Compressed Transitive Delta Encoding

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

Abstract

Given a source file S and two differencing files $\Delta(S,T)$ and $\Delta(T,R)$, where $\Delta(X,Y)$ is used to denote the delta file of the target file Y with respect to the source file X, the objective is to be able to construct R. This is intended for the scenario of upgrading software where intermediate releases are missing, or for the case of file system backups, where non consecutive versions must be recovered. The traditional way is to decompress $\Delta(S,T)$ in order to construct T and then apply $\Delta(T,R)$ on T and obtain R. The *Compressed Transitive Delta Encoding (CTDE)* paradigm, introduced in this paper, is to construct a delta file $\Delta(S,R)$ working directly on the two given delta files, $\Delta(S,T)$ and $\Delta(T,R)$, without any decompression or the use of the base file S. A new algorithm for solving CTDE is proposed and its compression performance is compared against the traditional "double delta decompression". Not only does it use constant additional space, as opposed to the traditional method which uses linear additional memory storage, but experiments show that the size of the delta files involved is reduced by 15% on average.

1. Introduction

Differential file compression represents a target file T with respect to a source file S. That is, both the encoder and decoder have available identical copies of S. A new file T is encoded and subsequently decoded by making use of S. Traditional differencing algorithms compress data by finding common strings between two versions of a file and replacing these substrings by a copy reference. The resulting file is often called a *delta* file.

There are many practical applications where new information is received or generated that is highly similar to information already present. For example, when a software revision is released to licensed users, the distributor can require that a user must perform the upgrade on the licensed copy of the existing version, and can then reduce the size of the upgrade file that must be sent to the user by employing differential file compression. In the case of several releases of upgrades of the same software, each revision is given as a delta file with respect to the previous release. Consider the case when a user is interested in a release which its source file is not present on his local computer. The traditional approach would be to start with the version he does have, working his way through all intermediate releases performing delta decoding to each version, until he receives the desired one. This paper suggests skipping delta encoding of upgrade releases when source files are missing, instead of obtaining unnecessary intermediate files. This will be done by generating a single delta file of the required version release with respect to the source file present on the user's computer. Since no assumptions are made about the resources available to the user on the distributor's website, and in order to reduce processing time, the construction of the new delta file is done in time proportional to the size of the input files,[1] i.e., without any decompression.

[1] In practical applications, the size of the compressed file is generally proportional to the size of its original version. However, the definition applies also to the case of a sublinear compressed size as in [2].

Working on the compressed given form, the user can save memory space on the distributor's computer. Moreover, transmitting the created delta file instead of the series of delta files, can reduce the number of I/O operations. Processing time is also saved by decompressing only the constructed delta file instead of several files in the traditional approach.

Another application of differential file compression is file system backups, including those done over a network. 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. Using the new framework, recovering the last file system based on several differential backups can be done by generating a new delta file of the last backup with respect to the current situation of the system, therefore avoiding delta decoding of unnecessary intermediate files which is done in the traditional way. In addition to the increased speed of decoding, restoring the file system requires only space for the old and new versions using the proposed technique, unlike a spare intermediate file used in the traditional method.

Generating a delta file of two given files S and T can be done in two typical ways, LCS (Longest Common Substring) based algorithms (e.g. [7]), and edit-distance based algorithms (e.g. [1, 19, 20]). Hunt, Vo, and Tichy [8] 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. In this research we construct a delta file based on edit distance and LZ techniques for differencing.

Factor, Sheinwald, and Yassour [4] 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. They achieve better compression by reducing the set of all shared files to the only relevant subset. 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.

Burns and Long [3] 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 [18] also 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 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.

Processing the compressed form of the file instead of working on the decompressed file in order to reduce processing time and memory storage was also carried out in the area of compressed pattern matching, i.e. performing pattern matching on the compressed form of the file, unlike the regular way of performing pattern matching on the decompressed file (Amir, Benson and Farach [2]). Compressed pattern matching was also studied in [5, 6, 9, 10, 11, 12, 13, 16, 17].

