In Place Differential File Compression

Dana Shapira and James A. Storer

Dept. of Computer Science, Brandeis University
Waltham, MA 02254, USA

We present algorithms for *in-place differential file compression*, where a target file T of size n is compressed with respect to a source file S of size m using no additional space, in addition to the space used to replace S by T; that is, it is possible to encode using $m + n + O(1)$ space and decode using $\text{MAX}(m,n) + O(1)$ space (so that when decoding the source file is overwritten by the decompressed target file). From a theoretical point of view, an optimal solution (best possible compression) to this problem is known to be NP-hard, and in previous work we have presented a factor of 4 approximation (not in-place) algorithm based on a sliding window approach (Shapira and Storer[2004b]). Here we consider practical in-place algorithms based on sliding window compression where our focus is on decoding; that is, although in place encoding is possible, we will allow $O(m+n)$ space for the encoder so as to improve its speed, and present very fast decoding with only $\text{MAX}(m,n) + O(1)$ space. Although NP-hardness implies that these algorithms cannot always be optimal, the asymptotic optimality of sliding window methods along with their ability for constant-factor approximation is evidence that they should work well for this problem in practice. We introduce the *IPSW* algorithm (*In-Place Sliding Window*), and present experiments that indicate that it compares favorably with traditional practical approaches, even those that do not decode in-place, while at the same time having low encoding complexity and extremely low decoding complexity. IPSW is most effective when S and T are reasonably well aligned (most large common substrings occur in approximately the same order). We also present a preprocessing step for string alignment that can be employed when the encoder determines significant gains will be achieved.

1. Introduction

Differential file compression represents a *target file T* with respect to a *source file S*. That is, we assume that both the encoder and decoder have available identical copies of S, and the encoder is free to make use of S in any way (that can be replicated by the decoder) to compress T. Perhaps the most common and effective form of differential file compression is to copy substrings from the source file and then insert new characters to fill the gaps. In fact, differential file compression is often defined to be the representation of a target file by a sequence of copies and inserts. The identification of common substrings of S and T also drives the algorithms presented here, where we employ data compression / decompression algorithms to fill in the gaps.

For a *target file T* of size n and a *source file S* of size m, we define a differential decompression algorithm to be *in-place* if it uses $m + n + O(1)$ space for encoding and $\text{MAX}(m,n) + O(1)$ space for decoding. The $O(1)$ term represents the constant amount of space to store the program itself along with a few local variables to index loops, etc. Beyond this constant amount of space, at no time does the decoder use more than $\text{MAX}(m,n)$ space to read bits of the compressed data stream and construct T (by overwriting S). Similarly, for an encoder to be in-place, beyond the constant O(1) space, it uses only $m + n$ space (no additional space beyond that to store S and T) to produce the bits of the compressed stream. Although in-place encoding is possible, we will allow $O(m+n)$ space for the encoder so as to improve its speed, and present very fast decoding with only $\text{MAX}(m,n) + O(1)$ space.

Decoding consumes the compressed stream in a read-only serial fashion that examines each bit of the compressed stream exactly once (such as coming from a communications line or pipe); the bits of the compressed stream are not part of the space being measured; that is, the MAX*(m,n)* + *O*(1) space for decoding is the space used by the decoder to consume the compressed stream and replace *S* by *T*, but does not include the space for storage of the compressed stream itself (for example, if it came from a storage device rather than a communications line). To handle the case when *T* is not compressible with respect to *S*, we assume that the compressed file format starts with a single bit that indicates whether the remainder of the file is compressed with respect to *S* or not. In practice, files distributed with differential compression would likely have a *O*(1) size file header that contain a number of bits of key information.

One application of differential compression is the distribution of successive versions of a file. 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. Fast in-place decompression is desirable, since the distributor may not want to make assumptions about the resources available to the user. 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. Although it may only be speed of encoding that is important for this application, if compressed archiving is also provided by the file backup system, then in-place decoding can also be desirable. Differential file compression can also be a powerful string similarity test in browsing and filtering applications.

Traditional differencing algorithms compress data by finding common strings between two versions of a file and replace 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. To illustrate the difference between the LCS and edit-distance approaches consider the two strings *ABC* and *BAC*. The longest common subsequence approach finds a LCS of two characters (*AC* or *BC*) and then inserts a third character (e.g., extract the common subsequence *AC* and then insert a *B* at the beginning). The edit-distance approach changes *ABC* to become *BAC* through a sequence of edits. If the only edit operations are insert-character and delete-character, then 2 operations suffice (e.g., delete *B* and insert it at the beginning), but if operations like move-character or transpose two adjacent characters are allowed, then one operation suffices for this example. For the case of unit cost insert and delete character operations, LCS and edit distance are equivalent (e.g., see the book of Storer [2002]). 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 [1984] 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 [1973] 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 add

command when no copy can be found); this left-to-right greedy approach is optimal (e.g., Burns and Long [1997a], Storer and Szymanski [1978, 1982]). Hunt, Vo, and Tichy [1998] 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 algorithm based on LZ techniques significantly outperform LCS based algorithms in terms of compression performance.

Ergun, Muthukrishnan, and Sahinalp [2002] consider the edit distance with block moves, deletes, and copies, and present a polynomial time algorithm with a factor of 12 approximation to optimal. Shapira and Storer [2004b] contains a proof of a factor of 4. Bentley and McIlroy [1999] present a pre-compression algorithm that employs finger print techniques to identify long repeated substrings. Ajtai, Burns, Fagin, and Long [2002] and Burns and Long [1997b] 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). Agarwal, Amalapurapu, and Jain [2003] speed up differential compression with hashing techniques and additional data structures such as suffix arrays. Heckel [1978] presents a linear time algorithm for detecting block moves using Longest Common Subsequences techniques. One of his motivations was the comparing of two versions of a source program or other file in order to display the differences.

Factor, Sheinwald, and Yassour [2001] employ Lempel-Ziv [1977, 1978] based compression 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. Contrarily to their model of using multiple files as a reference set, we use only a single reference file. Their experiments show that the best compression is often achieved using only a single reference file.

Burns and Long [1998] 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, causing a *write before read conflict*. Their algorithm modifies a standard delta file so that the new version could be built in the space occupied by the current version. They detect and eliminate such conflicts by replacing problematic copy items with character insertions, increasing the size of the delta encoding. Burns and Long [2003] show that when the input is a difference file and the objective is to modify it to be in-place reconstructable, the problem is NP-hard. They also show that when the input is source and target files and the objective is to find an in-place reconstructable difference file, the problem remains NP-hard.

Shapira and Storer [2004b] present a constant factor approximation algorithm for the general (NP-hard) problem of finding the edit distance with character insert, block move, block delete and block copy of two input strings. Here we consider practical in-place differential file compression algorithms based on traditional sliding window data compression, such as that used by the UNIX *gzip* compression utility. Although NP-hardness implies that these algorithms cannot always be optimal for particular finite length strings, the asymptotic optimality for an information source of sliding window methods, along with their ability for constant-factor approximation, is evidence that they should work well for this problem in practice. We introduce the *IPSW* algorithm (*In-Place Sliding Window*), which uses a sliding window method to decompress T by initializing the window to S, and restricting its maximum size to $max(|S|,|T|)$; IPSW is fast and the compression achieved compares well with other existing methods, including those that are not in-place. IPSW is most effective when there is a reasonable degree of *alignment* between S and T. That is, large matches between S and T occur in approximately the

same relative order. Such alignment is typical for many applications, such as subsequent backups of the same data, where when new data is inserted, and other data is deleted, the remaining data often comprises aligned common substrings.

We are also interested in good performance of in-place decompression even for those (rare) cases where S and T are not sufficiently well aligned. This could happen, for example, with a software update where for some reason the compiler moved large blocks of code around. An extreme case is when the first and second halves of S have been exchanged to form T ($S = uv$ and $T = vu$); to decompress T the IPSW algorithm overwrites u as it copies v, and then the ability to represent u with a single copy is lost (and the performance of IPSW will simply reflect the compressibility of u and v independently). Rather than modify IPSW, a very fast and practical method that suffices for many and perhaps most practical inputs, we present a preprocessing stage for IPSW that moves substrings within S to better align S with T. Compressed files can be preceded with an initial bit indicating whether preprocessing has occurred. The encoder can compress T with IPSW and compare that to compressing T with respect to S not in place (so at all times all substrings of S are available to be copied). If the difference is significant, this initial bit can be set, alignment preprocessing performed, and a list of moves prepended the normal IPSW encoding. The decoder can perform the moves in-place and then proceed in-place with normal IPSW decoding.

Because the IPSW algorithm works so well, one can view this preprocessing algorithm as something that may be rarely or never used in many practical applications, but which is available for special circumstances when significant gains may be achieved (and this form of access to the data is practical). We present experiments that indicate that these algorithms (IPSW by itself and in some cases IPSW with preprocessing) compare favorably with traditional practical approaches, even those that are not in-place, while at the same time having low encoding complexity and extremely low decoding complexity. An extended abstract of this work is presented in Shapira and Storer [2003,2004a].

