International Journal of Foundations of Computer Science

MODELING DELTA ENCODING OF COMPRESSED FILES

SHMUEL T. KLEIN
Department of Computer Science, Bar-Ilan University
52100 Ramat-Gan, Israel
tomi@cs.biu.ac.il

and

TAMAR C. SEREBRO
Department of Computer Science, Bar-Ilan University
52100 Ramat-Gan, Israel
t_lender@hotmail.com

and

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

Received (received date)
Revised (revised date)
Communicated by Editor's name

ABSTRACT

The Compressed Delta Encoding paradigm is introduced, i.e., delta encoding directly in two given compressed files without decompressing. Here we explore the case where the two given files are compressed using LZW, and devise the theoretical framework for modeling delta encoding of compressed files. In practice, although working on the compressed versions in processing time proportional to the compressed files, our target file may be considerably smaller than the corresponding LZW form.

Keywords: Differencing Encoding, Delta File, LZW

1. Introduction

Delta compression is a main field in data compression research. In this paper we introduce a new model of differencing encoding, that of *Compressed Differencing*. In this model we are given two files, at least one in its compressed form. The goal is to create a third file which is the differencing file (i.e. the delta file) of the two **original** files, in time proportional to the size of the input, that is, without decompressing the compressed files.

More formally, let S be the source file and T be the target file (probably two versions of the same file). The goal is to create a new file $\Delta(S,T)$ which is the differencing file of S and T. If both S and T are given in their compressed form, we call it the *Full Compressed Differencing Problem*. If one of the files is given in its compressed form we call it the *Semi Compressed Differencing Problem*. If none of the files are compressed, it refers to the original problem of differencing.

One motivation for this problem is when the encoder is interested in transmitting the compressed target file when both encoder and decoder have the source file in its compressed or uncompressed form. Creating the Delta file can reduce the transmitted file's size and therefore the number of I/O operations. Working on the compressed given form, the encoder can save memory space as well as processing time. Another motivation is detecting resemblance of a set of files when they are all given in their compressed form without decompressing them, perhaps saving time and space. When the difference file's size is less than the target file it indicates resemblance.

Traditional differencing algorithms compress data by finding common strings between two versions of a file and replacing substrings by a copy reference. The resulting file is often called a *delta* file. Two known approaches to differencing are the Longest Common Sub-sequence (LCS) method and the edit-distance method. LCS algorithms find the longest common subsequence between two strings, and do not necessarily detect the minimum set of changes. Edit distance algorithms find the shortest sequence of edits (e.g., insert, delete, or change character) to convert one string to another. One application which uses the LCS approach is the UNIX diff utility, which lists changes between two files in a line by line summary, where each insertion and deletion involves only complete lines. Line oriented algorithms, however, perform poorly on files which are not line terminated such as images and object files.

Tichy [16] uses edit-distance techniques for differencing and considers the string to string correction problem with block moves, where the problem is to find the minimal covering set of T with respect to S such that every symbol of T that also appears in S is included in exactly one block move. Weiner [17] uses a suffix tree for a linear time and space left-to-right copy/insert algorithm, that repeatedly outputs a copy command of the longest copy from S or an insert command when no copy can be found. This left-to-right greedy approach is optimal (e.g., Burns and Long[5], Storer and Szymanski [15]). Hunt, Vo, and Tichy [11] 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. Factor, Sheinwald, and Yassour [7] employ Lempel-Ziv based compression to compress S with respect to a collection of shared files that resemble S; resemblance is indicated by files being of same type and/or produced by the same vendor, etc. At first the extended dictionary includes all shared data. They achieve better compression by reducing the set of all shared files to only the relevant subset. Based on these researches we construct the delta file using edit distance techniques including insert and copy commands. We also reference the already compressed part of the target file for better compression.