Working on the compressed delta file without the use of the source file, was also done in the framework of Compressed Delta Encoding, which generates the delta files of two given files

S and T while processing their compressed form. Klein, Serebro and Shapira [14] and Klein and Shapira [15] explore the compressed differencing problem on LZW and LZSS compressed files, respectively, and devise a model for constructing delta encodings on compressed files. Experiments showed that the constructed delta file is much smaller than the corresponding input LZW and LZSS compressed files.

Let $\Delta(X, Y)$ denote the delta file of the target file Y with respect to the source file X. In this paper we explore the Compressed Transitive Delta Encoding (CTDE) paradigm, which is to construct a delta file $\Delta(S, R)$ working directly on the two given delta files, $\Delta(S, T)$ and $\Delta(T, R)$, in time proportional to the size of the input. Section 2 defines the problem formally, and introduces two other variations of the problem. Section 3 presents an algorithm for solving the CTDE problem, and finally Section 4 presents experimental results.

2. Formal Discussion of the Problem

Given a source file S and a target file T, the delta file $\Delta(S, T)$ can be formed using a format of individual characters and copy items, where copies refer either to S or to T itself. The copies are initially described in the form of ordered pairs. An ordered pair (*off*, *len*) in T 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, and the length of this substring (number of characters) is *len*. An ordered pair (*pos*, *len*) in S refers to a copy of a substring starting at position *pos*, and again the second component, *len*, is the number of its characters. To distinguish between copies from S and T we use an additional flag bit so that each copy item is in fact a triplet. For example, $(0, pos, len)$ could refer to copies of substrings in S, while $(1, off, len)$ would indicate that the copy is from T.

Let S, T, R be 3 files. Given $\Delta(S, T)$ and $\Delta(T, R)$ the goal is to be able to construct R. The traditional way is to decompress $\Delta(S, T)$ to construct T and then apply $\Delta(T, R)$ on T and obtain R. The *Compressed Transitive Delta Encoding (CTDE)* paradigm is to construct a delta file $\Delta(S, R)$ in time proportional to the size of the input, i.e., working directly on the two given delta files, $\Delta(S, T)$ and $\Delta(T, R)$, without any decompression or the use of the source file S.

The CTDE problem can be generalized to the case of n files, $T_1, T_2, \cdots, T_n$ (possibly successive versions of the same software or n sequential system backups). Given $n-1$ delta files $\Delta(T_i, T_{i+1})$ $(1 \leq i \leq n-1)$, the CTDE paradigm will avoid constructing and storing the intermediate versions $T_2, T_3, \cdots, T_{n-1}$ in order to obtain T_n. It will generate $\Delta(T_1, T_n)$ working directly on the delta files $\Delta(T_i, T_{i+1})$, with no additional storage. Solutions to the basic version of CTDE can serve as a solution to the generalized version by applying it $n-2$ times. In each stage $\Delta(T_1, T_{i+2})$ is constructed and used in the successive stage, until we end with the delta file of T_n with respect to T_1 as desired. The algorithm denoted by $GeneralCTDE()$ is given in Figure 1 and uses Algorithm $CTDE()$ which is presented in the following section as a solution to the basic CTDE problem.

To complete the model of Compressed Transitive Delta Encoding we also introduce the following problem that can be seen as the reverse operation of the basic version. It is called the *Reverse Compressed Transitive Delta Encoding (RCTDE)* Problem. Let S, T, and R be 3 files. Given $\Delta(S, T)$ and $\Delta(S, R)$, the goal is to construct $\Delta(T, R)$ without decompressing the delta files. This problem is intended for the case where all software revisions are given with respect to a source file S. In this case we have already downloaded T in the past and erased the source file S assuming by mistake it would not be needed any more. In order to upgrade T to R one can construct the delta file of R with respect to T remotely by processing the

```
GeneralCTDE(\{\Delta(T_i, T_i + 1)\}_{i=1}^{n-1}){
    for i ← 1 to n − 2
        Δ(T_1, T_{i+2}) ← CTDE(Δ(T_1, T_i + 1), Δ(T_{i+1}, T_{i+2}))
}
```

FIGURE 1: *Using $CTDE()$ Algorithm for solving the basic CTDE problem for a solution to the generalized CTDE problem.*

delta files $\Delta(S,T)$ and $\Delta(S,R)$, and transfer only the output delta file. In this case generating R cannot be done in any other way unless restoring the source file S, since S is no longer available. In this paper we investigate the compressed transitive delta encoding paradigm and leave the reversed version for future research.

