An Approximation to the Greedy Algorithm for Differential Compression of Very Large Files

Ramesh C. Agarwal [*] Suchitra Amalapurapu [†] Shaili Jain [‡]

Abstract

We present a new differential compression algorithm that combines the hash value techniques and suffix array techniques of previous work. Differential compression refers to encoding a file (a version file) as a set of changes with respect to another file (a reference file). Previous differential compression algorithms can be shown empirically to run in linear-time but they have certain drawbacks, namely they do not find the best matches for every offset of the version file. Our algorithm finds the best matches for every offset of the version file, with respect to a certain granularity (or *block size*) and above a certain length threshold. It has two variations depending on how we choose the block size. If we keep the block size fixed, we show that the compression performance of our algorithm is similar to that of the greedy algorithm, without the expensive space and time requirements. If we vary the block size linearly with the reference file size, we show that our algorithm can run in linear-time and constant-space to compress very large files. Our algorithm combines the techniques of hashing sections of the files to obtain *footprints* and the use of suffix arrays to find the longest match. We also show empirically that our algorithm performs better than *xdelta* [7], *vcdiff* [3], and the work of Ajtai et al. [1] in most cases in terms of compression and performs better than *vcdiff* and the work of Ajtai et al. in terms of speed.

1 Introduction

Differential compression or delta compression is a way of compressing data that have a great deal of similarities. Differential compression produces a delta encoding, a method of representing a version file in terms of an original file plus new information. Thus differential compression algorithms try to efficiently find common data between a reference and a version file to reduce the amount of new information which must

[*]IBM Almaden Research Center, 650 Harry Road, San Jose, CA, 95120. Email: ragarwal@us.ibm.com

[†]This work was done while the author was visiting IBM Almaden Research Center. Email: asmsuchi@yahoo.com

[‡]Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI, 48109. This work was done while the author was visiting IBM Almaden Research Center. Email: shailij@umich.edu

be used. By storing the reference file and this delta encoding, the version file can be reconstructed when needed.

The main uses of differential compression algorithms are in software distribution systems, web transportation infrastructure, and version control systems, including backups. In systems where there are multiple backup files, a great deal of storage space can be saved by representing each backup file as a set of changes of the original. Alternately, we can choose to represent the reference file in terms of a version file or represent each file as a set of changes of the subsequent version. Furthermore, transmission costs would be reduced since we only need to send the changes rather than the entire version file. In client/server backup and restore systems, network traffic is reduced between clients and the server by exchanging delta encodings rather than exchanging whole files [2]. Software updates can be performed this way with the added benefit of providing a level of piracy protection since the reference version is needed to reconstruct the version file. Thus software updates can be performed over the web, since the delta will be of very little use to those who do not have the original.

We will consider the problem of compressing a single version file along with a single reference file for simplicity purposes, but our results can be extended to situations involving a number of version files along with a single reference file. We compare our empirical results to those of *vcdiff* [3], *xdelta* [7], and the work of Ajtai et al. [1]. These three families of algorithms seem to be the most competitive differential compression algorithms currently available, in terms of both time and compression. Ajtai et al. describe a greedy algorithm based on the work of Reichenberger [9] that operates by computing hash values on the reference file at every possible offset and storing all such values. The greedy algorithm runs in quadratic-time and linear-space in the worst case. However in highly correlated data, the greedy algorithm shows almost linear behavior [1]. The *vcdiff* algorithm [3] combines the string matching technique [12] and block-move technique [11]. It uses the reference file as part of the dictionary to compress the version file. Both Lempel-Ziv algorithm and the block-move algorithm of Tichy run in linear time. The *xdelta* algorithm is a linear-time, linear-space algorithm and was designed to be as fast as possible, despite sub-optimal compression [6]. It is an approximation of the quadratic-time linear-space greedy algorithm [2]. Most recently, Shapira and Storer present an algorithm that works in-place and uses the sliding window method [10].

2 Proposed Algorithm

Our algorithm much like previous algorithms hashes on blocks of the reference and version file. The hash value is an abbreviated representation of a block or substring. It is possible for two different blocks to hash to the same value giving a spurious match. Thus if the hash values match, we must still check the blocks using exact string matching to ensure that we have a valid match. We use the Rabins-Karp hash method [4] because we believe it gives the least amount of collisions in most situations and it is efficient to compute for a sliding window. We hash modulo a Mersenne prime,

$2^{31} - 1$ because it allows us to compute hash values very efficiently. Our algorithm utilizes the suffix array introduced by Manber and Myers [8].