Ajtai, Burns, Fagin, and Long [2] and Burns and Long [5] present several differential compression algorithms for when memory available is much smaller than S and T, and present an algorithm named checkpointing that employs hashing to fit footprints of substrings of S into memory; matching substrings are found by looking only at their footprints and extending the original substrings forwards and backwards (to reduce memory, they may use only a subset of the footprints). Heckel [10] presents a linear time algorithm for detecting block moves using Longest Common Subsequences techniques. One of his motivations was the comparison of two versions of a source program or other file in order to display the differences. Agarwal et al.[1] speed up differential compression with hashing techniques and additional data structures such as suffix arrays. In our work we use tries in order to detect matches.

Burns and Long [6] achieve in-place reconstruction of standard delta files by eliminating write before read conflicts, where the encoder has specified a copy from a file region where new file data has already been written. Shapira and Storer [14] also study in-place differential file compression. The non in-place version of this problem is known to be NP-Hard, and they present a constant factor approximation algorithm for this problem, which is based on a simple sliding window data compressor. 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, achieved in-place without additional memory, with compression that compares well with existing methods (both in-place and not in-place). Our delta file is not necessarily in place, but minor changes (such as limiting the offset's size) can result in an in place version of the file.

If both files, S and T, are compressed using Huffman coding (or any other static method), generating the differencing file can be done in the traditional way (perhaps a sliding window) directly on the compressed files. The delta encoding is at least as efficient as the delta encoding generated on the original files S and T. Common substrings of S and T are still common substrings of the compressed versions of S and T. However the reverse is not necessary true, since the common substrings can exceed the codeword boundaries. For example, consider the alphabet $\Sigma = \{a, b, c\}$ and the corresponding Huffman code $\{$`00, 01, 1`$\}$. Let $S =$`abab` and $T =$`cbaa`, then $\mathcal{E}(\mathcal{S}) =$`00010001` and $\mathcal{E}(\mathcal{T}) =$`1010000`. A common substring of S and T is `ba` which refers to the substring `0100` in the compressed file. However, this substring can be extended in the compressed form to include also the following bit, as the LCS is 01000 in this case.

The problem is less trivial when using adaptive compression methods such as Lempel-Ziv compressions. The encoding of a substring is determined by the data, and depends on its location. For this reason the same substring is not necessarily encoded in the same way 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 this paper we explore the compressed differencing problem on LZW compressed files and devise a model for constructing delta encodings on compressed files. In Section 2 we perform Semi Compressed Differencing using compressed pattern matching, where the source file is encoded using the corresponding compression method. In Section 3 we present an optimal algorithm in terms of processing time for the Semi and Full versions of the compressed differencing problem using tries.

2. Delta encoding in compressed files using compressed pattern matching

Many papers have been written in the area of compressed pattern matching, i.e. performing pattern matching on the compressed form of the file. Amir, Benson and Farach [4] propose a pattern matching algorithm in LZW compressed files which runs in $O(n + m^2)$ processing time or $O(n \log m + m)$, where n is the size of the compressed text and m is the length of the pattern. Kida et al. [12], present an algorithm for finding all occurrences of multiple patterns in LZW compressed texts. Their algorithm simulates the Aho-Corasick pattern matching machine, and runs in $O(n+m^2+r)$ processing time, where r is the number of pattern occurrences. Compressed pattern matching was also studied in [9, 8, 13] and in many others. In this section we use any compressed pattern matching algorithm to perform compressed differencing using the same compression method. Given a file T and a compressed file $\mathcal{E}(\mathcal{S})$, our goal is to present T as a sequence of pointers and individual characters. The pointers point to substrings that either occur in S, or previously occurred in T. This can be done by processing T from left to right and repeatedly finding the longest match between the incoming text and the text of S or to the left of the current position in T itself, and replacing it by a pointer, or by a single character, if a match of two or more characters cannot be found.