3. Solving CTDE

This section presents an algorithm for solving the CTDE problem working directly on the delta files. It first uses a utility method named $BaseDelta(S,T)$ for transferring a given delta file, $\Delta(S,T)$, to a file with non self pointers, i.e., with no pointers from T pointing backwards to previous occurrences of the string starting at the current location in the already processed portion of T.

3.1 The BaseDelta Algorithm

Given a differencing file $\Delta(S,T)$, the $BaseDelta(S,T)$ utility method describes a reconstruction of the given delta file so that it includes only pointers to the base file S. It is done by removing all self pointers working on $\Delta(S,T)$, without the use of S.

FIGURE 2: *Schematic representation of pointers in $\Delta(S,T)$. Pointers of $\Delta(S,T)$ pointing into S are colored gray together with their sources. Pointers into the previous scanned text (in T) are uncolored rectangles with dotted sources.*

Figure 2 shows the difficulty of the problem. A pointer p can refer to a previous occurrence of the substring starting at the current location. The corresponding location of a previous occurrence of the string represented by p in $\Delta(S,T)$ can be located by summing up the length components of pointers and ones for raw characters that precede it. Once the location is found, it can again contain pointers to the already scanned part of T, which recursively should be replaced. One can face this problem by working from left to right, and replacing a pointer to its previous scanned text into the corresponding characters and/or pointers into S. Working left to right guarantees that farther replacements, for those that point to locations to their left, would not contain T pointers.

For example, let p be the pointer (1,300,50), which represents the encoding of the string starting at position 1000 in T, and refers to positions 700 to 749 in T. Let us assume that the 50 characters in T starting at position 700 are contained in the string obtained by decoding the sequence (0,400,30)`abc`(1,200,60) in $\Delta(S,T)$. In particular, it refers to the 20 higher characters of (0,400,30) (positions 410 - 429 in S) the 3 raw characters `abc`, and the 27 lower characters of (1,200,60), (positions 500-526 in T). Note that the second component of a pointer in S is the absolute address of the previous occurrence of the string, while in T it is taken relative to the current position of the original string. The pointer p is replaced by a shorter pointer to S (e.g., (0,410,20)), the three raw characters `abc` and the adjusted encoding of the previous occurrence in T, which occurs to its left, and is composed without T pointers. The following figure presents the algorithm for transferring a general delta file into one which does not include self references. The concatenation is denoted below by $\cdot$. Pointers to S and to T are distinguished by their preceding flag bit.

```
BaseDelta(Δ(S,T)) {
    tmp ⟵ ϵ //initialize with empty string
    while Δ(S,T) not empty {
        input delta item x
        if x is an individual character
            tmp ⟵ tmp · x
        else if x is a pointer into S
            tmp ⟵ tmp · x
        else// x is a pointer into T
            E(x) ⟵ the encoding of x in Δ(S,T) possibly truncating first and last pointers
            tmp ⟵ tmp · E(x)
    }
    return tmp
}
```

FIGURE 3: *BaseDelta Algorithm returns a new delta file by removing self pointers from a given $\Delta(S,T)$ delta file.*

Once a pointer into T is encountered it is replaced by the corresponding characters and S pointers. This is in fact partial decoding of the file $\Delta(S,T)$, but can be justified as follows:

1. It is constructed by working on $\Delta(S,T)$ without the use of S.
2. It is only applied on pointers to T and not on pointers to S.
3. This kind of decoding is only applied on $\Delta(S,T)$ and not $\Delta(S,R)$ when solving the CTDE problem.
4. The only effect of this partial decoding on $\Delta(S,R)$ is when pointers from R to T in $\Delta(T,R)$ correspond to pointers from T to itself in $\Delta(S,T)$.

