Compressed Differential Encoding for Semi-Static Compressors

ASHUTOSH GUPTA[1]
PANKAJ[2]

M.J.P. Rohilkhand University
Department of Computer Science and Information Technology
Bareiily-India
[1]ashutosh333@rediffmail.com
[2]pankaj_mnnit@rediffmail.com

Abstract. The Compressed Differential Encoding is introduced, i.e., delta encoding directly in two specified compressed files without decompressing. In this paper, we investigate the case where the two given files are compressed using WBTC, and formulate the theoretical framework for modeling delta encoding of compressed files. In put into practice, even though working on the compressed versions in processing time proportional to the compressed files, our target file may be noticeably smaller than the corresponding WBTC form.

Keywords: Delta Encoding, Delta File, WBTC.

(Received September 17, 2010 / Accepted December 29, 2010)

1 Introduction

There has been an enormous expansion of textual information over web through digital libraries, office automation systems, electronic mail, web browsing etc. This requires organizing, managing and transporting of the data from one point to the other on data communications links with limited bandwidth. These facts show the importance of text compression.

The benefits of compression are two fold. First, it takes less space to store the information and therefore takes lesser time to transfer the compressed data through network. Second, it takes lesser bandwidth in comparison to transmitting raw data. Obviously, downloading the text in its compressed form also takes less time.

The current trend of research on text compression involves developing *Differential compression* or *delta compression* algorithms. Differential compression or delta compression is a method of compressing data that have a huge covenant of similarities. Differential compression produces a delta encoding, a scheme of representing a version file (V) in terms of an original file (S) plus new information. Thus differential compression algorithms attempt to efficiently locate common data between a reference and a version file to reduce the amount of new information which must be used. By storing the reference file and this delta encoding, the version file can be reconstructed when needed.

More formally, a differencing algorithm finds and outputs the changes made between two versions of the similar file by locating common strings to be copied and unique strings to be added explicitly. Let S be the source file and T be the target file. Probably these two are versions of the same file. The goal is to create a *delta file* ($\triangle$). The *delta file* ($\triangle$) is the encoding of the output of a differencing algorithm.. If both S and T are known in their compressed form, we call it the *Full Compressed Differencing* Problem. If one of the files is specified 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*.

The stimulus for this problem is when the encoder is fascinated 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 file size during transmission and therefore the number of I/O operations. Functioning on the compressed given form, the encoder can save memory space as well as processing time. Another motivation is identifying similarity of a set of files when they are all given in their compressed form without decompressing them, possibly saving time and space. When the file size of delta file is less than the target file it indicates similarity.

Conventional differencing algorithms compress data by identifying common strings between two versions of a file and substitute 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 come across the longest common subsequence between two strings, and do not inevitably detect the minimum set of changes. Edit distance algorithms find the shortest sequence of edits (e.g., insert, delete, or change character) to translate 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 outline, 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 [21] uses edit-distance techniques for differencing and considers the string to string rectification 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 incorporated in exactly one block move. Weiner [22] uses a suffx tree for a linear time and space left-to-right copy/insert algorithm that continually 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 [4], Storer and Szymanski [20]). 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 show that delta compression algorithms based on LZ techniques [13, 6, 8, 12, 17, 23] notably outperform LCS based algorithms in terms of compression performance.

Factor, Sheinwald, and Yassour [5]uses Lempel-Ziv based compression to compress S with respect to a group of shared files that look like S; similarity is indicated by files being of similar type and/or produced by the same vendor, etc. At first the comprehensive dictionary includes all shared data. They achieve improved compression by reducing the set of all shared files to only the appropriate subset.

Ajtai, Burns, Fagin, and Long [2] and Burns and Long [4] present a number of differential compression algorithms for when memory existing is much smaller than S and T, and present an algorithm named checkpointing that uses 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.

Heckel [10] presents a linear time algorithm for identifying 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.

Burns and Long [3] attain in-place reconstruction of standard delta files by eliminating write before read conflicts, wherever the encoder has specified a copy from a file region where new file data has already been written. Shapira and Storer [19] also study in-place differential file compression. They introduce a constant factor approximation algorithm for this problem, which is based on a simple sliding window data compressor. Inspired by the constant bound approximation factor they change the algorithm so that it is suitable for in-place decoding and present the In-Place Sliding Window Algorithm (IPSW). The benefit of the IPSW approach is simplicity and speed, achieved in-place without additional memory, with compression that compares sound with existing methods (both in-place and not in-place).