The input of any given compressed pattern matching algorithm is a specific pattern P we are interested in searching for. Since we are interested in locating the longest possible match at the current position, we only know the position the pattern starts with but not the one it ends with. A naıve algorithm can, therefore, try to locate all substrings of T starting at the current position by concatenating the following character to the pattern each time a match in S or in the already scanned part of T can still be found. A formal algorithm is given in Figure 1. We use u to denote the length of the uncompressed file. We denote by $cpm(P, \mathcal{E}(\mathcal{S}))$ any compressed pattern matching algorithm for matching a given pattern P in the compressed file $\mathcal{E}(\mathcal{S})$. It returns the position of the (last) occurrence of P in the original text S, and 0 if such location is not found. Similarly, $pm(P, T)$ denotes any pattern matching algorithm for matching a given pattern P in T, which returns the position of the (last) occurrence of P in T, and 0 if no such location is found. If the match between the pattern starting at the current position and the compressed form of S or the previous scanned part of T can not be extended, we output the match we have already found. If this longest match is only of a single character we output the character at the current position, and advance the position in T by one.

For analyzing the processing time of the naive algorithm presented in Figure 1,

1 $i \longleftarrow 1$;
2 **while** $i \leq u$ **do**
{
2.1 $P \longleftarrow T_i$;
2.2 $j \longleftarrow 0$;
2.3 **while** $i + j \leq u$ **and**
$(pos1 \longleftarrow cpm(P, \mathcal{E}(\mathcal{S})) \neq 0$ **or** $pos2 \longleftarrow pm(P, T_1 \cdots T_{i-1}) \neq 0)$
2.3.1 $j \longleftarrow j + 1$;
2.3.2 $P \longleftarrow P \cdot T_{i+j}$; // concatenate the next character
2.4 **if** $j = 0$ // no match was found
2.4.1 output T_i;
2.5 **elseif** $pos1 \neq 0$; //output a pointer to S
2.5.1 output a pointer $(pos1, j)$;
2.6 **else** // output a pointer to T
2.6.1 output a pointer $(pos2, j)$;
2.7 $i \leftarrow i + j + 1$
}

Figure 1: Solving the Semi Compressed Differencing problem using Compressed Pattern Matching.

let us assume that the compressed pattern matching algorithm is optimal in terms of processing time. By the definition of Amir and Benson [3] of optimal compressed matching algorithms the running time is $O(n + m)$, where n is the size of the compressed text and m is the length of the pattern. Thus the total running time of the algorithm presented in Figure 1 is $O(u^2(n + m))$, even for optimal compressed pattern matching algorithms. Our goal is, therefore, to reduce the processing time. In the following chapter we concentrate on differencing in LZW files.

3. Delta Encoding in LZW Files

The LZW algorithm by Welch [18] is a common compression technique which is used for instance by the compress command of UNIX. The LZW algorithm parses the text into phrases and replaces them as pointers to a dictionary trie, i.e., the nodes of the trie are labeled by characters of the alphabet, and the string associated with each node is the concatenation of the characters on the path going from the root down to that node. The trie initially consists of all the characters of the alphabet. At each stage of the algorithm it looks for the longest match between the string starting at the current position and the previous scanned text. It then updates the dictionary trie to include the node corresponding to the string of the match concatenated with the following character in the text.

3.1. Semi Compressed Delta Encoding

```
1       construct the trie of E(S);
2       i ⟵ 1;
3       while i ≤ u
        {
3.1         P ⟵ T_i T_{i+1}...T_u;
3.2         Starting at the root, traverse the trie using P
3.3         when you encounter a leaf v corresponding to a prefix of length ℓ of P
3.3.1           output the position in S corresponding to v;
3.4         i ⟵ i + ℓ;
        }
```

Figure 2: Semi Compressed Differencing algorithm for LZW compressed files.