2.1 Important Data Structures

We will introduce three data structures that are crucial to the running time of our algorithm, but do not affect the complexity of our algorithm. The purpose of the *quick index array* is to serve as an early exit mechanism if a hash value is not present in the hash table. The reason we desire the *quick index array* is that there are 2^{31} possible hash values. The number of distinct hash values is always less than the number of blocks, or l, which is typically less than 2^{20}. This means our hash table is about 0.05% populated! The size of the *quick index array* can be described as a tradeoff between storage and the efficiency of our early exit scheme. The *quick index array* is an array of bits that contains 2^t bits or 2^{t-3} bytes and is initialized to contain all 0s. It is formed by a single pass on the suffix array. For each hash value in the suffix array, we extract the first t bits of the hash value to index the location in the *quick index array* and place a 1 in that bit location. Thus if a hash value is present in the suffix array, there will be a 1 present in the location indexed by the first t bits of the hash value. If a hash value is not present in the suffix array, there could be either a 1 or 0 in the location indexed by the first t bits of the hash value depending on whether there is another hash value with the same leading t bits. We decided on the number of bits to use by the following reasoning: There are at most l different hash values, thus at most l of the 2^t bits in the hash table are 1. Thus for a random hash value which is not present in the hash value array, the probability of finding a 1 in the quick index array is less than 2^{-5}. Since we make only one pass on the suffix array, our data structure takes $O(l)$ time to construct, where l is the number of hash values in the hash array.

The *block hash table* is an array where the reference file hash values are re-organized for efficient matching. The even locations contain the hash values in the same order that they appear in the suffix array. Each 31-bit hash value, $H(r)$, is stored in a 32-bit integer with the leading bit indicating if the hash value is unique. If the hash value $H(r)$ is unique, the odd locations contain its location in the hash value array. If the hash value $H(r)$ is not unique, the first odd location contains the *count* or the total number of occurrences of this hash value in the suffix array. The first pair after the hash value and *count* represents the lowest ranked substring that starts with $H(r)$ and the last pair represents the highest ranked substring that starts with $H(r)$. All of the substrings are ordered, as they are in the suffix array since the block hash table is created by a single pass of the suffix array.

The *pointer array* is an array of indices to the *block hash table*. It is indexed by the first s, or $\lceil \log_2 l \rceil - 2$, bits of the hash value. This location contains a pointer to the smallest reference file hash value with the same leading s bits. Thus the *pointer array* reduces the number of hash values we must process in order to find if a match exists. There are at most l different hash values in the hash table. So if we use s bits, each pointer in the *pointer array*, on average, will map to approximately four hash values in the *block hash table*, assuming that we have a good hash function.

These three data structures are used for efficient matching (see figure 1) between the version file hash values and the reference file hash values.

```
best-match-algorithm
input: hash value $H(v_c)$ that we are seeking to match

  1. if (the corresponding entry to $H(v_c)$ in the quick index array is 0)
      (a) return a null value since $H(v_c)$ is not in the hash table
  2. else
      (a) use the corresponding entry in the pointer array to get a hash value in the block hash table
      (b) if (the hash value in the block hash table $> H(v_c)$)
            i. return a null value since $H(v_c)$ is not in the table
      (c) else
            i. now sequentially process the values in the block hash table until we either find an entry equal to
               $H(v_c)$ or we find the first entry that is larger than $H(v_c)$
           ii. if the latter case is true
                 • return a null value because $H(v_c)$ is not in the table
          iii. else (we have a match)
                 • if the match is unique, indicated by the leading bit of the 32-bit hash value
                     – return the match
                 • else (the match is not unique)
                     – perform binary-search-for-suffixes as described in Figure ?? to find the longest match
                       where the initial range is count number of values
                     – return the longest match

output: the best match for the input hash value, $H(v_c)$, if it exists and a null value otherwise
```