Section 2 reviews the algorithm of Burns and Long [1998]. Section 3 presents a practical approach that employs "off-the-shelf" compression algorithms. Section 4 presents the IPSW algorithm and Section 5 present experiments with IPSW. Section 6 outlines the move preprocessing algorithm and Sections 7 and 8 present efficient implementations of key steps of the move preprocessing algorithm. Section 9 presents experiments that compare the performance of IPSW with move preprocessing to that of competing methods that are not in-place, and to performing traditional file compression without the use of differencing. Section 10 concludes.

2. Burns and Long's Algorithm

Delta encoding is a sequence of *copy* and *insert* commands. A *copy* command copies a substring from its source location to its destination, and has the form *(src,dst,l)*, where *src* is the source position of the substring, *dst* is its destination, and *l* is its length. An *insert* command adds a string at a given position, and has the form *(pos, str)* where *pos* points to the position where the string *str* should be located.

Burns and Long [1998] post process delta files so that they are suitable for reconstructing the new version of the file in-place. Given an input delta file, they partition the commands in the file into copies and inserts. Then they sort the copies by increasing write offsets, and construct a directed graph from these commands. The vertices are associated with the copies and an edge from vertex *i* to vertex *j* means that the copy *i* reads from the interval to which *j* writes. They end the process by performing a topological sort on the nodes of the graph, where during this sort, cycles in the graph are broken by eliminating copy nodes. The reconstructed delta file is the topological order of the copies encoded by each node. Copies of the deleted vertices are converted into inserts of the corresponding strings. Each cycle in this graph represents a *write before read conflict*. For example, consider the two strings $S=a^nb^n$ and $T=b^na^n$. The delta file commands constructed are *copy(0,n,n)* for copying a^n, and *copy(n,0,n)* for copying b^n. After using the first command *copy(0,n,n)* the temporary string a^{2n} is constructed. The second command *copy(n,0,n)*, which meant to copy b^n to its final location, now reads the string a^n which results in an incorrect decoded file. Burns and Long [1998] suggest two different heuristics for eliminating vertices to break cycles. The *constant time policy* breaks individual cycles using time *O(1)*, and the *locally minimum* policy eliminates the node that encodes the smallest copied string.

Burns and Long [2003] show that when the input is a difference file and the objective is to modify it to be in-place reconstructable, the problem is NP-hard. This result justifies the use of these heuristics for cycle braking, which can give good performance in practice, but have unbounded performance in the worst case, as can be seen in the following example. Let $S = (ab)^n(ba)^n$ and $T = (ba)^n(ab)^n$. The smallest delta file constructed includes the commands $copy(0,2n,2n)$ for copying $(ab)^n$, and $copy(2n,0,2n)$ for copying $(ba)^n$. In this case, in-place decoding faces, again, the "write before read conflict" since the temporary string $(ab)^{4n}$ is constructed after performing the first copy. Burns and Long eliminate one of these copies (since their cost are equal), and place an insert command (e.g, $(0,ba^n)$ when eliminating the first copy) at the end of the delta file, which increases the delta file from $O(1)$ to $O(n)$. However, decoding can be done in-place by the delta commands $copy(0,2n,2n)$ for copying $(ab)^n$, and $copy(1,0,2n)$ for copying $(ba)^n$. Thus, the delta file's size can remain constant even with in-place decoding.

In this paper we present algorithms for differential file compression that are especially built for in-place decoding, instead of changing the delta files after they are built. We empirically show that our algorithm is competitive with even non in place differencing algorithms. In particular our algorithm compares favorably to the delta compression algorithm of Ajtai, Burns, Fagin, and Long [2002], which is the basis of Burns and Long's algorithm, and an upper bound of Burns and Long's compression performance since they start with the correcting 1.5-pass delta compression algorithm of Ajtai et al. [2002], and then "damage" the compression by eliminating cycles. The 1.5-pass delta compression algorithm of Ajtai et al. uses a hash table which records only a single offset for each footprint value and extends the match backwards and forwards. It first makes a pass over the source file in order to collect some partial information about substrings occurring in the source file. It then uses this information in a pass over the target and source files to find matching substrings.

3. Moving Substrings In-Place

For highly similar files, once the large matches have been identified and appropriately moved, it may not be critical how the remainder of the file is represented, and it may be convenient to use a compression algorithm or standard already available. That is, for a source string S, we process input to transform *S* to *T* as follows (assume that the input has the form of a sequence of move instructions followed by some compressed data):

1. **for** each move instruction received **do**
   Perform the move in-place, using only $O(1)$ space in addition to that used by *S*.
2. Receive data that has been compressed with some standard compression algorithm, perform the appropriate decompression, and use the decompressed data to fill in the gaps between the moved strings.

This approach uses $O(1)$ space in addition to whatever space is used by the decoder. An advantage of this approach is the ability to plug in any lossless compression method for Step 2; when it is a method that uses only $O(1)$ additional space (e.g., UNIX *compress* or *gzip*), the entire algorithm uses only $O(1)$ additional space.

We will now look at both Steps 1 and 2 in more detail. To describe implementations of Step 1, let *S* be a string and use the following notation:

s = a substring of *S* to be moved

l = the length of *s*

x = source location of *s*

y = destination location of *s*

Since the problem is symmetric for left versus right moves, and since we can easily incorporate an offset, assume $x=0$, $y>0$, and y specifies the character that follows *s* after the move (e.g., if *S* = *abcdefgxyz*, $l=3$, and $y=7$, then *abc* is moved so that *S* = *defgabcxyz*).

A naive algorithm, that uses $O(1)$ additional space and $O(y^2)$ time, moves the characters of *s* individually, shifting the entire string between the source and destination locations at each step. For a more efficient algorithm, observe that each character of *s* goes a distance $d = y-l$, and the move is tantamount to rearranging positions *0* through $y-1$ by the permutation $i \rightarrow (i+d) \text{ MOD } y$.

Example: If *S* = ABCDEFGHIJUVW, *s* = ABCDE, and $y=10$, then $l=5$, and *s* is moved to attain FGHIJABCDEUVW. That is, *S* has positions 0 through 12, y points to U that is located at position 10, $d = y - l = 5$, and the permutation $i \rightarrow (i+5) \text{ MOD } 10$ is used ($0 \leq i \leq 10$).

A standard in-place permutation algorithm (e.g., see the book of Storer [2002]) starts with positions 0 through $y-1$ "unmarked" and then for each position, if it is unmarked, follows its cycle in the permutation and marks each position visited, for a total of $O(y)$ time and y additional bits of space for the mark bits. Here, mark bits for only $\text{MIN}(l,d)$ positions are needed, since every cycle passes through at least one of the positions 0 through $l-1$ and at least one of the positions l through $y-1$.

It is likely that in many (or most) practical applications, the space for these mark bits is not significant. However, for theoretical purposes, or perhaps for special purpose applications, we now consider in detail how we can ensure that the additional space used for mark bits is only $O(1)$. If l or d is $O(\log(m))$, then we use $O(1)$ space under our model, since we are assuming at least enough space to store the program and $O(1)$ local variables, which are each capable of

holding $O(\log(m))$ bits (e.g., for $m < 4$ billion, 32 bits suffice to store an index variable in a loop that counts from 1 to m). Otherwise, if we cannot afford the additional bits, in $O(y^{1/2}\log(y))$ time we can proceed as follows:

Step 1: Test y, y–1, y–2, ... until we find the largest prime $p \le y$; that is, $d^{1/2}$ operations suffice to test, and approximately $\ln(y)$ positions are tested, where *ln* denotes the natural logarithm (see the "Prime Number Theorem" in a text such as Rosen [2000]).

Step 2: Let $d' = p–l$, and using no mark bits, move s most of the way by the permutation $i \rightarrow (i+d') \text{ MOD } p$ (since p prime implies that the permutation is a single cycle that can be traversed).

Step 3: After adjusting the offset so that s starts at position 0, move s to position $y' = l+(y–p)$ with the permutation $i \rightarrow (i+(y–p)) \text{ MOD } y'$; since $(y–p) \approx \ln(y) = O(\log(m))$, by the argument above, this can be done with $O(1)$ additional space in our model.

Example:

S = ABCDEFGHIJUVW

s = ABCD

$l = 4$

$y = 10$

$d = 6$

The largest prime ≤ 10 is 7, so $d' = 3$, and in Step 2 we use the permutation $i \rightarrow (i+3)$ MOD 7 to attain S = EFGABCDHIJUVW, by moving s 3 characters instead of 6.

In Step 3 we move s the remaining distance of $d'' = 3$ characters. We first adjust the indexes by $x'' = 0$ and $y'' = 7$ (so we are really looking at the substring ABCDHIJUVW of S), and use the permutation $i \rightarrow (i + 3) \text{ MOD } 7$ to attain our final goal EFGHIJABCDUVW.

It happens to be that in this example d'' and y'' are prime, so no mark bits are needed for moving the string the remaining distance.

This construction works by first moving all of s most of the way, using no mark bits, and then moving s the remaining $O(\log(m))$ distance using mark bits. Alternately, using a slightly stronger version of the prime number theorem, we could move most of s all of the way using no mark bits, and then move the remainder $O(\log(m))$ characters of s using mark bits. We make sure that the permutation $i \rightarrow (i+d) \text{ MOD } y$ is a single cycle by forcing d and y to be relatively prime. This is done by making s shorter (i.e., checking smaller values for l). We then move the rest of the characters of s using mark bits. Again, we should emphasize that this construction may be unneeded in many practical applications where the space for mark bits is not significant in any case.