3.2 CTDE Algorithm

Given $\Delta(S,T)$ and $\Delta(T,R)$, the objective of the Compressed Transitive Delta Encoding (CTDE) Problem is to construct $\Delta(S,R)$ without decompressing the delta files and without the use of S. The algorithm $CTDE()$ for solving the CTDE problem scans the file $\Delta(T,R)$

which is used to construct R based on T, and replaces all pointers pointing to positions in T by raw characters or pointers into S. If a pointer in $\Delta(T,R)$ points to a previous occurrence of the string starting at the current position in the file R, it is transferred directly into $\Delta(S,R)$, since also at the corresponding position of $\Delta(S,R)$ it pointes to the same previous occurrence of the same substring of R. If a pointer p in $\Delta(T,R)$ points to an occurrence of the current substring in its source file T, p must be replaced with a non reference to T. The file $\Delta(S,T)$ is then scanned to locate the encoded portion, $\mathcal{E}(p)$, of pointers which correspond to the substring represented by p. Since self pointers in $\Delta(S,T)$ were removed by applying $BaseDelta(\Delta(S,T))$ Algorithm, the pointers in $\mathcal{E}(p)$ are only S pointers, and can be transferred to $\Delta(S,R)$. The first and last pointers are adjusted so that their decoding start and end on the first/last character of the string represented by p. Other intermediate pointers of $\mathcal{E}(p)$ are transferred to $\Delta(S,R)$ with no changes. The algorithm for solving the basic version of CTDE is given in Figure 4. While working through $\Delta(T,R)$ the delta file $\Delta(S,R)$ is constructed by outputting items from either $\Delta(T,R)$ or $\Delta(S,T)$. Again $\cdot$ is used for concatenation, and pointers from the source or target file are distinguished be a flag bit.

```
CTDE(Δ(S,T), Δ(T,R)) { // construct Δ(S,R)
    Δ(S,T) ⟵ BaseDelta(Δ(S,T))
    tmp ⟵ ϵ //initialize with empty string
    while Δ(T,R) not empty {
        input delta item x
        if x is an individual character
            tmp ⟵ tmp · x
        else if x is a pointer into R
            tmp ⟵ tmp · x
        else// x is a pointer into T
            E(x) ⟵ the encoding of x in Δ(S,T) possibly truncating first and last pointers
            tmp ⟵ tmp · E(x)
    }
    return tmp
}
```

FIGURE 4: *CTDE Algorithm for constructing $\Delta(S,R)$ out of $\Delta(S,T)$ and $\Delta(T,R)$ without decompressing the delta files.*

The following example is given to demonstrate the above CTDE algorithm. Let `S=abcdxdce`, `T=abcddceabc` and `R=ceabcdxyzaxyz`. A greedy delta encoding algorithm produces $\Delta(S,T) = (0,1,4)(0,6,3)(0,1,3)$ and $\Delta(T,R) = (0,6,5)\texttt{dxyza}(1,4,3)$, where 0 indicates a copy item copied from the source file, and 1 indicates a copy item of a string lying in the previous scanned text (a self pointer). The indices are numbered starting from 1. When producing $\Delta(S,R)$ using CTDE, the stream $\Delta(T,R)$ is scanned. The pointer (0,6,5) refers to the string `ceabc` which is a substring of the string represented by the encoding of $(0,6,3)(0,1,3)$ in $\Delta(S,T)$. This is computed by processing the length components of $\Delta(S,T)$. The first pointer is truncated to form the encoding $(0,7,2)(0,1,3)$, which is output to $\Delta(S,R)$. The following raw characters `dxyza` in $\Delta(T,R)$ are concatenated to $\Delta(S,R)$ together with the self pointer $(1,4,3)$. That is, $\Delta(S,R) = (0,7,2)(0,1,3)\texttt{dxyza}(1,4,3)$ using CTDE Algorithm, as opposed to $(0,7,2)(0,1,5)\texttt{yza}(1,4,3)$ delta file which is constructed directly from S and R using a greedy algorithm. In this example the compression performance of the delta file generated

directly from S and R is slightly better than the CTDE compression results. However, delta files of successive software releases or successive system backups tend to look more similar to their previous version rather than the one that comes before this previous version, as in this artificial example.