Figure 1: The Best Match Algorithm

2.2 Algorithm Pseudocode

The main differential compression algorithm is given in Figure 2. The binary search method we use is similar to that presented in [8]. We maintain a $lowRankPtr$ and a $highRankPtr$ which are pointers to index the block hash table. Note that we can reduce the amount of hash value comparisons in our binary search method by keeping track of $min(|lcp(low,w)|, |lcp(w,high)|$, where low is the string in the hash value array corresponding to $lowRankPtr$ and $high$ is the string in the hash value array corresponding to $highRankPtr$. This binary search method exploits a property of suffix arrays that when searching for a particular string w in a suffix array, if the length of the longest common prefix is x blocks in length, all suffixes that match w to x blocks will be adjacent to each other in the array.

```
differential-compression-algorithm
input: reference file string, version file string

1.  (a) initialize $r_c = 0$.
    (b) while $(r_c < m - p)$
        i. compute $H(r_c)$ and store in the *hash value array*
        ii. increment $r_c$ by $p$

2. Call suffix-array-construction with *hash value array* as input.

3. In a single pas of the suffix array, create the quick index array, the pointer array, and the block hash table as
   described in section 2.1.

4. Initialize $v_c = 0$ and $v_{prev} = 0$.

5. while $(v_c < n - p)$
    (a) initialize best-seen-match to null
    (b) compute $H(v_c)$
    (c) call best-match-algorithm as described in Figure 1 with $H(v_c)$ as input
    (d) if matching algorithm returns null
        i. increment pointer $v_c$ by 1
    (e) else (we have a match)
        i. store match in the best-seen-match variable
        ii. $v_{temp} = v_c$
        iii. for $(v_c = v_{temp} + 1$ to $v_c = v_{temp} + p - 1)$
            • compute $H(v_c)$ and call the best-match-algorithm with $H(v_c)$ as input
            • if matching algorithm returns a match and is longer than the best-seen-match
                – update the best-seen-match variable to contain the current match
        iv. check the validity of the best-seen-match and extend it as far left and right as possible and finally
            encode the match as a copy command
        v. encode the information starting at $v_{prev}$ to the start of the match as an insert command
        vi. update the $v_c$ and $v_{prev}$ pointers to the end of the match

6. Encode the last bit of information from $v_{prev}$ to the end of the file as an insert command.

postcondition: delta encoding with copy and insert commands and new information to be encoded.
```

Figure 2: The Differential Compression Algorithm

2.3 Time and Space Analysis

2.3.1 Space Analysis

The main data structures that require space proportional to the number of blocks in the reference file are the *hash value array*, the *suffix array*, the *rank array* in the suffix array construction routine, the *quick index array*, the *pointer array*, and the *block hash table*. The block hash table is created in place using the space occupied by the suffix array and the rank array. The *hash value array* contains l hash values, each a four-byte value. The *suffix array* contains l indices in the odd locations and l hash values in the even locations, each a four-byte value. The *quick index array* contains $2^{\lceil \log_2 l \rceil + 5}$ bits or $2^{\lceil \log_2 l \rceil + 2}$ bytes and the *pointer array* contains $2^{\lceil \log_2 l \rceil - 2}$ entries or $2^{\lceil \log_2 l \rceil}$ bytes. The *block hash table* contains $2 \cdot l$ entries if all the hash values are

Data Structure	Space (bytes)
hash value array	$4 \cdot l$
suffix array	$8 \cdot l$
rank array	$4 \cdot l$
quick index array	$4 \cdot l$ to $8 \cdot l$
pointer array	l to $2 \cdot l$
block hash table	$8 \cdot l$ to $12 \cdot l$
total	$21 \cdot l$ to $26 \cdot l$

Step	Complexity
hashing reference file	$O(m)$
suffix array construction	$O(l \log l)$
data structure construction	$O(l)$
hashing version file	$O(n)$
processing version file	$O(n \log l)$
total	$O(m + l \log l + n \log l)$

Table 1: The space and time analysis of our algorithm.

unique. When the hash values are not unique, the block hash table contains at most $3 \cdot l$ values. Our algorithm is structured such that the block hash table shares memory with the suffix array and the rank array. Thus, the worst case memory requirement is only $26 \cdot l$ bytes (see Table 1).

2.3.2 Time Analysis

Rather than using the quick index array, the pointer array, and the block hash table in our time analysis, we shall just consider matching to be done as binary search on the entire suffix array. This is reasonable since the quick index array, pointer array, and block hash table are simply implemented to make the matching process more efficient. Hence we know our algorithm performs at least as well as the simplified version of our algorithm. Another assumption that we make is that there are no spurious multi-block matches, that is instances where two or more consecutive hash values match, but the block themselves do not match. This is reasonable since the probability of having a spurious multi-block match is very low.