During LZW decompression an identical trie to the LZW compression trie is constructed. Although the LZW decompression algorithm takes linear time in size of the original file, if the reconstructed file is not needed, the construction of the dictionary trie can be done in time proportional to the size of the compressed file. Figure 2 presents an improved algorithm for constructing the delta file of the compressed file $\mathcal{E}(\mathcal{S})$ and a given file T. It uses the dictionary trie constructed by $\mathcal{E}(\mathcal{S})$. Starting from the root, it traverses the trie with T until it reaches a leaf. It then outputs the position of the string corresponding to the leaf (this information is kept in the nodes of the trie when it is constructed). The prefix of T corresponding to the matched string is truncated, and traversing the trie continues with the remaining part of T starting from the root repeatedly. Since the trie is initialized with nodes corresponding to all the characters of the alphabet, outputting the positions of these nodes to the delta file correspond to inserting individual characters. The processing time of this algorithm is $O(|\mathcal{E}(\mathcal{S})| + |\mathcal{T}|)$, which is linear in the size of the input. In order to improve the compression performance we can add pointers to the portion of T that has already been processed. This can be done by constructing the trie for T in addition to the trie for S. This is done in the algorithm for full compressed delta files of the next sub-section.

3.2. Full Compressed Delta Encoding

In this section we present a linear time algorithm for the Full Compressed Delta Encoding problem. Figure 3 presents the algorithm for constructing the delta file of S and T given $\mathcal{E}(\mathcal{S})$ and $\mathcal{E}(\mathcal{T})$. It constructs the dictionary trie of S recording for each node the position of the last occurrence of the corresponding string and its length (the depth of the node) and the first character of the corresponding string. In addition, each node of the trie records its code indicating the order the nodes were created, which also corresponds to the node's LZW code. *Dictionary*[*code*] returns a pointer to the node in the trie having the corresponding code. Since the original file T is not needed, we construct the trie of T in parallel to traversing the trie of S. New nodes are introduced during this phase of the algorithm when substrings of T correspond to nodes that are not present in the trie of S. When adding or updating a node (recording the position in T and changing its code) it outputs to the delta file the position and length of its parent, or the position and length of the node itself, depending on whether the node exists or not. This ordered pair $(position, length)$ can refer to a substring of S or T, depending on the last update of the output node. The variable $flag$ indicates whether the character k (the first character of the previous code) was written to the delta file in the preceding stage. If so, we should eliminate k from the current pointer by skipping a single character in the copy's position and shortening the copy's length by 1. The processing time of this algorithm is $O(|\mathcal{E}(\mathcal{S})| + |\mathcal{E}(\mathcal{T})|)$, which is again linear in the size of the input.

```
1       construct the trie of E(S);
2       flag <-- 0; // output character k
3       input oldcode from E(T);
4       while oldcode ≠ NULL // still processing E(T)
        {
4.1         input code from E(T);
4.2         node <-- Dictionary[oldcode];
4.3         if (Dictionary[code] ≠ NULL)
4.3.1           k <-- first character of Dictionary[code];
4.4         else
4.4.1           k <-- first character of node
4.5         if ((node has a child k) ) and (code ≠ NULL))
4.5.1           output the position+flag and length−flag
                                        corresponding to child k of node;
4.5.2           update position of child k of node;
4.5.3           flag <-- 1; // character k has been outputed
4.6         else
4.6.1           output the position+flag and length−flag corresponding to node;
4.6.2           add child k to node;
4.6.3           flag <-- 0; // output character k
4.7         oldcode <-- code;
        }
```

Figure 3: Full Compressed Differencing algorithm for LZW compressed files.

To improve the compression performance of the delta file, we can check whether each ordered pair of the form $(position, length)$ can be combined with its previous ordered pair, i.e., if two consecutive ordered pairs are of the form (i, ℓ_1) and $(i + \ell_1, \ell_2)$ where i denotes a position in S or T and ℓ_1 and ℓ_2 denote lengths, we combine them into a single ordered pair $(i, \ell_1 + \ell_2)$. The combined ordered pair can then be combined with successive ordered pairs.