Because, performing each move in-place as described above is potentially $O(m)$ time, processing a sequence of moves to transform a string S to a string T may be non-linear. A set of moves is *aligned* if when we write T below S and draw a straight line from each character in S that is moved to where it is moved in T, then no lines cross. For applications where it is typical that moves are aligned, such as subsequent versions of the same software, we can perform all moves for a string of length m in a total of $O(m)$ time and $O(1)$ additional space, using only simple

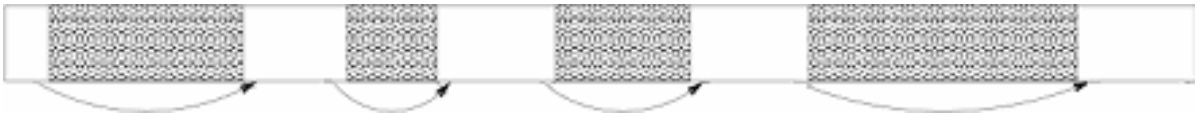
string copies (that may overwrite characters that are no longer needed). The file is scanned twice, first from left to right to copy blocks that go to the left and then from right to left to copy blocks that go to the right. Let x_i, y_i and l_i, $1 \le i \le n$, denote the source and destination locations and the lengths of k aligned move operations for a string stored in the array A:

for i:=1 **to** n **do if** i is a source position such that $x_i > y_i$ **then**
    for j:=1 **to** l_i **do** $A[y_i + j] := A[x_i + j]$

for i:=n **downto** 1 **do if** i is a source position such that $x_i < y_i$ **then**
    for j:=l_i **downto** 1 **do** $A[y_i + j] := A[x_i + j]$

Since this approach is intended for when large aligned moves exist, any reasonable encoder (which need not work in-place) to construct a sequence of aligned moves may suffice in practice. For example, a greedy approach can parse the text twice, using two thresholds: C_1 and C_2, where C_1 is much larger than C_2. First extract matches longer than C_1, and then search between two subsequent aligned matches for matches longer than C_2.

Assuming that a move operation is represented by three components (source position, destination position, and length), after the preprocessing step is complete, and the decoder has performed all of the move operations, the gaps can now be filled in by running a standard decompressor on the remaining bits so long as we are careful during preprocessing to remember these positions. This can be done by linking them together as depicted in the following figure; the encoder can be implemented to insure that each gap is large enough to hold a pointer (e.g., 4 bytes to handle moves of 4 billion characters).



From the point of view of the off-the-shelf decoder, a contiguous string is produced, which we just happen to "chop" up to fill the gaps. Another approach is to modify the off-the-shelf decoder slightly so that gaps are compressed in their local context. For example, with a sliding window method, the window could be defined to go contiguously back to the left end of the current gap and on back into the previous block, or alternately, a single pointer value could be "robbed" to use as an escape (followed by a displacement and length) to a match into the decoded string rather than into its own $O(1)$ memory that stores a fixed length sliding window (e.g. 4K to 32K for UNIX *gzip*).

4. In-Place Sliding Window (IPSW)

As reviewed in Section 1, given a string S of length m and a string T of length n, and the set of operations *character insertions, block deletions, block moves,* and *block copies*, computing the minimum number of operations required to transform S to T is known to be NP-complete. However, it is also known that this problem can be approximated to within a constant factor by a simple left-to-right greedy copying algorithm (e.g., Ergun, Muthukrishnan, and Sahinalp [2002] show a factor of 12 and Shapira and Storer [2004b] show a factor of 4), which we refer to as the *Full Window Algorithm*:

Full Window (FW) Encoding Algorithm:

Step 1: Form the string ST.

Step 2: Compress T with a sliding window of size $m+n$. Use unit-cost *greedy* parsing. That is, start with the first position of T, then repeatedly find the longest match between the incoming text and the text to the left of the current position, and replace it by a copy of unit cost (or by a single character, also of unit cost, if a match of 2 or more characters cannot be found).

Step 3: Delete S.

Motivated by the existence of FW (a greedy approximation algorithm that is not in-place), for in-place differencing of a target file of size n with respect to a source file of size m, we do:

ISPW Encoding Algorithm:

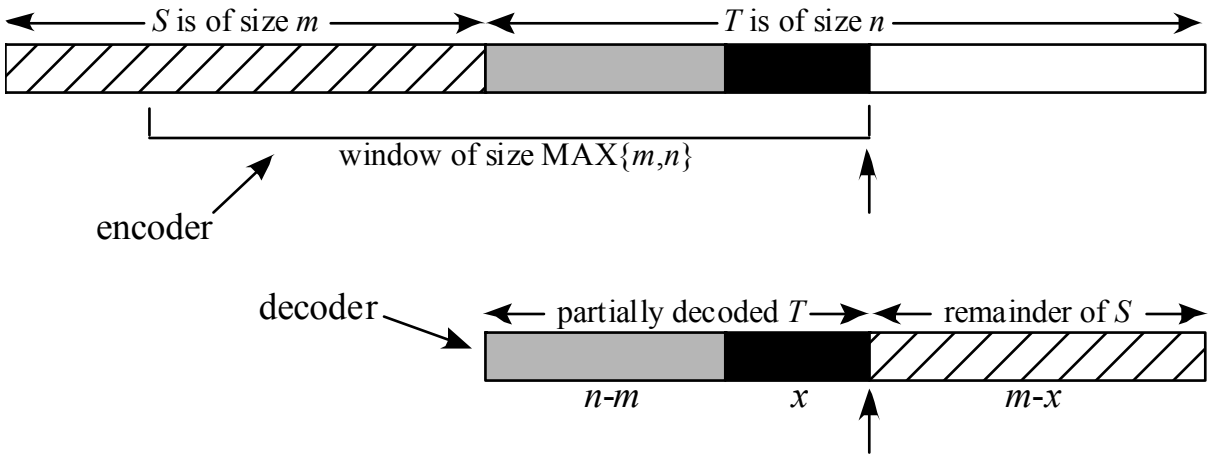
Step 1: Form the string ST.

Step 2: Compress T with a sliding window of size MAX(m,n). Use unit-cost greedy parsing.

The only difference between ISPW and FW, is that we slide a window of size MAX(m,n) (so except at the start, copies may not be able to reach all the way back to the start of S). By overwriting the no longer accessible portion of S from left to right, we can decompress in-place, using MAX(m,n) + O(1) space, and are effectively deleting S as we go, rather than all at once at the end as in FW. For many practical file differencing applications where there is a high degree of alignment between S and T, this more aggressive deletion will not diminish practical performance.

Step 1 of IPSW encoding amounts to initializing the sliding window to be the source file and could be implemented by dividing a pointer that spans the source and target files into two memory references. That is, if the pointer happens to point to some string which starts at some position of S and ends at a position in T, we can break it into two: one pointer which points to a suffix of S, and the other which points to a prefix of T. If an additional k units of space are available, encoding can be generalized to use a window of size $m+k$ (although variable length coding of offsets might have to be tweaked for the larger window); however, here we limit our attention to the case $k = 0$.

We assume that some reasonable implementation of sliding window encoding is used and do not address the details of encoding time or space. We focus on the fast in-place decoding that is possible with this approach.



The figure above depicts encoding (top) and decoding (bottom) for when the target file *T* is 50 percent larger than the source file *S*. The arrow points to the current position. The hatched region is all of *S* on the top and the remaining portion of *S* on the bottom. The lightly shaded region is the *n–m* characters of *T* that have already been encoded and decoded by the decoder without having to overwrite *S*. The dark shaded region is the portion of *T* that has already been encoded by overwriting the first *x* characters of *S*. The remaining *m–x* characters of *T* have yet to be encoded, and hence the rightmost *m–x* characters of *S* are still available to the decoder. The decoder's window can be viewed as two pieces: the already decompressed portion of *T* (the lightly shaded and darkly shaded regions) that is to the left of the pointer and the remaining portion of *S* that is to the right of the pointer (but was to the left of the lightly shaded and darkly shaded regions when encoding). So for each pointer decoded, at most two string copies must be performed: a single copy when the match is contained entirely in *S* (hatched region) or entirely in *T* (lightly shaded and darkly shaded regions) or two copies when the match starts in *S* and ends in *T* (the encoder encodes a match that crosses the boundary between the hatched region and the lightly shaded region). Since in many practical applications matches that cross this boundary are really just "lucky" (i.e., we may be primarily looking for large matches to *S* and if they are not found then a match in *T* that is a shorter distance away), an alternate implementation is to forbid copies that cross this boundary, in order to further simplify decoding.

A key observation, which is an inherent property of sliding window compression, is that besides (possibly) some variable length decoding of the length and offset fields of the pointer, the only computation is a simple string copy (or in this case at most two string copies). Decoding can be implemented to be in-place and about as fast as one could possibly hope for. This inherent decompression speed is one of the reasons that sliding-window based compression utilities such as *gzip* have been so widely used. Assuming that decompression is done in memory, our use of a possibly very large window (depending on the source file size) with an in-place implementation will have no effect on speed, other than possibly a minor effect on the variable length encoding of pointer fields to accommodate large matches to the source file. For example, it is trivial to modify the UNIX *gzip* utility to use a simple escape code for long pointers (e.g. reserve the longest possible length or displacement for the escape code) so that the same algorithm can be used for normal matches and the increased pointer length is only charged to large matches in the source file.

