1983

The String-to-String Correction Problem with Block Moves

Walter F. Tichy

Report Number:

83-459

The String-to-String Correction Problem with Block Moves

Walter F. Tichy

Purdue University
Department of Computer Science
West Lafayette, IN 47907

CSD-TR 459

ABSTRACT

The string-to-string correction problem is to find a minimal sequence of edit operations for changing a given string into another given string. Extant algorithms compute a Longest Common Subsequence (LCS) of the two strings and then regard the characters not included in the LCS as the differences. However, an LCS does not necessarily include all possible matches, and therefore does not produce the shortest edit sequence.

We present an algorithm which produces the shortest edit sequence transforming one string into another. The algorithm is optimal in the sense that it generates a minimal, covering set of common substrings of one string with respect to the other.

Two runtime improvements of the basic algorithm are also presented. Runtime and space requirements of the improved algorithms are comparable to LCS algorithms.

Categories and Subject Descriptors: D.2.2 **[Software Engineering]**: Tools and Techniques—*programmer workbench, software libraries*; D.2.6 **[Software Engineering]**: Programming Environments; D.2.7 **[Software Engineering]**: Distribution and Maintenance— *version control*

General Terms: Algorithms

Additional Key Words and Phrases: String-to-string correction, block moves, deltas, differences, source control, revision control

October 26, 1983

The String-to-String Correction Problem with Block Moves

Walter F. Tichy

Purdue University
Department of Computer Science
West Lafayette, IN 47907

CSD-TR 459

Introduction

The string-to-string correction problem is to find a minimal sequence of edit operations for changing a given string into another given string. The length of the edit sequence is a measure of the differences between the two strings. Programs for determining differences in this manner are useful in the following situations.

(1) Difference programs help determine how versions of text files differ. For instance, computing the differences between revisions of a software module helps a programmer trace the evolution of the module during maintenance[6], or helps create test cases for exercising changed portions of the module. Another application is the automatic generation of change bars for new editions of manuals and other documents.

(2) Frequently revised documents like programs and graphics are stored most economically as a set of differences relative to a base version[10, 12]. Since the changes are usually small and typically occupy less that 10% of the space needed for a complete copy[10], difference techniques can store the equivalent of about 11 revisions in less space than would be required for saving 2 revisions (one original and one backup copy) in cleartext.

(3) Changes to programs and other data are most economically distributed as "update decks" or "deltas", which are edit sequences that transform the old version of a data object into the new one. This approach is used in software distribution. A related application can be found in screen editors and

This work was supported in part by the National Science Foundation under grant MCS-8109513.

graphics packages. These programs update display screens efficiently by computing the difference between the old and new screen contents, and then transmitting only the changes to the display[2].

(4) In genetics, difference algorithms compare long molecules consisting of nucleotides or amino acids. The differences provide a measure of the relationship between types of organisms[11].

Most of the existing programs for computing differences are based on algorithms that determine a Longest Common Subsequence (LCS). An LCS has a simple and elegant definition, and algorithms for computing an LCS have received some attention in the literature[13, 4, 6, 7, 5, 9]. An LCS of two strings is one of the longest subsequences that can be obtained by deleting zero or more symbols from each of the two given strings. For example, the longest common subsequence of *shanghai* and *sakhalin* is *sahai*. Once an LCS has been obtained, all symbols that are not included in it are considered differences. A simultaneous scan of the two strings and the LCS isolates those symbols quickly. For example, the following edit script, based on the LCS *sahai*, would construct the target string *sakhalin* from *shanghai*.

```
M 0,1
M 2,1
A "k"
M 5,2
A "l"
M 7,1
A "n"
```

An edit-command of the form $M\ p,l$, called a *move*, appends the substring $S[p, \ldots, p+l-1]$ of source string S to the target string, and an *add* command of the form $A\ w$ appends the string w to the target string. In the above example, the edit script takes up much more space than the target string, and none of the savings mentioned earlier are realized. In practical cases, however, the common subsequence is not as fragmented, and a single *move* command covers a long substring. In addition, if this technique is applied to text, one usually chooses full text lines rather than single characters as the atomic symbols. Consequently, the storage space required for a *move* is negligible compared to the that of an *add* command, and it is worth minimizing the occurrence of the *add* commands. Note that in the above example, the last *add* command could be replaced with a *move*, since the symbol n appears in both strings.

Unfortunately, the definition of an LCS is such that the n cannot be included in the LCS. The algorithm presented below does not omit such matches.