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 conventional way (perhaps a sliding window) straight on the compressed files. The delta encoding is at least as competent as the delta encoding generated on the original files S and T. Frequent 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 Σ = a, b, c and the corresponding Huffman code 11, 10, 0. Let S =abab and T =cbaa, then E(S) =11101110 and E(T) =0101111. A common substring of S and T is ba which refers to the substring 1011 in the compressed file. However, this substring can be extended in the compressed form to include also the following bit, as the LCS is 10111 in this case. Our delta file removes this limitation because of using *self-synchronized* codes. We have used Word-based encoding scheme given in [9].

In this paper we explore the compressed differencing problem on WBTC compressed files and devise a model for constructing delta encodings on compressed files. In Section 2 we briefly describe the word-based tagged code (WBTC), 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

2 Word based Compressors

As the distribution of frequencies of characters in natural language texts is not much biased [18], this results in poor compression. Moffat [15] uses the words of the text as units to be compressed. Taking words as units means that the table of symbols in the compressor is exactly the vocabulary of the text, which makes it easier and faster to integrate compression with inverted indexes [7, 24, 14]. The vocabulary of the compressor is not only useful for searching a pattern in compressed text but also permits faster decompression. Since in natural language text the word frequency distribution is much more biased than that of characters, the idea proposed by Moffat works well because a text is more compressible when regarded as a sequence of words rather than that of characters.

A natural language text is composed of words and separators. An efficient way to deal with words and separators is to use a method called spaceless words [16]. If a word is followed by a space, just the word is encoded. If not, the word and then the separator are encoded. At decoding time, it is presumed that a space follows each word, except if the next symbol corresponds to a separator.

As words are considered as basic units of text information retrieval systems, we use a word-based tagged code [9]. The code generated by this technique always ends with either 01 or 10. This implies that the bit combination 01 or 10 act as a flag to indicate the end of code.

The coding procedure of Word based Tagged Code is simple. First, source text is parsed and all the statistics of vocabulary in the text is gathered. The vocabulary is sorted with non increasing frequency. Each codeword in WBTC will be generated with the help of 2 bit patterns (00, 11, 01, 10). Following procedure is used for the assignment of codes to the vocabulary.

1. At the very first level l=1, the first 2^l words (rank: 0 to 2^0) of the vocabulary are assigned codes as 01 and 10 respectively.
2. For the next level l=2, 2^l words in positions from $2^1 + 0$ to $2^2 + 2^0$ are encoded using four bits by adding 00 and 11 as prefix to all the codes of previous level.
3. In general, for any value of level l, next 2^l words present in the positions from $2^{l-1}+(2^{l-2} + ...+0)$ to $2^l+(2^{l-1}+...+2^0)$ of vocabulary are assigned codes using $2 \times l$ bits, by adding 00 and 11 as prefix to all the codes generated at preceding level.
4. The above procedure is repeated until all the N words are encoded.

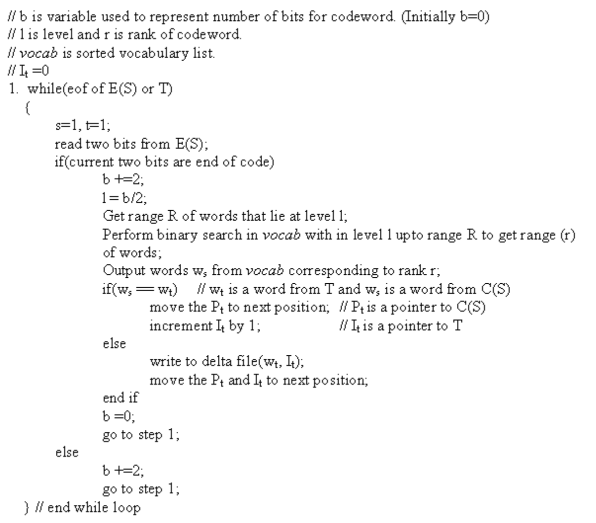
The feature of this coding scheme is that the code depends only upon the rank of the words, rather than their actual frequency. In this coding scheme, we neither to store the frequencies nor the codewords are required to store along with compressed file. This makes vocabulary very small as compared to the case of Huffman code, where either the codewords in form of tree or the frequencies must be stored with the vocabulary. The coding technique is a prefix code, no codeword is a proper prefix for any other code, and hence it is instantaneously decodable.