5. Experiments with IPSW

Here we present some preliminary experiments with IPSW; later we will present additional experiments after developing the move preprocessing algorithm. Since IPSW uses a sliding window, its processing time is similar to the running time of gzip. Here we focus on the compression achieved by IPSW.

To perform experiments with our IPSW algorithm, we used a pointer format that begins with a control integer between 0 and 4 that indicates one of the following cases:

Case 0: No displacement/length; the bits to follow indicate a single raw character.

Case 1: Displacements less than 4,096; match length < 256 characters.

Case 2: Displacements between 4KB and 32KB; match length < 256 characters.

Case 3: Displacements larger than 32KB; match length < 256 characters.

Case 4: Displacements larger than 32KB; match length ≥ 256 characters.

This format is constructed around key parameters of the UNIX *gzip* utility that uses a window of size 32K and maximum match length of size 256 characters. Unlike the "tuned" variable length encoding used by *gzip*, for simplicity, we encoded the control integers, the raw character for Case 0, and the length fields for Cases 1 through 3 with separate off-the-shelf arithmetic encoders. Although a relatively powerful technique such as arithmetic coding is not needed for efficient encoding of sliding window fields, rather than try to re-code or modify the *gzip* algorithm ourselves, we have essentially approximated *gzip* performance in the 32K portion of the window by using a fancier length encoding and a more crude displacement encoding (a fixed code of either 12 or 15 bits for Cases 1 and 2, a number of bits needed to reference a position in the original file for Cases 3 and 4). To verify that we are not "cheating" and using a more powerful algorithm than used by the very fast and practical *gzip* algorithm, we first performed experiments without file differencing on the standard Calgary Corpus to see how the compression achieved by our simple sliding window implementation compares to that of *gzip*. As can be seen from Table 1, IPSW consistently performs comparably to *gzip* (usually slightly worse), and one can expect that the compression we achieve here will only be similar or slightly improved by using a *gzip* implementation.

	size	**gzip**	**IPSW**
Paper1	53,161	18,577 (34.9%)	19,579 (36.8%)
Paper2	82,199	29,753 (36.2%)	30,957 (37.7%)
Bib	111,261	35,063 (31.5%)	35,559 (32.0%)
news	377,109	144,840 (38.5%)	138,361 (36.7%)
pic	513,216	56,442 (11.0%)	59,234 (11.5%)
book1	768,771	313,376 (40.8%)	300,632(39.1%)

Table 1: Compression Performance without differencing.

Our second experiment is to compare the performance of IPSW and NIP (stands for Not IN Place, that is, IPSW with a window of length $|S|+|T|$ so that at every step, matches can reach any position in S or the already compressed portion of T) with the work of Factor, Sheinwald and Yassour [2001] (FSY), who compressed *Netscape Communicator* version 4.7 with respect to version 4.5. They compressed *ldap3230-47.dll* against several sets of reference files. Starting with a large set, sorted by frequency divided by length, they removed files from the end of the list as long as the compression ratio did not increase. They found compression performance best when compressing *ldap3230-47.dll* only against *ldap3230-45.dll*. They also compressed the main Netscape Version 4.7 executable *netscape-47.exe* of 5,530,208 byte long, with respect to its 4.5 version, *netscape-45.exe*, of 5,485,680 byte long. They ran two variants of their algorithm: the one pass greedy selection of longest possible match at each step, which compressed to 1,584,094 bytes, and multi-pass selection of matches in descending order of lengths, which compressed to 1,482,740 bytes. As can be seen, IPSW, (even with our crude implementation of sliding window pointer encoding) performs comparable to FSY while working in-place. As a reference of how compressible the files are on a standard stand-alone basis, we have included the compression achieved by gzip. We have also included figures for the UNIX *bzip2* utility (based on the Burrows-Wheeler transform and not in-place), which is becoming a common reference for practical file compressibility.

	size	**gzip**	**bzip2**	**FSY**	**IPSW**	**NIP**
Ldap3230-47.dll	132,608	60,058	57,980	28,179	31,952	29,044
Netscape-47.exe	5,530,208	2,602,880	2,508,166	1,482,740	1,401,295	1,422,723

Table 2. Comparison with Factor, Sheinwald and Yassour (FSY) algorithm.

We now compare performance for in-place differencing, when two versions of the same software are given. We used following files:

- GNU *sql* server (portable multi-user relational database management), versions 0.7b3, 0.7b4, 0.7b5, 0.7b6,, 0.7b6.1, source code.
- GNU *gcc* (C compiler), versions 3.0, 3.1, 3.1.1, 3.2 and 3.2.1, executables.
- GNU *xfig* (utilities to draw and manipulate objects), versions 3.2.1, 3.2.2 and 3.2.3, source code and executables.

The differencing results presented in Table 3 were obtained by compressing each file against its previous release listed in its preceding column. Thus, the first column shows the compression performance without file differencing. We have compared our compression performance against the UNIX *xdelta* and *zdelta* differencing utilities with compression; these algorithms compare the files and compress their difference using the *zlib* library (default compress level was used). Again, *NIP*, *gzip* and *bzip2* numbers are included for reference.

As can be see from Table 3, the amount of compression we achieve is comparable to that of *xdelta* and *zdelta* and to that of Factor, Sheinwald and Yassour [2001] (sometimes a bit worse, sometimes a bit better) and is done in-place. Further more, an algorithm like *gzip* could easily be modified to perform efficient encoding and very fast decoding (with possibly slightly greater compression performance than we have reported here). Note that IPSW is the same as NIP in the first column, since compression in this column is without use of a source file.

	sql-0.7b3	sql-0.7b4	Sql-0.7b5	sql-0.7b6.0	sql-0.7b6.1
size	2,722,338	2,920,666	5,861,140	3,371,146	2,981,200
gzip	618,623	666,545	1,342,795	719,702	682,383
bzip2	508,537	547,856	1,103,992	583,031	562,714
xdelta	618,933	118,835	27,357	163,024	14,084
zdelta	638,017	101,890	687,468	262,770	79,557
IPSW	556,400	89,087	16,343	111,265	8,111
NIP	556,400	90,836	16,676	111,256	8,305

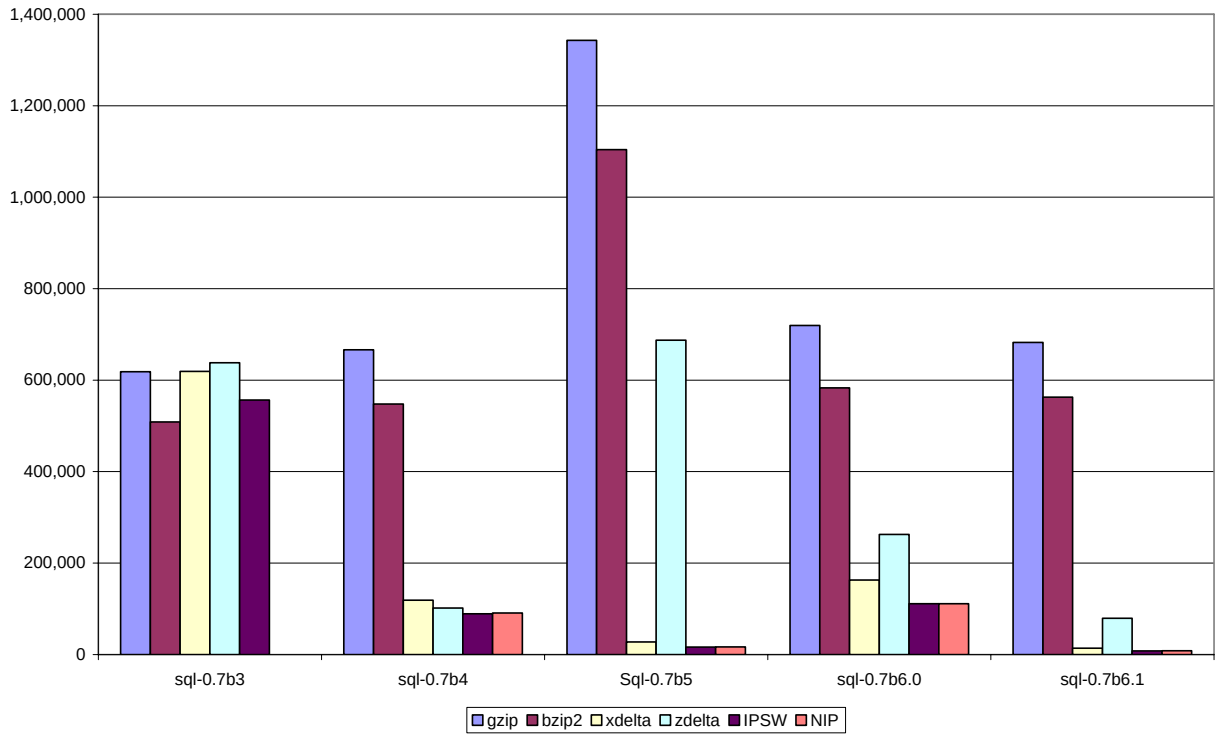


Table 3a: Sql - compression performance given a reference file.

