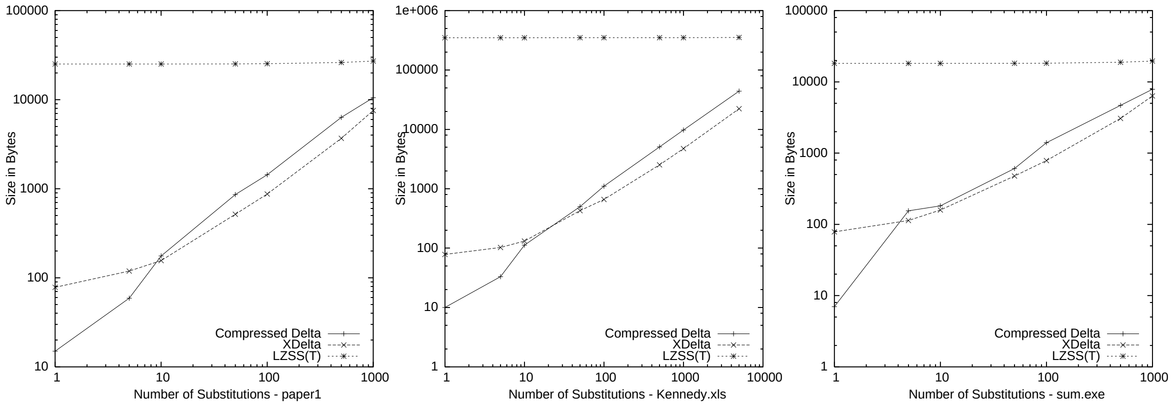

FIGURE 7: *Comparing file sizes*

We see that consistently, CDA is preferable to XDelta for a very small number of substitutions, but as this number increases, the file produced by CDA gets larger than the corresponding XDelta one. This is not surprising as one should bear in mind that CDA works on the compressed versions alone, and there is a price to pay for this restriction. Both CDA and XDelta outperform by far LZSS applied on T, which does not take advantage of the similarity between T and S.

References

[1] Abrahamson K.R. (1987). Generalized String Matching, *SIAM J. Comput.* **16**(6), 1039–1051.

[2] Agarwal R.C., Gupta K., Jain S., Amalapurapu S. (2006). An approximation to the greedy algorithm for differential compression, *IBM Journal of Research and Development* **50**(1), 149–166.

[3] Ajtai M., Burns R., Fagin R., Long D.D.E. (2002). Compactly Encoding Unstructured Inputs with Differential Compression, *Journal of the ACM* **49**(3), 318–367.

[4] Amir A., Landau G., Lewenstein M., Lewenstein N. (1998). Efficient Special Cases of Pattern Matching with Swaps, *Information Processing Letters* **68**(3), 125–132.

[5] Amir A., Lewenstein M., Porat E. (2004). Faster algorithms for string matching with k mismatches, *J. of Algorithms* **50**(2), 257–275.

[6] Burns R.C., Long D.D.E., (1997). Efficient Distributed Backup with Delta Compression, *Workshop on I/O in Parallel and Distributed Systems (IOPADS)*, 27–36.

[7] Burns R.C., Long D.D.E., (1998). In-place Reconstruction of Delta Compressed Files, *Proc. ACM Conference on the Principles of Distributed Computing (PODC),* 267–275.

[8] Factor M., Sheinwald D., Yassour B., (2001). Software Compression in the Client/Server Environment, *Proc. Data Compression Conference*, IEEE Computer Soc. Press, 233–242.

[9] Hunt J.J., Vo K.P, Tichy W., (1998). Delta algorithms: An Empirical Analysis, *ACM Trans. on Software Engineering and Methodology* **7**, 192–214.

[10] Klein S.T., Serebro T.C. and Shapira D., (2006). Modeling Delta Encoding of Compressed Files, *Prague Stringology Conference (PSC),* Prague, 162–170.

[11] Landau G.M., Vishkin U. (1986). Efficient String Matching with k mismatches, *Theor. Comput. Science* **43**, 239–249.

[12] Nelson M., Gailly J., (1995). *The Data Compression Book* 2nd edition, M & T Books, New York, NY.

[13] Shapira D., Storer J.A., (2005). In Place Differential File Compression, *The Computer Journal* **48**, 677–691.

[14] Storer J.A., Szymanski T.G., (1982). Data Compression via Textual Substitution, *Journal of the ACM* **29**(4), 928–951.