4. Experimental Results

The first goal was to program a delta compression software for a given source file S and target file T in order to simulate delta compression utilities. Five different formats were used for encoding items in the delta file, which were distinguished by control integers between 0 and 4. The control integer 0 was used to indicate a raw character. Other control integers were used to represent pointers in T with offsets and length as follows, while pointers in S used only the last format.

control integer	offset	length
1	$< 4K$	< 259
2	$4K \leq .. < 32K$	< 259
3	$\geq 32K$	< 259
4	$\geq 32K$	≥ 259

TABLE 1: *Pointer format*

An off-the-shelf arithmetic encoder was used to encode the control integers, the raw characters and the length fields for cases 1 through 3. We used a fixed length code for the offset fields (12, 15 and log(maximum size of the file) for cases 1, 2, and 3-4, respectively). This format together with the arithmetic encoder was constructed to approximate the UNIX gzip utility that uses a window of size 32K and maximum match length of size 256 characters.

For our experiments we consider three typical series of software releases. First, 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 3.0.3, 3.0.4, 3.1, 3.1.1, 3.2 and 3.2.1, executables. Second, the GNU SQL server which is a portable multiuser relational database management system, versions 0.7b3, 0.7b4, 0.7b5, 0.7b6,, 0.7b6.1, source code. And, finally, the GNU xfig utilities to draw and manipulate objects, versions 3.2.1, 3.2.2 and 3.2.3, source code. Table 2 presents the set of software releases that were used in our experiments and their corresponding sizes in bytes.

gcc30	gcc303	gcc304	gcc31	gcc311	gcc32	gcc321
274,981	277,055	277,335	299,671	299,687	264,805	302,897

sql3	sql4	sql5	sql60	sql61	xfig321	xfig322	xfig323
2,722,338	2,920,666	5,861,140	3,371,146	2,981,200	831,748	831,748	926,785

TABLE 2: *Sizes in bytes of original files*

To evaluate the compression performance of our delta simulation, its compression results were compared to those of *xdelta* and *zdelta* UNIX utilities. As can be seen from Table 3 the compression performance of our delta algorithm, denoted by *SimDelta*, approximates the compression results of *xdelta* and *zdleta*. As this simulation was only constructed as a tool to produce the input delta files for our main algorithm, these reasonable compression results

are satisfying. The CTDE software was constructed on the basis of this simulation, using the same format of pointers. One can expect that the compression we achieve here will only be improved by using a gzip implementation.

	30-303	303-304	304 - 31	31-311	311-32	32-321
Xdelta	48,028	30,755	97,256	44,791	70,682	95,379
Zdelta	33,716	16,174	104,888	22,182	67,146	106,451
SimDelta	58,156	30,958	99,882	41,034	85,487	96,571

	321-322	322-323
Xdelta	903	300,904
Zdelta	911	255,432
SimDelta	499	337,216

	sql3-4	sql4-5	sql5-60	sql60-61
Xdelta	118,835	27,375	163,024	14,084
Zdelta	101,890	687,468	262,770	79,557
SimDelta	108,883	52,220	131,726	8,713

TABLE 3: *Compression performance of the delta file as opposed to xdelta and zdelta*

The following experiment was applied in order to show the loss in compression performance when removing self pointers in a given delta file using algorithm $BaseDelta()$ which was presented in Figure 3. Figure 5 represents the sizes of the delta files before and after $BaseDelta$ was applied. Although compression loss is significant in some cases, the following experiment shows that its effect on the overall compression results has less influence. Note the exception of the sql 4-5 case, where the restriction of using pointers only from sql4 improves the compression performance. This is explained by the overhead of using 4 pointer formats instead of 1.