	gcc-3.0	Gcc-3.0.3	gcc-3.0.4	gcc-3.1	gcc-3.1.1	gcc-3.2	gcc-3.2.1
Size	274,981	277,055	277,335	299,671	299,687	264,805	302,897
Gzip	95,237	95,915	95,976	111,538	111,491	91,443	112,866
Bzip2	91,596	92,282	92,446	110,700	110,611	87,842	111,713
Xdelta	96,381	48,028	30,755	97,256	44,791	70,682	95,379
Zdelta	99,377	33,716	16,174	104,888	22,182	67,146	106,451
IPSW	102,073	45,961	27,328	90,680	57,694	84,826	88,901
NIP	102,073	45,962	25,248	90,417	34,796	73,879	87,094

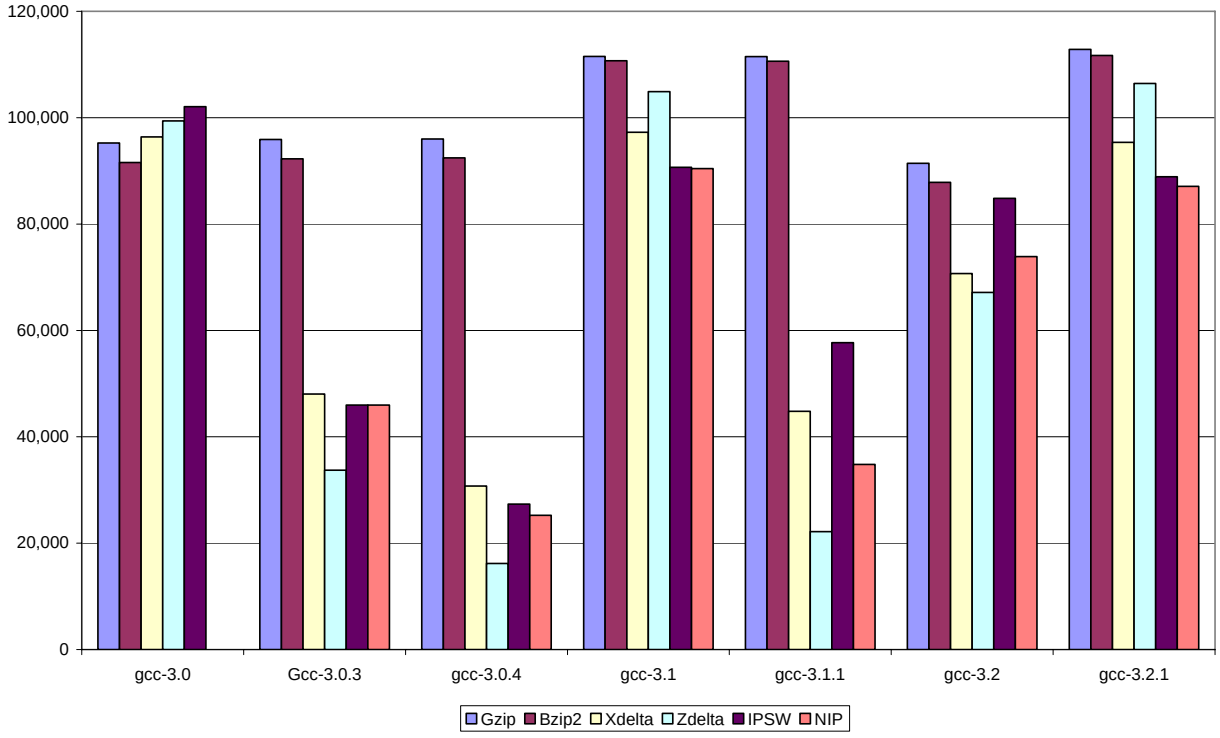


Table 3b: Gcc - compression performance given a reference file.

	xfig.3.2.1	xfig.3.2.2	xfig.3.2.3	xfig.3.2.1.exe	xfig.3.2.2.exe	xfig.3.2.3.exe
Size	5,371,715	5,372,916	5,996,776	831,748	831,748	926,785
Gzip	1,971,093	1,971,299	2,109,395	332,956	332,957	373,274
Bzip2	1,838,434	1,839,168	1,907,723	320,651	320,701	358,147
xdelta	1,967,704	96,098	1,253,875	334,107	903	300,904
zdelta	1,999,542	1,335,540	2,105,610	345,193	911	255,432
IPSW	1,924,023	134,434	1,165,696	351,244	491	288,473
NIP	1,924,023				426	287,719

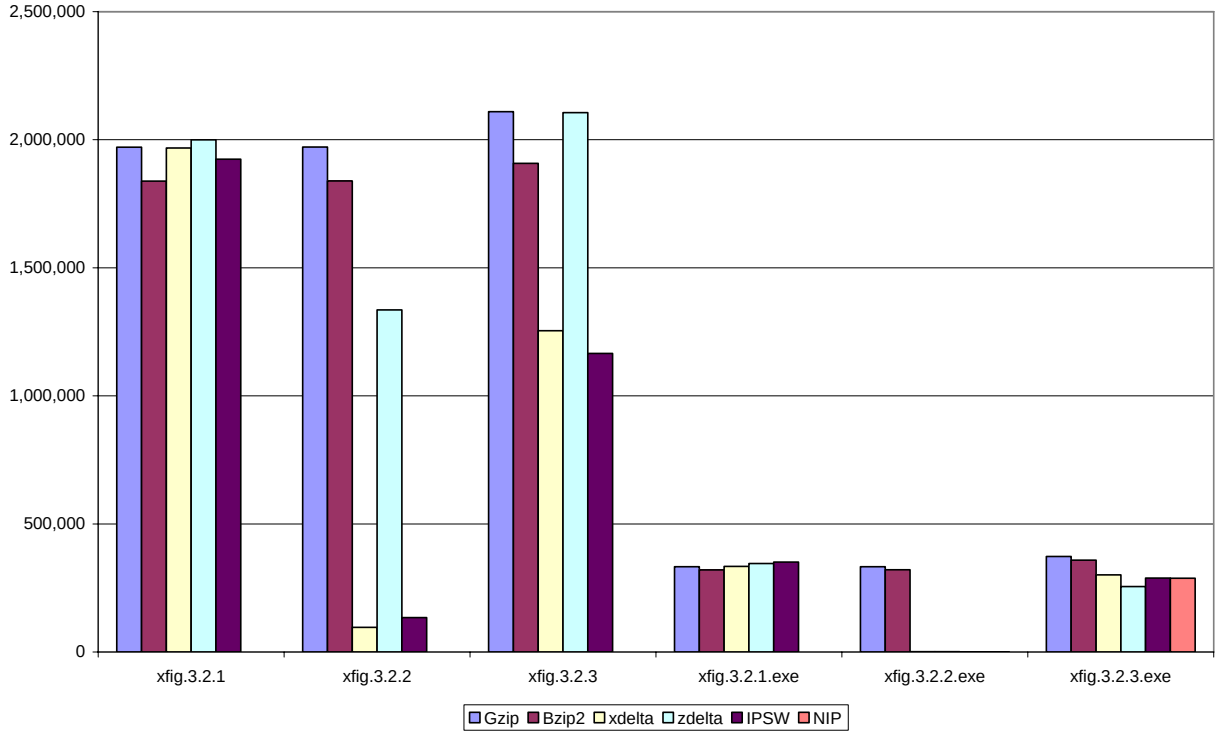


Table 3c: zfig - compression performance given a reference file.

6. The Move Preprocessing Algorithm

If the encoder determines that S and T are not well aligned, then preprocessing for the IPSW algorithm finds a set of substring moves to convert S to a new string S' that is well aligned with T. We limit our attention to moves that are non-overlapping, where the moves define a *parsing* of S and T into *matched blocks*, $\{b_i\}_{i=1..r}$, and *junk blocks*, $\{x_i,y_i\}_{i=1..r}$; that is, $S=x_0 \cdot b_{\sigma(1)} \cdot x_1 \cdot b_{\sigma(2)} \cdot x_2 \cdots x_{r-1} \cdot b_{\sigma(r)} \cdot x_r$ and $T=y_0 \cdot b_1 \cdot y_1 \cdot b_2 \cdot y_2 \cdots y_{r-1} \cdot b_r \cdot y_r$. When using the sliding window method, we would like to copy substrings of S only from the part that was not yet overwritten by the reconstruction of T. That is, we would like to perform only *left copies*, i.e., a copy (s_i, d_i, l_i) that copies a substring with l_i characters from position s_i to position d_i that satisfies $s_i \geq d_i$. Right copies (i.e., when a copy (s_i, d_i, l_i) satisfies $s_i < d_i$) raises the *write before read conflict*.

By employing a reduction from the edit-distance with block moves problem (shown NP-complete in Shapira and Storer [2002]), it can be shown that given a parsing of two strings S and T into match and junk blocks, it is NP-complete to determine whether S can be rearranged with cost $\leq k$, so that all match blocks are left copies.

Since this implies that an efficient optimal algorithm is unlikely to be practical, we use an approach motivated by the approximation algorithm presented in Shapira and Storer [2002].

We employ two different kinds of rearrangements of the blocks. *Moves* rearrange the blocks $\{b_i\}_{i=1..r}$ so that they occur in S and T at the same order. *Jigglings* move the junk blocks of S backwards in the file, so that, as a side effect, the matched blocks are moved forwards.