Consider the following example: $S = \texttt{abccbaaabccba}$, $T = \texttt{ccbbabccbabccbba}$. Applying LZW we get that $\mathcal{E}(\mathcal{S}) = \texttt{1233219571}$ and $\mathcal{E}(\mathcal{T}) = \texttt{33221247957}$. The dictionary trie of $\mathcal{E}(\mathcal{S})$ and the combined dictionary of $\mathcal{E}(\mathcal{S})$ and $\mathcal{E}(\mathcal{T})$ are given in the following figures. In the combined trie, dotted nodes indicate new nodes that were introduced during the parsing of $\mathcal{E}(\mathcal{T})$. Bold numbers represent data that was updated during the parsing of $\mathcal{E}(\mathcal{T})$, and therefore corresponds to positions in T.

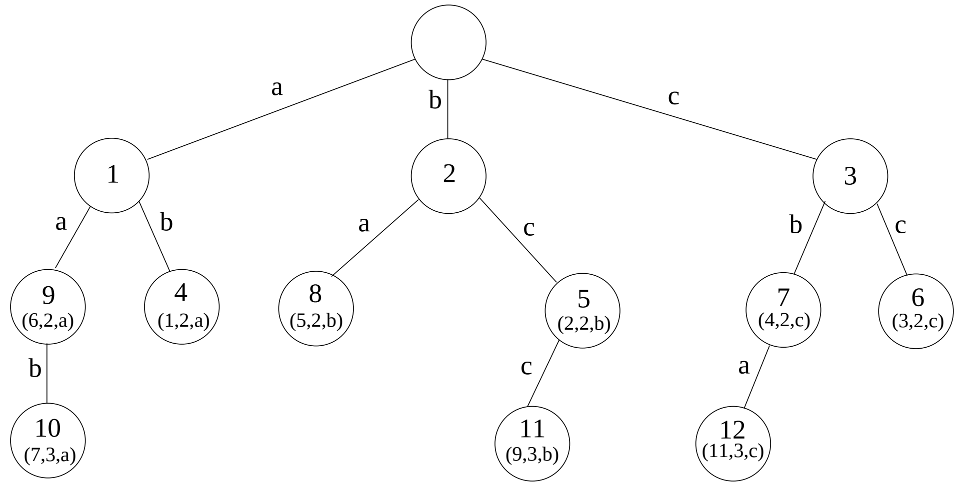

FIGURE 4: *The Dictionary Trie for* $\mathcal{E}(\mathcal{S})$

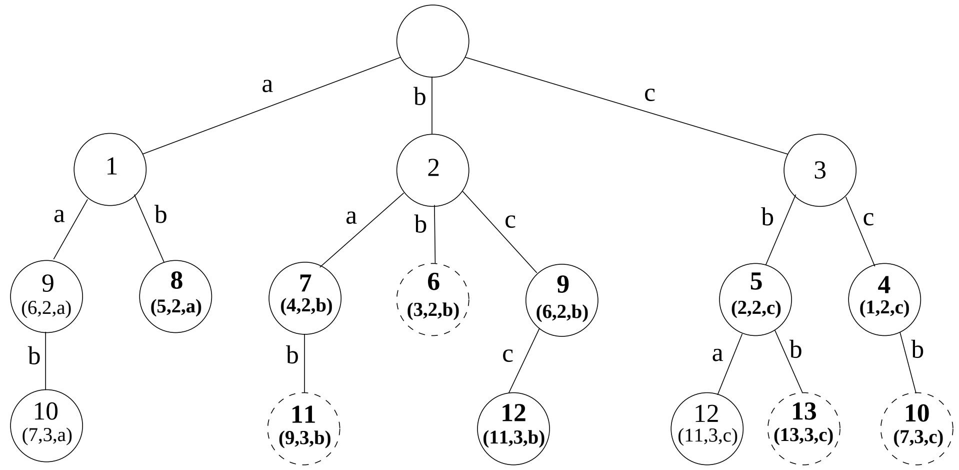

FIGURE 5: *The Combined Dictionary Trie for* $\mathcal{E}(\mathcal{S})$ *and* $\mathcal{E}(\mathcal{T})$

We get that $\Delta(S,T) = \langle 3,2\rangle\langle 5,1\rangle\langle 5,2\rangle\langle 2,1\rangle\langle 3,1\rangle(2,1)(4,2)\langle 9,3\rangle(3,1)(4,2)$, where pointers to S are delimited with brackets, and pointers to T with parentheses. Two ordered pairs can be combined thus $\Delta(S,T)$=$\langle 3,3\rangle\langle 5,2\rangle\langle 2,2\rangle\texttt{c}(4,2)\langle 9,3\rangle b(4,2)$.