The single pass of the reference file and the hash computation in Step 1 of Figure 2 requires $O(m)$ time, since the hash computation requires use of every byte in the reference file. The creation of the suffix array requires $O(l \log l)$ time. The single pass of the suffix array to create the quick index array, the pointer array, and the block hash table require $O(l)$ time. Now let us consider Step 5 of Figure 2, processing of the version file. In Step 5(b), we could compute hash values at every offset of the version file. Computing hash values on every offset of the version file has complexity $O(n)$. This argument relies on a property of the Karp-Rabin hash computation [4] which requires $O(1)$ computation on $H(x)$ to produce $H(x+1)$. The binary search routine used in the processing of the version file, has complexity $O(d + \log l)$, where d is the depth of the longest match in terms of blocks, as shown in [8]. It is easy to show that the worst-case complexity of processing the version file amortized to each byte, regardless of if there is a match or not, is $O(\log l)$. Hence processing the entire version file takes $O(n \log l)$. The time requirements of our algorithm are summarized in Table 1.

The total complexity of our algorithm is $O(m + l \log l + n \log l)$. This can also be written as $O(m + \frac{m}{p} \log \frac{m}{p} + n \log \frac{m}{p})$. If p remains constant as m increases, the complexity can be simplified to $O((m+n) \log m)$. In the case where p increases proportionally with m, $\frac{m}{p}$ or l is a constant. Thus the complexity reduces to $O(m+n)$. In other words, if we increase p proportionally with m, that is fix l, we have a linear-time constant-space algorithm. However, in practice, we get much better running time than the asymptotic complexity derived above because of the creation of the quick index array, the pointer array, and the block hash table, which narrow the scope of our binary search and the matching procedure.

2.3.3 Performance Compared to the Greedy Algorithm

The compression performance of our algorithm is equivalent to the performance of greedy algorithm, but it is more efficient, for the following reasons. In the greedy algorithm, hash values are computed on the reference string for all offsets. Thus it can find matching substrings which are at least one block long. Our algorithm computes hash values only on the block boundaries of the reference file. Thus, our algorithm will miss matches which are longer than a block but do not contain a full reference block, or in other words, if the common substring straddles two blocks. This is because there is no single reference file hash value that represents the information in this match. However, we are sure to find any match which is at least two blocks long, because, in this case, the match contains at least one complete block of the reference file, for which we have computed the hash value. Thus, we can get results equivalent to the greedy algorithm if we choose our block size to be half of the block size used by the greedy algorithm. The greedy algorithm as described by Ajtai et al. requires storing $n - p + 1$ hash values (approximately n hash values) where n is the length of the reference string. Our algorithm will require using only $\frac{2 \cdot n}{p}$ hash values where p is the block size used by the greedy algorithm. In this step, our algorithm reduced the space requirement by approximately $\frac{p}{2}$ without giving up any capability in finding small matches.

3 Experimental Results

We compare our algorithm with *xdelta*, *vcdiff*, the halfpass algorithm of Ajtai et al., and the greedy algorithm implemented by Ajtai et al. on a data set of files ranging in size from 78KB to 55MB. Due to space limitations, we only present results gathered from the "in-memory" version of the algorithm.

Our algorithm uses *bzip2* on token streams to provide additional compression. For standardization purposes, we implemented *bzip2* on the outputs of *xdelta*, *vcdiff*, the halfpass algorithm of Ajtai et al., and the greedy algorithm. We have used a block size of 12 when the reference file is smaller than 1MB and a block size of 24 when the reference file is larger than 1MB. All experiments were run on an IBM Intellistation Model M Machine running the Red Hat Linux Operating System, with a Pentium III 997 MHz Processor and 512 MB of RAM. The timing results were averaged over 10 simulations of each compression algorithm after the cache had been warmed up. We ensured that the compression program was the only user executed program on the

CPU, trying to homogenize the environment for running the simulations.

For our experiments, we used two types of data. The first type of data is synthetically created and the second type is various versioned software. The synthetic data was created by taking an existing binary file and then performing a series of operations (insert, delete, copy, move) on it to obtain a version file. Typically, for an original file, we would use part of it as the reference file and then the rest of it as the source for the insert operations. Rows 1-9 of Table 2 contain this synthetically created test data. For the versioned software, we used different versions of *gcc*, *gnusql server*, *linux* files, *netscape* files, and *samba* files.