Move Preprocessing Algorithm:

Step 1: Find Non Overlapping Common Substrings (NOCS) from longest to shortest down to length of a constant K; These are the $\{b_i\}_{i=1..r}$ blocks.

Step 2: Compute the minimum number of rearrangements (moves and jigglings) in S so that the blocks $\{b_i\}_{i=1..r}$ are left copies within S and T.

Step 3: Generate S' according to step 2.

Step 4: Compute IPSW(S',T).

The next subsection shows how to implement Step 1 in $O(m+n)$ time using a suffix tree, and the following subsection shows how Step 2 can be done in quadratic time of the number of blocks, which is in the worst case $O((n/K)^2)$. However, in practice, the number of blocks is much smaller than $\sqrt{n}$, and Step 2 is linear time. Generating S' in Step 3 can be done in linear time by copying the strings to a different location (again, we are allowing the encoder to work not in-place for improved efficiency of encoding). Step 4 uses a sliding window and can be performed in linear time (e.g., see the book of Storer [2002]). Hence, the total time is linear in practice.

6.1 Non-Overlapping Common Substrings (NOCS)

In this subsection we outline a linear time algorithm to construct a list of non-overlapping common substrings. Lopresti and Tomkins [1997] (see also Shapira and Storer [2002]) prove that finding a list of non-overlapping common substrings of S and T with minimum number of substrings is NP-hard. A *maximal unique match* (MUM) is a unique match of S and T that is not a substring of any other unique match of S and T. In contrast to our problem, MUM(S,T) can include overlapping substrings. Delcher at al. [1999] and Hon and Sadakane [2002] present linear time algorithms for computing MUMs. They take advantage of the fact that only the lowest level internal nodes in a suffix tree may represent a MUM. However, to compute a list of greedy non-overlapping common substrings, one must consider common substrings represented by nodes higher up in the tree. The *string statistics problem* consists of preprocessing a string of

length n such that given a query pattern of length m, the maximum number of non-overlapping occurrences of the query pattern in the string can be reported efficiently. Apostolico and Preparata [1985] introduce the minimal augmented suffix tree (MAST) as a data structure for the string statistics problem, and show how to construct the MAST in time $O(n \log^2 n)$, and how it supports queries in time $O(m)$ for constant sized alphabets.

Finding the longest common substring of S and T in linear time can be solved by constructing a compact suffix tree out of the string $S\$T\#$, where \$ and # are new characters, and performing a post-order traversal to find the deepest node that has both S and T leaves descended from it (for example see the book of Storer [2002]). We use this in the following algorithm:

Algorithm for finding the NOCS of S and T :

Step 1: Given two strings S and T, construct a compact suffix tree for $S\$T\#$.

Step 2: Sort all common substrings by length, from longest to shortest.

Step 3: Start from an empty list L, go through the sorted set of substrings and insert to L only those that do not overlap the substrings already in L.

6.2 Computing the NOCS in linear Time

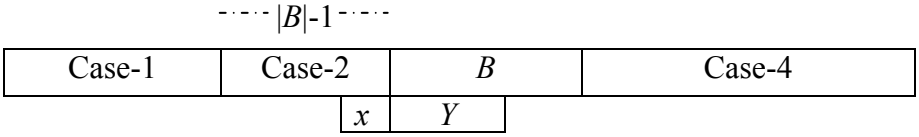
We now describe how the algorithm for finding *NOCS of S and T* can be implemented to run in linear time.

Step 1 can be done in linear time using any of the well-known algorithms (assuming a constant size alphabet — e.g., Ukkonen [1995], McCreight [1976], or Weiner [1973]). All leaves descended from an internal node (either directly or indirectly) point to a suffix of S or T whose prefix is the substring represented by that node. If both S and T leaves are descended from this node, then it represents a common substring of S and T. Since some nodes may represent common substrings within S or within T, but not between S and T we first wish to filter them out. Step 2 can employ a bucket sort. We first perform a post-order traversal of the suffix tree and mark all nodes that have both S and T leaves descended from them. We create an array, A, of MIN(m,n) linked lists, and insert each one of these suffix tree nodes into a linked list based on the length of the common substring it represents. Nodes representing common substrings of length i go into the i^{th} linked list. Processing each internal node take constant time. Since there are at most $m+n$ internal nodes, sorting the nodes by length into the array A takes O($m+n$) time.

It remains to prove that Step 3 can also be implemented in linear time. We show that finding each NOCS of size $|B|$ takes O($|B|$) processing time. Since these NOCS do not overlap, the total time is linear.

Step 3 incrementally builds L to obtain NOCS(S,T). We are interested in maintaining the suffix tree so that finding the next longest common substring of S and T that does not overlap the existing ones in L, takes linear time in total. Assume that we are working on the current longest common substring B that corresponds to an internal node v and a leaf V associated with the suffix of S starting at B (for simplicity, we refer to S leaves only). The length of the common substring is denoted by $|B|$. As shown in the figure below, the substring B divides S into 4 sections. A prefix of S is unaffected by choosing B, because all future common substrings will be at most $|B|$ characters in length. So all leaves that correspond to positions of S that are $|B|$ or more characters before B are not effected. We classify leaves that correspond to this prefix as *Case-1 leaves*. The leaves that correspond to the first $|B|$ characters before B are affected by choosing B; they belong in *Case-2 leaves*, and may still be available for later matches. *Case-3* leaves represent suffixes of

S that overlap the common substring *B*. The remaining leaves are *Case-4* leaves and are unaffected by choosing *B*.



Case-2 leaves are leaves whose potential match length is shortened by the selection of that particular instance of the common substring. There are |*B*|–1 Case-2 leaves. We move each of these leaves to a shorter match length in our array *A,* so that the common substring they represent does not overlap the current common substring *B*. Let *W* be a Case-2 leaf corresponding to a common substring *w* that overlaps *B*. That is, *w* is of the form *xy*, where x and y are at least one character long, *x* is a suffix of *w,* and *y* is a prefix of *B*. Let *x=az* for some character *a* and string *z*. In the suffix tree of *S$T#* the node associated with *zy* is linked to the node associated with *azy*. Since *y* is a prefix of *B* it corresponds to a prefix of the path from the root to the internal node *v*. We can reach *w* by |*x*| *reversed suffix links*. Each reversed suffix link deals with one Case-2 leaf in O(1) time, thus the total processing time for Case-2 leaves is O(|*B*|). Once the internal node *w* has been reached, we shorten the length of the suffix *W* from its current match length (the depth of *w*) to |*x*| by moving it to the |*x*| position in *A*.

Case-3 leaves point to suffixes that start within the current common substring *B*. They clearly overlap it and therefore can never be added to *L*. Therefore, they all need to be removed from our array *A*. Since Case-3 leaves correspond to a continuous substring of *S*, we can scan through *S*, starting at *B,* until we have removed all |*B*| Case-3 leaves. This operation takes O(1) time for each leaf, for a total of O(|*B*|) time for all case-3 leaves. Given the fact that the sum of the lengths of the non-overlapping common substrings is at most *m*+*n* the entire processing time for all Case-2 and Case-3 leaves is *O*(*m*+*n*).

Example:

Consider *S* = *bbaaba***abcb** and *T* = **abcb***aaabbb* and assume that we are currently dealing with the LCS *abcb*, which corresponds to node *v* in the suffix tree. When processing both leaves descended from *v*, we are dealing with the occurrence of *abcb* in both *S* and *T*. For simplicity, we refer to cases only in *S*. The Case-2 leaves include positions 4 through 6. The Case-2 leaf, *aab*, corresponding to *aab* at position 6, must be shortened to include only *a*, since its suffix *ab* overlaps the LCS. It is therefore moved to the list corresponding to a shorter common substrings of length 1. Other Case-2 leaves do not represent a common substring that overlap *abcb*, and therefore are unaffected. The Case-3 leaves are those corresponding to positions 7-10, which need to be removed from the array of common substrings.

6.3 Finding Minimum Number of Rearrangements

Once large matching blocks of *S* and *T* are identified, we can permute them to better align *S* with *T*. Shapira and Storer [2002] show that it is NP-complete to compute minimum edit distance when block moves are allowed (for moves of single characters, they present an optimal dynamic programming algorithm). Bafna and Pevzner [1998] study block moves when *S* is a permutation of integers 1 through *r*, and give a 1.5 approximation algorithm (the restriction that all characters are distinct greatly changes the problem). Given the theoretical difficulty of the problem, here we present a computationally efficient approach that works well in practice.

The NOCS that were found in the previous section are renamed using different characters. After performing the algorithm of Shapira and Storer for edit-distance with character moves, we attain

the moved characters for the minimum cost which correspond to moving the NOCS. Our next goal is to produce a (source, destination) format for the NOCS moves to be sent to the decoder based on the character moves. For example, when dealing with 5 NOCS renamed by {1,2,3,4,5} one possibility to obtain 12345 from 15432 by character moves is by moving 2 and 3 forwards, and moving 4 backwards. Another option for transforming 15432 to 12345 is by moving 3 and 5 backwards, and moving 4 forwards. Each one of these solutions can be obtained from the dynamic-programming table from different optimal paths going from cell [*r*,*r*] (for integers 1 through *r*) back to cell [0,0]. We can extract their different alignment, as shown in the following figure:

1	-	-	-	5	4	3	2		1	5	4	3	2	-	-	-
1	2	3	4	5	-	-	-		1	-	-	-	2	3	4	5

The source and destination positions depend on the order the moves are performed. Therefore, the encoder performs and reports one character move at a time, updating the source and destination locations, before proceeding with the next move. We define each item's destination to be the corresponding positions just after the last aligned item to its left. If there is no such item, the item is moved to the beginning of the array. The move causes a shift of all items to its right, and the item is marked as an aligned item after the move is performed.