3 Delta encoding in WBTC Files

The WBTC algorithm [9] computes the codewords for all the words of the sorted vocabulary and stores them in a data structure named as CodeVector. In this way, the first pass generates codes for all the vocabulary words of the input text. In second pass, algorithm read the words and assigns their codes from the CodeVector generated in first pass. Thus, compressed file is generated. The vocabulary is stored along with the compressed text in order to decompress it later. There is no need to store either the codewords (in any form such as a tree) or the frequencies in the compressed file. It is enough to store the plain words sorted by frequency. Therefore, the vocabulary will be slightly smaller because no extra information other than the words alone is here.

3.1 Semi-Compressed Differential Encoding

During WBTC decompress, first load the words that compose the vocabulary in a separate data structure. Since these words were stored in sorted form (with respect to frequency) along with the compressed text during compression, the vocabulary which is retrieved is sorted by frequency. Once the sorted vocabulary is obtained the decoding of codewords can start by replacing the codwords to corresponding vocabulary words. Decompression takes O(v) time, being v the size in bytes of the compressed text. Figure 1 presents an algorithm for constructing the delta file of the compressed file C(S) and a given file T. It uses the vocabulary *vocab* constructed by C(S). Starting from the initial index of C(S), it traverses the compressed file C(S) with T until either C(S) or T reaches end-of-file. It then outputs the position of the code corresponding to the word of T, if word corresponding to the current code of C(S) is different than word in T. The processing time of this algorithm is O(|C(S)|+|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 creating a doublet (wt, It) for T. In next subsection, we present an algorithm for full compressed delta files.



```
// b is variable used to represent number of bits for codeword. (Initially b=0)
// l is level and r is rank of codeword.
// vocab is sorted vocabulary list.
// $I_t$ =0
1.  while(eof of E(S) or T)
    {
        s=1, t=1;
        read two bits from E(S);
        if(current two bits are end of code)
            b +=2;
            l = b/2;
            Get range R of words that lie at level l;
            Perform binary search in vocab with in level l upto range R to get range (r)
            of words;
            Output words $w_s$ from vocab corresponding to rank r;
            if($w_s$ == $w_t$)    // $w_t$ is a word from T and $w_s$ is a word from C(S)
                move the $P_t$ to next position;  // $P_t$ is a pointer to C(S)
                increment $I_t$ by 1;             // $I_t$ is a pointer to T
            else
                write to delta file($w_t$, $I_t$);
                move the $P_t$ and $I_t$ to next position;
            end if
            b =0;
            go to step 1;
        else
            b +=2;
            go to step 1;
    } // end while loop
```

Figure 1: Semi Compressed Differential algorithm for WBTC compressed file.

3.2 Full Compressed Differential Encoding

In this section we present a linear time algorithm for the Full Compressed Delta Encoding problem. Figure 2 shows the algorithm for constructing the delta file of S and T given E(S) and E(T). First, the vocabulary is loaded into the memory. The decoder reads two bits at a time from E(S) and E(T). The WBTC coding is not a suffix free scheme. It may be possible that code c1 can be a suffix of code c2. Considering this fact, there may be four cases arise.

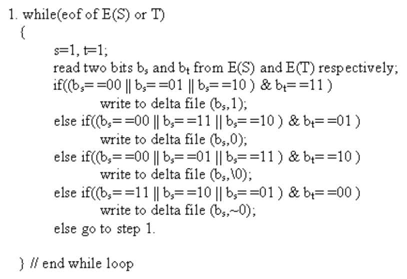
Case 1: two bit pattern in E(S) may be 00 or 01 or 10 and two bit pattern in E(T) is 11. The decoder sends a doublet (bits_counter, 1). Here bit 1 indicates pair of bits 11. The bit_counter indicates the number of 2 bit pair already parsed.

Case 2: two bit pattern in E(S) may be 00 or 11 or 10 and two bit pattern in E(T) is 01. The decoder sends a doublet (bits_counter, 0). Here bit 0 indicates pair of bits 01.

Case 3: two bit pattern in E(S) may be 00 or 01 or 11 and two bit pattern in E(T) is 10. The decoder sends a doublet (bits_counter,
0). Here escape symbol (
) followed by bit 0 indicates pair of bits 10.

Case 4: two bit pattern in E(S) may be 11 or 01 or 10 and two bit pattern in E(T) is 00. The decoder sends a doublet (bits_counter, $\tilde{0}$). Here s̃ymbol followed by bit 0 indicates pair of bits 00.