At this stage ordered pairs of length 1 are translated to the corresponding character.

The compression performance of the above algorithm is similar to that of LZW. To improve that, we tried various degrees of partial decoding, and preliminary tests with these variants gave encouraging results. For example, using the executable file `xfig.3.2.1.exe` as source file S to compress the next version `xfig.3.2.2.exe`, playing the role of T, the resulting delta file was smaller than 3K, whereas the original size of T was 812K, `gzip` would reduce that only to 325K, and `LZW`, the method on which the delta encoding has been applied here, would yield a file of size 497K. The coding used was a combination of different Huffman codes for the characters, the offsets and the lengths. Non-compressed delta encoding could achieve even better results, but loose the advantage of working directly with the compressed files.

4. Future Work

Our exposition here has been mainly theoretical, presenting optimal algorithms (in the sense defined in [3]) for constructing delta files from LZW compressed data. We intend to explore the semi and full compressed delta encoding problems for LZ77 encoded files, both theoretically and practically. LZ77 based compressions resembles more to the basic delta encoding scheme, thus we expect that the compression performance will be better than LZ78 based compressions. Since the general form of the problem is difficult, in [**?**]we relax the problem to include only a restricted set of edit distance operations that were done when transforming the source file into the target file.

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] Ajtai, M., Burns, R. C., Fagin, R., and Long, D. D. E.: *Compactly encoding unstructured inputs with differential compression*. Journal of the ACM, 49(3) 2002, pp. 318–367.

[3] Amir, A. and Benson, G.: *Efficient two-dimensional compressed matching*, in Proceedings of the Data Compression Conference DCC–92, IEEE Computer Soc. Press, 1992, pp. 279–288.

[4] Amir, A., Benson, G., and Farach, M.: *Let sleeping files lie: Pattern matching in z-compressed files*. Journal of Computer and System Sciences, 52 1996, pp. 299–307.

[5] Burns, R. C. and Long, D. D. E.: *Efficient distributed backup and restore with delta compression*, in Workshop on I/O in Parallel and Distributed Systems (IOPADS), ACM, 1997.

[6] 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.

[7] Factor, M., Sheinwald, D., and Yassour, B.: *Software compression in the client/server environment*, in Proceedings of the Data Compression Conference,

IEEE Computer Soc. Press, 2001, pp. 233–242.

[8] Farach, M. and Thorup, M.: *String matching in lempel-ziv compressed strings*, in Proceedings of the 27th Annual ACM Symposium on the Theory of Computing, 1995, pp. 703–712.

[9] Gąsieniec, L. and Rytter, W.: *Almost optimal fully lzw-compressed pattern matching*, in Proceedings of the Data Compression Conference, IEEE Computer Soc. Press, 1999, pp. 316–325.

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

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

[12] Kida, T., Takeda, M., Shinohara, A., Miyazaki, M., and Arikawa, S.: *Multiple pattern matching in lzw compressed text.* Journal of Discrete Algorithms, 1(1) 2000, pp. 130–158.

[13] Navarro, G. and Raffinot, M.: *A general practical approach to pattern matching over ziv-lempel compressed text*, in Proceedings of the 10th Annual Symposium on Combinatorial Pattern Matching CPM–99, vol. 1645, LNCS, Springer Berlin / Heidelberg, 1999, pp. 14–36.

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

[15] J. A. Storer: *An Introduction to Data Structures and Algorithms*, Birkhauser/Springer, 2001.

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

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

[18] T. A. Welch: *A technique for high-performance data compression.* IEEE Computer, 17 June 1984, pp. 8–19.