Let $\{b_i\}_{i=1..r}$ be a set of r matched blocks to be moved from source positions s_i to destination position d_i, and let $\{x_i\}_{i=1..r}$ be the 'junk' blocks we wish to jiggle to the beginning of S, so that all blocks $\{b_i\}_{i=1..r}$ perform only left copies, i.e., $s_i \geq d_i$. After performing the block moves in S to obtain S', we have $S' = x_0 b_1 \cdot x_1 \cdot b_2 \cdot x_2 \cdots x_{r-1} \cdot b_r \cdot x_r$, and $T = y_0 b_1 \cdot y_1 \cdot b_2 \cdot y_2 \cdots y_{r-1} \cdot b_r \cdot y_r$. To see that it is always possible to perform jigglings so that the matched blocks become left copies consider the worst situation, where all blocks $\{b_i\}_{i=1..r}$ are shifted all the way to the right, without ruining their order obtained from the edit-distance algorithm. Since the space available is at least $|T|$, we are left with only left copies. Thus, in the worst situation we perform r-1 moves and r jigglings. We are interested in minimizing this number.

Each block that was moved already in the edit-distance with move algorithm, is farther moved to the right, so that it is now adjacent to its right non-junk neighbor. These moves are done for free, since they can be moved to the final point directly. At each stage of the jiggling algorithm, when we reach an illegal copy (i.e., a right copy), we choose the longest junk block to its right, and move it to the left of the leftmost illegal copy, which shifts all other blocks to the right.

7. Experiments with MP+IPSW

We implemented the move preprocessing algorithm and evaluated its performance on some test files when used as preprocessing for IPSW. The Move Preprocessing Algorithm finds NOCS(*S*,*T*) from longest down to a length of 256 bytes, assigning increasing indices. Their index, positions, and lengths in *S* and *T* were then recorded into tables *SA* and *TA* respectively. Sorting *SA* by position permutes the indices. The *junk-blocks* in *S* are the blocks between successive NOCS blocks. (*r+1* junk blocks for *r* NOCS blocks). We then computed the edit-distance between the two sets of indices and recorded the move operations. The moves were then performed on *SA* recording the (source position, destination position, length) for each moved block in a list *M* of all moves. Going from right to left in *SA*, and for each item that has moved, we shifted it to the right of its junk, and updated the destination positions of such moves in *M*. Next we performed the jigglings on *SA* and recorded additional moves in *M*. According to array *SA* the string *S*' was constructed from *S*. The compressed representation of *T* consists of the number of moves and jigglings, followed by the list of the moves and jigglings, and by the compressed file obtained from IPSW*(S',T)*.

Table 4 lists the files used for our experiments, where each has an *S* version and a *T* version, and is listed with its size (in bytes) and version number. The first 5 files are executables from *MS Windows*, the *GNU sql server* files are source code, the *GNU gcc* files and *samba* files are executables. The last file, *Random*, is 1 million randomly generated bytes; two *T* files were formed by choosing *r*–1 random positions to define *r* blocks, and then randomly permuting the blocks; for *r*=20 and *r*=100.

	S	**Version**	***T***	**Version**
Ldap3230.dll	113,152	Netscape 4.5	132,608	Netscape 4.7
Agm.dll	582,144	3.2.28.1	582,656	3.2.28.1
Msxml.dll	1,118,720	8.20.8730.1	1,120,768	8.20.9415.0
Shell32.dll	1,391,376	4.72.3812.600	2,354,448	5.0.3502.6144
MSHTML.dll	2,759,952	5.50.4933.1800	2,787,840	6.0.2800.1170
Gcc-a	277,335	3.0.4	299,671	3.1
Gcc-b	299,671	3.1	299,687	3.1.1
Gcc-c	299,687	3.1.1	264,805	3.2
Gcc-d	264,805	3.2	302,897	3.2.1
Sql-a	2,722,338	0.7b3	2,920,666	0.7b4
Sql-b	2,920,666	0.7b4	5,861,140	0.7b5
Sql-c	5,861,140	0.7b5	3,371,146	0.7b6.0
Sql-d	3,371,146	0.7b6.0	2,981,200	0.7b6.1
Samba	5,849,994	1.9.18p10.cat	9,018,170	2.0.0bctal.cat
Random	1,000,000	---	1,000,000	---

Table 4: Test files and their sizes.

For reference, Table 5 presents:

SW: The size of *T* (in bytes) after compressed by IPSW without using *S*.

gzip, *bzip2*: The size of *T* (in bytes) after compressed by the UNIX *gzip* and *bzip2* utilities (without using *S*).

Xdelta, *Zdelta*: The size of *T* (in bytes) after compressed by the UNIX *Xdelta* and *Zdelta* utilities, which compress *T* with respect to *S*.

	SW	***Gzip***	***bzip2***	***Xdelta***	***Zdelta***
Ldap3230.dll	62,678	60,058	57,980	33,823	**24,061**
Agm.dll	320,233	304,186	303,460	136,795	136,794
Msxml.dll	569,556	551,759	512,544	340,881	529,545
Shell32.dll	923,300	899,726	844,834	813,695	907,743
MSHTML.dll	1,575,469	1,567,581	1,452,599	1,527,748	1,569,007
Gcc-a	111,620	111,538	110,700	97,256	104,888
Gcc-b	111,597	111,491	110,611	44,791	**22,182**
Gcc-c	97,867	91,443	87,842	70,682	**67,146**
Gcc-d	112,868	112,866	111,713	95,379	106,451
Sql-a	578,693	666,545	547,856	118,835	101,890
Sql-b	583,011	1,342,795	1,103,992	27,357	687,468
Sql-c	629,447	719,702	583,031	163,024	262,770
Sql-d	594,190	682,383	562,714	14,084	79,557
Samba	1,575,094	1,965,219	1,512,294	1,035,321	1,696,174
Random-20	1,001,682	1,000,181	1,005,296	292	934,903
Random-100	1,001,675	1,000,185	1,004,753	807	964,549

Table 5: Sliding window encoding of IPSW on *T* alone, *gzip* and *bzip* on *T* alone, and compression of *T* with respect to *S* using the UNIX *xdelta* and *zdelta* utilities.

It can be seen from Table 5 that *gzip* typically achieves slightly better compression than SW. As mentioned earlier, the fancier length coding and more crude displacement coding of SW tend to balance each other, and the only way that SW can truly improve upon gzip is when very large matches are present (Case 4 pointers), as is the case in the *Sql* and *Samba* files. The bold numbers in tables 5, 6 and 7 indicate the best compression performance out of the in-place methods. Note that the Burns and Long's compression performance is bounded by Ajtai's compression performance, since they replace copy commands in Ajtai's correcting 1.5-pass delta file that correspond to write before read conflicts with character insertion (add commands).

Table 6 presents five ways of compressing *T* with respect to *S*.

NIP: IPSW with a window of length |*S*|+|*T*| so that at every step, matches can reach any position in *S* or the already compressed portion of *T*.

IPSW: IPSW is the in-place sliding window of Shapira and Storer [2003].

MP+IPSW: MP+IPSW is the compression achieved by preceding IPSW with the Move Preprocessing algorithm of this paper. The last column shows the number of moves and jiggles (moves, jiggles) used by the preprocessing of MP+IPSW.

Agarwal: The compression achieved by Agarwal et al. [2003] differential algorithm.

Ajtai: The compression performance of Ajtai et al [2002] as presented in Agarwal et al. [2003].

	NIP	***Ajtai***	***Agarwal***	***IPSW***	***MP+IPSW***	***#m, #j***
ldap3230.dll	29,044	N/A	N/A	31,952	31,807	4,2
Agm.dll	108,854	N/A	N/A	114,854	**111,801**	6,1
Msxml.dll	273,255	N/A	N/A	501,686	**331,545**	18,3
Shell32.dll	793,837	N/A	N/A	797,965	**795,814**	100, 2
MSHTML.dll	1,407,755	N/A	N/A	1,504,315	**1,444,081**	5,34
Gcc-a	90,417	102,135	97,645	**90,680**	*not needed*	
Gcc-b	34,796	44,428	46,391	57,694	37,085	0,2
Gcc-c	73,879	73,009	70,972	84,826	74,889	9,1
Gcc-d	87,094	99,082	94,300	**88,901**	*not needed*	
Sql-a	90,836	96,325	92,533	**89,087**	*not needed*	
Sql-b	16,676	25,197	27,137	**16,343**	*not needed*	
Sql-c	111,256	123,645	**109,240**	111,265	*not needed*	
Sql-d	8,305	9,037	8,498	**8,111**	*not needed*	
Samba	640,917	765,901	708,873	731,454	**673,732**	1276,1
Random-20	101	N/A	N/A	422,982	**150**	19,0
Random-100	479	N/A	N/A	499,067	**734**	97,0

Table 6: Not in place with a window of size |*S*|+|*T*|, Ajtai et al. [2002], Agarwal et al [2003], IPSW, MP+IPSW, number of moves and jiggles used by MP+IPSW.