Some of the rows in Table 2 have the label, NO INSERTS. For these pairs of files, we used only delete, move, and copy operations on the reference file to create the version file. This NO INSERTS scenario is the most demanding scenario for any differential compression algorithm because matching substrings could be anywhere in the file. Some of the rows in Table 2 are marked ID ONLY. This stands for insert/delete only. For these test cases, we created the version file by performing a series of insert and delete operations on the reference file. The main characteristics of these test cases is that they preserve the relative order of information between reference file and the version file. Hence, there are no transpositions and substrings from the reference file are not copied to another location in the version file. This is a particularly easy scenario for algorithms that use a windowing techniques - they keep a section of the two files in memory. The *vcdiff* algorithm and the work of Ajtai et al. fall in this category.

In many of these cases, we used The Bible as the starting point. The entire text of The Bible is slightly larger than 4 MB and it contains several long repeated segments. The longest repeated phrase is about 500 bytes. We truncated The Bible text to create three reference files labelled Bible-small, Bible-medium, and Bible-large. We used the remaining part of The Bible as source for the inserts in generating the corresponding version files. Row 9 consists of files by the name of foo5 and foo6 that are an unusual and somewhat challenging data set for differential compression algorithms given to us by Fred Douglis of IBM Watson Research Center. These data sets contain repeating patterns of different lengths.

4 Conclusions

We have presented a differential compression algorithm that finds the optimal matches between a reference file and a version file at the granularity of a certain block size. Our compression algorithm operates in linear-time and constant-space when we increase the block size proportional to the reference file size. Furthermore, our algorithm provides better compression than *xdelta*, *vcdiff*, and the code of Ajtai et al. in most cases. Overall, our algorithm is somewhat slower compared to *xdelta* but significantly faster compared to *vcdiff* and the code of Ajtai et al. We have also shown that our algorithm is competitive with the greedy algorithm implemented by Ajtai et al. both empirically and theoretically, without the expensive space and time requirements. In