FIGURE 5: *Compression performance with and without self pointers using DeltaBase Algorithm.*

For our main experiment which evaluates the Compressed Transitive Delta Encoding Algorithm (CTDE), presented in Figure 4, we used three files, say, S, T and R in each experiment. CTDE was applied on $\Delta(S,T)$ and $\Delta(T,R)$ and generated $\Delta(S,R)$. The size of this output file was compared against the sum of the sizes of $\Delta(S,T)$ and $\Delta(T,R)$, which must be used in the traditional way (for example, when transferring it from a remote computer), and to the size of generating $\Delta(S,R)$ directly on the decompressed files S and R. The first column lists the 3 input files, columns 2,3 and 4 show the sizes in bytes of the output files of CTDE, sum of files, and direct delta compression, in correspondence. As can be seen, there is a correlation between the compression loss in $\Delta(S,T)$ of not using self pointers, and the

compression performance of CTDE on $\Delta(S,T)$ and $\Delta(T,R)$. One can first apply *BaseDelta* on $\Delta(S,T)$ and use it as an indication of whether to apply CTDE or not.

Input files	CTDE	Traditional	Direct
gcc 30-303-304	74,942	89,114	61,593
gcc 303-304-31	101,214	130,840	99,899
gcc 304-31-311	126,645	142,047	99,782
gcc 31-311-32	86,166	115,473	85,503
gcc 311-32-321	103,396	166,061	69,901
sql 3-4-5	209,083	161,103	156,703
sql 4-5-60	153,208	183,946	143,644
sql 5-60-61	164,239	140,439	98,547
xfig 321-322-323	337,596	337,715	337,461

TABLE 4: *Compression performance of CTDE as opposed to the traditional incremental delta encoding and the construction of a direct delta file*

The compression performance achieved by applying delta encoding on the decompressed forms of the files is better than CTDE and the traditional method, as expected. However, in the case where these decompressed files are unavailable on the user's computer, as in the case of software distribution on remote computers or system backups over a network, one should transfer both encoded forms and decompress them in order to create this delta file. When comparing the number of bytes transformed over the network, CTDE outperforms the traditional way in most cases. Moreover, CTDE uses only a constant amount of additional space, as opposed to both traditional and direct methods which use linear additional memory storage.

5. Conclusion

In this paper we have defined the problem of merging consecutive delta files in order to obtain the final version, without producing the intermediate ones. Experiments with the presented algorithm show that not only do we save storage, but the single delta file constructed is smaller than the original delta files, thus saving I/O operations in case these files are stored remotely.

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] 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, 299–307.

[3] 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.

[4] 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.

[5] 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, 703–712.

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

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

[8] 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.

[9] 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, 130–158.

[10] Klein, S.T. and Shapira, D., *A New Compression Method for Compressed Matching.* Data Compression Conference, DCC-2000, Snowbird, Utah, 2000, 400–409.

[11] Klein, S.T. and Shapira, D., *Pattern Matching in Huffman Encoded Texts.* Information Processing and Management, IP&M *Volume 41(4)*, 2005, 829–841.

[12] Klein, S.T. and Shapira, D., *Searching in Compressed Dictionaries.* Proc. Data Compression Conference, DCC-2002, Snowbird, Utah, 2002, 142–151.

[13] Klein, S.T. and Shapira, D.: *Compressed Pattern Matching in JPEG Images.* International Journal of Foundations of Computer Science, 17(6), 2006, 1297–1306.

[14] Klein, S.T., Serebro, T.C. and Shapira, D., *Modeling Delta Encoding of Compressed Files.* Proc. Prague Stringology Club, PSC-2006, 2006, 162–170.

[15] Klein, S.T. and Shapira, D., *Compressed Delta Encoding for LZSS Encoded Files.* Proc. Data Compression Conference, DCC-2007, 2007, 113–122.

[16] 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, 1999, 14–36.

[17] Shapira, D. and Daptardar, A., *Adapting the Knuth-Morris-Pratt algorithm for pattern matching in Huffman encoded texts.* Information Processing and Management, IP&M, *Volume 42(2)*, 2006, 429–439.

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

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

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