Table 7 compares MP+IPSW to compressions which especially deal with delta encoding of executable files, and take advantage of how executable files change

Bsdiff: The compression performance of Percival [2004] which uses suffix sorting of Larsson and Sadakane [1999] and bzip2.

Vcdiff: The compression performance of part of the *Vczip* compressor of Korn and Vo [2002], which uses the LZ77 compressor with a simple Huffman coder.

The NIP sizes are listed in the first column because they are good reference bounds on the compressed size of *T*, although in some cases IPSW (and in one case Ajtai) does slightly better (even though IPSW can be viewed as a restricted version of NIP, small gains may be achieved due to a the differences in the way *S* ends up being parsed). In general it can be seen that the SW, *gzip*, and *bzip2* numbers of Table 5 are worse than the *Xdelta* and *Zdelta* numbers of Table 5, which are in turn worse than all columns of Table 6. However, there are some interesting exceptions that we shall now discuss. The entries N/A are ones where we did not have the data.

	Bsdiff	***Vcdiff***	***IPSW***	***MP+IPSW***
ldap3230.dll	22,271	28,588	31,952	31,807
Agm.dll	51,249	100,356	114,854	111,801
Msxml.dll	183,509	280,182	501,686	331,545
Shell32.dll	805,264	873,527	797,965	795,814
MSHTML.dll	1,339,179	1,584,089	1,504,315	1,444,081
Gcc-a	92,832	90,850	90,680	*not needed*
Gcc-b	7748	27,868	57,694	37,085
Gcc-c	74,128	75,530	84,826	74,889
Gcc-d	89,991	89,352	88,901	*not needed*
Sql-a	92,261	100,574	89,087	*not needed*
Sql-b	23,535	68,591	16,343	*not needed*
Sql-c	128,663	131,550	111,265	*not needed*
Sql-d	10,758	21,477	8,111	*not needed*
Samba	942,877	1.324,104	731,454	673,732
Random-20	315	365,776	422,982	150
Random-100	892	22,556	499,067	734

Table 7: IPSW as compared to *Bsdiff* and *Vcdiff*, which were built specifically for executables and do not work in place.

Many of the MP+IPSW columns in Table 7 are labeled *not needed* because the IPSW compression is already good enough as compared to NIP; That is, the encoder compares IPSW and NIP and performs the preprocessing stage only if there is a big loss of IPSW as opposed to NIP. In fact, it could be argued that a number of other entries in the MP+IPSW column should also be listed as *not needed*. This is one of the key advantages of our preprocessing approach.

IPSW works very well most of the time, and preprocessing can be performed only when it yields significant improvements.

In a number of rows in Table 6 MP+IPSW improves over IPSW (using a relatively small number of moves and jiggles). In one case, Gcc-b, a Table 5 entry, UNIX *Zdelta*, does better than any column in Table 6. However, the 14,903 difference between MP+IPSW and *Zdelta* is only less than 5% of the original *Gcc-b* file size of 299,671 (whereas there are many entries for IPSW and MP+IPSW in Table 6 that improve on corresponding entries in Table 5 by significant factors). The performance on the *Random* files is what one would expect (a large gain for MP+IPSW over just IPSW, a number of moves about the same as the *r* parameter, and no jiggles). In Table 7 the compression performance of IPSW with and without the move preprocessing algorithm compares favorably (sometimes a bit better sometimes a bit worse) to the utilities Bsdiff and Vcdiff that have been built especially for executables and do *not* work in-place.

8. Conclusion

In this paper we study in-place differential file compression. The non in-place version of this problem is known to be NP-Hard, and we 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 we 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). The preprocessing presented in Section 6 can be (optionally) used to improve *IPSW* for inputs with poor alignment. Experiments show that the amount of compression we achieve is comparable to that of the xdelta and zdelta utilities and to that of Factor, Sheinwald and Yassour [2001] (sometimes a bit worse, sometimes a bit better) while working in-place. Further more, an algorithm like gzip could easily be modified to perform efficient encoding and very fast decoding (with possibly slightly better compression performance than we have reported here).

Acknowledgments

The authors wish to thank Francesco Rizzo and Giovanni Motta for their help with the implementation of the algorithms presented.

References

Shapira, D. and Storer, J. A. (2004b), “Edit Distance with Multiple Block Operations”, Technical Report, Brandeis University.

Storer, J. A. (2001). An Introduction to Data Structures and Algorithms, Birkhauser/Springer.

Tichy, W. F. (1984). "The String to String Correction Problem with Block Moves", *ACM Transactions on Computer Systems*, 2(4), 309–321

Weiner, P. (1973). "Linear Pattern Matching Algorithms", Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory (FOCS), 1–11.

Burns, R. C. and Long, D. D. E. (1997a). "Efficient Distributed Backup and Restore with Delta Compression", Workshop on I/O in Parallel and Distributed Systems (IOPADS), ACM.

Storer, J. A. and Szymanski, T. G. (1978). "The Macro Model for Data Compression", *Proc. 10th Annual ACM Symp. Theory of Computing*, San Diego, CA, 30-39.

Storer, J. A. and Szymanski, T. G. (1982). "Data Compression Via Textual Substitution", *JACM* 29:4, 928-951.

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

Ergun, F. Muthukrishnan, S. and Sahinalp S. C. (2003). "Comparing Sequences with Segment Rearrangements", DIMACS technical report.

Bentley, J. and Mcilroy, D. (1999). "Data Compression Using Long Strings", *Proceedings Data Compression Conference*, IEEE Computer Society Press, 287–295.

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

Burns R. C. and Long, D. D. E. (1997b). "A Linear Time, Constant Space Differencing Algorithm", *Proceedings International Performance, Computing and Communications Conference* (IPCCC), IEEE.

Agarwal, R. C. Amalapurapu, S. and Jain, S. (2003). "An Approximation to the Greedy Algorithm for Differential Compression of Very Large Files", Tech. Report, IBM Alamaden Res. Center.

Heckel, P. (1978). "A Technique for Isolating Differences Between Files", *CACM* 21:4, 264–268.

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

Ziv, J. and Lempel, A. (1977), A Universal Algorithm for sequential data compression, *IEEE Trans. on Information Theory*, (1977) 337–343.

Ziv, J. and Lempel, A. (1978), Compression of individual sequence via variable rate coding, *IEEE Trans. on Information Theory*, (1978) 530–536.

Burns, R. C. and Long, D. D. E. (1998). "In-place Reconstruction of Delta Compressed Files", Proceedings of the ACM Conference on the Principles of Distributed Computing, ACM.

Burns, R. C. Stockmeyer, L. and Long, D. D. E. (2003). "In-Place Reconstruction of Version Differences”, *IEEE Transactions on Knowledge and Data Engineering* 15:4, 973–984.

Shapira, D. and Storer, J. A. (2003). "In-Place Differential File Compression", *Proceedings Data Compression Conference*, IEEE Computer Society Press, 263–272.

Shapira, D. and Storer, J. A. (2004a), “In-place differential file compression of non-aligned files with applications to file distribution and backups”, *Proceedings Data Compression Conference, DCC-2003*, Snowbird, Utah (2004) 82--91.

Rosen, K. H. (2000), Elementary Number Theory. *Addison-Wesley*.

Shapira, D. and Storer, J. A. (2002). "Edit Distance with Move Operations", *Proceedings Conference of Combinatorial Pattern Matching* (CPM), Springer, 85–98.

Lopresti, D. and Tompkins, A. (1997). "Block Edit Models for Approximate String Matching", *Theoretical Computer Science* 181, 159–179.

Delcher, A. L. Kasif, S. Fleischmann, R. D. Peterson, J White, O and Salzberg, S. L. (1999), “Alignment of whole genomes”, *Nucl. Acids. Res*, 27,11, 2369-2376

Hon, W. K. and Sadakane, K. (2002), "Space-Economical Algorithms for Finding Maximal Unique Matches", *Proceedings 13th Annual Symposium on Combinatorial Pattern Matching (CPM)*, 145- 152.

Apostolico, A. and Preparata, F. (1985). "Structural Properties of the String Statistics Problem". *Journal of Computer and System Sciences* 31, 394–411.

Ukkonen, E. (1995). "On-line Construction of Suffix Trees", *Algorithmica* 14, 249–260.

Mccreight, E. M. (1976). "A Space-Economical Suffix Tree Construction Algorithm", *Journal of Algorithms* 23, 262–272.

Bafna, V. and Pevzner, P. A. (1998), Sorting by Transpositions, *SIAM Journal of Discrete mathematics* 11:2, 124–240.

Percival, C.(2004), Naïve Differences of Executable Code, http://www.daemonology.net/bsdiff/

Larsson, J. and Sadakane, K. (1999). “Faster Suffix Sorting”, Technical report 99-214, Lund University, Sweden.

Korn, D. G. and Vo, K.P. (2002), The VCDIFF generic Differencing and Compression Data Format, *Technical Report.*