File set	file sizes	Our work	Ajtai et al.	vcdiff	xdelta	greedy algorithm
The Bible small	80,947	904	950	992	1,709	1,041
NO INSERTS	78,436	(0.04s)	(0.92s)	(0.02s)	(0.01s)	(0.05s)
The Bible medium	485,687	1,036	1,102	4,629	2,380	1,164
NO INSERTS	470,671	(0.10s)	(1.12s)	(0.37s)	(0.03s)	(0.23s)
The Bible large	3,642,652	2,395	2,707	545,029	5,420	2,679
NO INSERTS	3,496,353	(0.47s)	(2.72s)	(5.66s)	(0.24s)	(2.69s)
Web Data	32,949,807	2,830	3,852	9,211,077	5,430	3,340
NO INSERTS	31,694,955	(4.06s)	(17.80s)	(43.52s)	(2.19s)	(948.62s)
The Bible small	80,947	2,024	2,417	2,080	2,667	2,422
ONLY ID	78,969	(0.04s)	(0.92s)	(0.02s)	(0.01s)	(0.06s)
The Bible medium	485,687	4,938	5,366	5,612	5,596	5,025
ONLY ID	473,455	(0.12s)	(1.14s)	(0.37s)	(0.04s)	(0.27s)
The Bible large	3,642,652	25,183	30,873	206,508	32,403	27,781
ONLY ID	3,552,886	(0.56s)	(2.85s)	(3.63s)	(0.31s)	(3.04s)
Web Data	32,949,807	142,863	167,122	8,248,737	184,061	152,359
ONLY ID	32,176,333	(4.74s)	(18.36s)	(41.71s)	(2.61s)	(950.80s)
foo5	3,010,560	51,397	52,848	108,725	69,599	
foo6	1,441,393	(1.72s)	(2.24s)	(1.70s)	(0.73s)	
linux-2.0.9.cat	22,578,079	831	999	1,539	1,194	908
linux-2.0.10.cat	22,577,611	(2.88s)	(11.22s)	(9.14s)	(1.44s)	(66430.20s)
cp32e406.exe	17,738,110	7,116,994	7,127,597	7,105,957	7,077,299	7,144,958
cp32e407.exe	17,742,163	(6.47s)	(24.46s)	(14.43s)	(6.35s)	(60767.64s)
samba-1.9.18p10.cat	5,849,994	708,873	765,901	2,037,304	1,032,931	748,443
samba-2.0.0beta1.cat	9,018,170	(6.32s)	(7.31s)	(7.82s)	(2.83s)	(54414.57s)
gcc-2.95.1.tar	55,746,560	58,572	69,494	71,707	116,449	57,452
gcc-2.95.2.tar	55,797,760	(8.02s)	(25.79s)	(19.51s)	(3.85s)	(7+ hours)
gcc-2.95.2.tar	55,797,760	141,227	173,468	13,407,749	280,957	
gcc-2.95.3.tar	55,787,520	(8.59s)	(26.21s)	(57.06s)	(4.25s)	
gcc-3.0.4	277,335	97,645	102,135	98,764	96,792	102,550
gcc-3.1	299,671	(0.31s)	(1.31s)	(0.52s)	(0.36s)	(0.94s)
gcc-3.1	299,671	46,391	44,428	25,647	44,636	49,457
gcc-3.1.1	299,687	(0.23s)	(1.17s)	(0.19s)	(0.19s)	(0.55s)
gcc-3.1.1	299,687	70,972	73,009	78,086	70,435	73,432
gcc-3.2	264,805	(0.24s)	(1.23s)	(0.38s)	(0.41s)	(0.73s)
gcc-3.2	264,805	94,300	99,082	95,121	95,056	99,712
gcc-3.2.1	302,897	(0.29s)	(1.29s)	(0.57s)	(0.33s)	(0.90s)
sql-0.7b3	2,722,338	92,533	96,325	200,855	118,375	93,325
sql-0.7b4	2,920,666	(1.05s)	(2.43s)	(1.97s)	(0.53s)	(4.78s)
sql-0.7b4	2,920,666	27,137	25,197	727,399	27,346	24,038
sql-0.7b5	2,920,666	(0.73s)	(2.36s)	(4.24s)	(0.35s)	(4.45s)
sql-0.7b5	5,861,140	109,240	123,645	211,872	160,918	121,151
sql-0.7b6.0	3,371,146	(1.88s)	(4.26s)	(2.24s)	(0.96s)	(19.28s)
sql-0.7b6.0	3,371,146	8,498	9,037	88,273	14,011	8,543
sql-0.7b6.1	2,981,200	(0.72s)	(2.35s)	(1.57s)	(0.24s)	(4.88s)

Table 2: Comparison of compression performance for a set of files. The table gives the size of the delta file in bytes and compression time in seconds, shown in parenthesis.

most cases, our code provides slightly better compression compared to the greedy algorithm. We expect our method of differential compression not only to be used in compression applications but as a general string matching technique to be used in web crawling as well as computational biology.

References

[1] M. Ajtai, R. Burns, R. Fagin, D. D. E. Long, and L. Stockmeyer. Compactly encoding unstructured input with differential compression. *Journal of the ACM* 49(3), 318-367, 2002.

[2] R. C. Burns and D. D. E. Long. Efficient distributed backup and restore with delta compression. *Proceedings of the Fifth Workshop on I/O in Parallel and Distributed Systems*, 1997.

[3] J. J. Hunt, K.-P. Vo, and W. F. Tichy. Delta algorithms: An empirical analysis. *ACM Transactions on Software Engineering and Methodology*, 7(2), 1998.

[4] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development.* 1987.

[5] D. G. Korn and K.-P. Vo. The VCDIFF generic differencing and compression format. Technical Report Internet-Draft draft-vo-vcdiff-00.

[6] J. P. MacDonald. File System Support for Delta Compression. Master's Thesis at the University of California Berkeley. Available at http://www.xcf.berkeley.edu/ jmacd/.

[7] J. P. MacDonald. Versioned file archiving, compression, and distribution. UC Berkeley. Available via http://www.cs.berkeley.edu/ jmacd/

[8] U. Manber and E. W. Myers: Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal of Computing.* 22(5): 935-948 (1993)

[9] C. Reichenberger. Delta storage for arbitrary non-text files. *Proceedings of the 3rd International Workshop on Software Configuration Management*, 1991.

[10] D. Shapira and J. Storer. In-Place Differential File Compression. *Proceedings of the Data Compression Conference*, 2003.

[11] W. F. Tichy. The string-to-string correction problem with block move. *ACM Transactions on Computer Systems*, 2(4), 1984.

[12] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory.* 1977.