The processing time of this algorithm is O(|E(S)|+|E(T)|), which is again linear in the size of the input. Consider the following example: S = abc cba a abc cba, T = ccbb cba b abc ccb. The vocabulary *vocab* and *CodeVector* corresponding to S and T is given in Table 3.2.



```
1. while(eof of E(S) or T)
   {
        s=1, t=1;
        read two bits $b_s$ and $b_t$ from E(S) and E(T) respectively;
        if(($b_s$==00 || $b_s$==01 || $b_s$==10 ) & $b_t$==11 )
              write to delta file ($b_s$,1);
        else if(($b_s$==00 || $b_s$==11 || $b_s$==10 ) & $b_t$==01 )
              write to delta file ($b_s$,0);
        else if(($b_s$==00 || $b_s$==01 || $b_s$==11 ) & $b_t$==10 )
              write to delta file ($b_s$,\0);
        else if(($b_s$==11 || $b_s$==10 || $b_s$==01 ) & $b_t$==00 )
              write to delta file ($b_s$,~0);
        else go to step 1.

   } // end while loop
```

Figure 2: Full Compressed Differencing algorithm for WBTC compressed file.

Table 1: vocabulary and Codevector.

vocab	CodeVector
abc	01
cba	10
a	0001
ccbb	0010
b	1101
ccb	1110

Applying WBTC we get that E(S) = 01 10 0001 01 10 and E(T) = 0010 10 1101 01 1110. The delta file ($\triangle$) corresponding to E(S) and E(T) will be : $(1,\tilde{0})(3,$ $0)(4,1)(6,0)(7,01)(8,11)(9,10)$. As E(S) is reached to

EOF first, rest of the bits are sent to delta file as it is along with their respective positions. The doublets (7,01)(8,11)(9,10) corresponds to this case.

The compression performance of the above algorithm is similar to that of WBTC. The preliminary tests gave hopeful results. For example, using the books book1 as source file S to compress the book2, playing the role of T, the resulting delta file was smaller than 2K, whereas the original size of T was 596K, gzip would reduce that only to 204K, and WBTC, the method on which the delta encoding has been applied here, would yield a file of size 218K. Non-compressed delta encoding could attain even improved results, but slack the benefit of functioning directly with the compressed files.

4 Future Work

Our explanation here has been primarily theoretical, presenting optimal algorithms for constructing delta files from WBTC compressed data. We propose to discover the semi and full compressed delta encoding problems for WBTC encoded files, both in theory and practically. WBTC based compression resembles more to the basic delta encoding scheme, thus we expect that the compression performance will be better than adaptive based compressions.

References

[1] Agarwal, A. S., R. C. and Jain, S. An approximation to the greedy algorithm for differential compression of very large files. Technical report, IBM Alamaden Res. Center, 2003.

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

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

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

[5] Factor, S. D., M. and Yassour, B. Software compression in the client/server environment. In *in Proceedings of the Data Compression Conference, IEEE Computer Soc. Press,*, pages 233–242, 2001.

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

[7] G. Navarro N. Ziviani, E. M. and Baeza-Yates., R. Compression: A key for next- generation text retrieval systems. *IEEE Computer*, 33(11), 2000.

[8] Ga¸sieniec, L. and Rytter, W. Almost optimal fully lzw-compressed pattern matching. In *in Proceedings of the Data Compression Conference, IEEE Computer Soc. Press*, pages 316–325, 1999.

[9] Gupta, A. and Agarwal., S. A scheme that facilitates searching and partial decompression of textual documents. *Intl. Journal of Advanced Computer Engineering*, 1(2), 2008.

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

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

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

[13] Klien, S. T. and Shapira., D. Compressed delta encoding for lzss encoded files. *IEEE DCC*, 2007.

[14] M. Neubert Nivio Ziviani Gonzalo Navarro, E. S. d. M. and Baeza-Yates., R. Adding compression to block addressing inverted indexes. *Information Retrieval*, 3(1):49–77, 2000.

[15] Moffat., A. Word based text compression. *Software Practice and Experience*, 1989.

[16] N. Ziviani E. Silva de Moura, G. N. and Baeza-Yates., R. Fast and flexible word searching on compressed text. *ACM Transaction on Information Systems*, 18(2):113–139, 2000.

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

[18] Salomon., D. *Data Compression.* Springer Verlag, 1998.

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

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

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

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

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

[24] Witten, A. M. I. and Bell., T. *Managing Gigabytes.* Morgan Kaufmann Publishers Inc., second edition, 1999.