Problem Statement

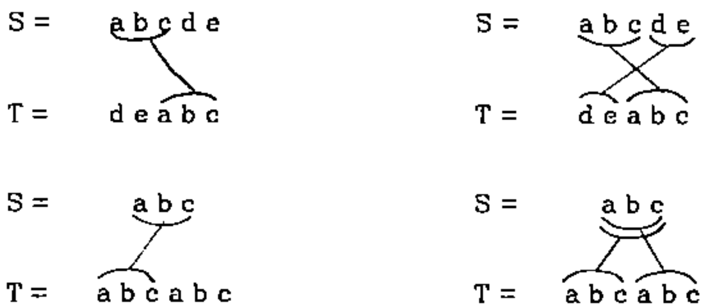
Given 2 strings $S=S[0, \ldots, n]$, $n \geq 0$ and $T=T[0, \ldots, m]$, $m \geq 0$, a *block move* is a triple (p, q, l) such that $S[p, \ldots, p+l-1] = T[q, \ldots, q+l-1]$ ($0 \leq p \leq n-l+1$, $0 \leq q \leq m-l+1$, $l > 0$). Thus, a block move represents a non-empty, common substring of S and T with length l, starting at position p in S and position q in T. A *covering set of T with respect to S*, denoted by $\delta_S(T)$, is a set of block moves, such that every symbol $T[i]$ that also appears in S is included in exactly one block move. For example, a covering set of T=*abcab* with respect to S=*abda* is $\{(0,0,2),(0,3,2)\}$. A trivial covering set consists of block moves of length 1, one for each symbol $T[i]$ that appears in S.

The problem is to find a *minimal* covering set, $\Delta_S(T)$, such that $|\Delta_S(T)| \leq |\delta_S(T)|$ for all covering sets $\delta_S(T)$. The coverage property of $\Delta_S(T)$ assures that all possible matches are included, and the minimality constraint makes the set of block moves (and therefore the edit script) as small as possible.

Because of the coverage property, it is apparent that $\Delta_S(T)$ includes the LCS of S and T. (Consider the concatenation of the substrings $T[q_j, \ldots, q_j+l_j-1]$, where (p_j, q_j, l_j) is a block move of $\Delta_S(T)$, and the substrings are concatenated in order of increasing q_j.) The minimality constraint assures that the LCS cannot provide a better "parcelling" of the block moves.

False Starts

Before presenting the solution, it is useful to consider several more or less obvious approaches, all of which fail. The first approach is to use the LCS. As we have seen, an LCS has the property of not necessarily generating a covering set of block moves. For example, the following two pairs of strings have the LCS *abc*, which does not include the (moved) common substring *de* nor the (repeated) common substring *abc*. The LCS match is shown on the left, $\Delta_S(T)$ on the right.



Heckel[3] pointed out similar problems with LCS techniques and proposed a linear-time algorithm to detect block moves. The algorithm performs adequately if there are few duplicate symbols in the strings. However, the algorithm gives poor results otherwise. For example, given the two strings *aabb* and *bbaa*, Heckel's algorithm fails to discover any common substring.

An improvement of the LCS approach is to apply the LCS extraction iteratively. For instance, after finding the initial LCS in the above examples, one could remove it from the target string T and recompute the LCS. This process is repeated until only an LCS of length 0 remains. The iterative LCS strategy succeeds in finding a covering set, but not necessarily the minimal one. The following example illustrates.

Assuming again that S is the source string and T is the target string, the left diagram shows the match obtained via an iterative LCS algorithm. The first LCS is *cda*, the second one is *b*. Since *cda* is not a substring of S, we obtain a total of 3 block moves. The minimal covering set, shown to the right, consists of 2 block moves.

Another tack is to search for the longest common sub*string* rather than the longest common sub*sequence**. Computing the longest common substring iteratively results in a covering set, but again not necessarily a minimal one. Con-

* Recall that a subsequence may have gaps, a substring may not.

sider the following example.

S = a b c d e f d e a b   S = a b c d e f d e a b

T = c d e a b c   T = c d e a b c

The left diagram shows the block moves obtained by searching repeatedly for the longest common substring of S and T. The result is a set of 3 block moves, although 2 are minimal. Searching for the longest common substring is too "greedy" a method, since it may mask better matches.

Basic Algorithm

A surprisingly simple algorithm does the job. Start at the left end of the target string T, and try to find prefixes of T in S. If no prefix of T occurs in S, remove the first symbol from T and start over. If there are prefixes, choose the longest one and record it as a block move. Then remove the matched prefix from T and try to match a longest prefix of the remaining tail of T, again starting at the beginning of S. This process continues until T is exhausted. The recorded block moves constitute a $\Delta_S(T)$, a minimal covering set of block moves of T with respect to S, as will be shown later. The following example illustrates several steps in the execution of the algorithm. The string to the right of the vertical bar is the unprocessed tail of T.

Step 1:

S = u v w u v w x y

T = | z u v w x w u   longest block move starting with T[0]: none

Step 2:

S = u v w u v w x y

T = z| u v w x w u   longest block move starting with T[1]: (3,1,4)

Step 3:

S = u v w u v w x y

T = z u v w x| w u   longest block move starting with T[5]: (2,5,2)

In step 1, we search for a prefix of $T[0, \ldots, 6]$ in $S[0, \ldots, 7]$. Since there is none, we search for a prefix of $T[1, \ldots, 6]$ in the next step. This time we find 2 matches, and choose the longer one, starting with $S[4]$. In step 3, we search for a prefix of $T[5, \ldots, 6]$ in $S[0, \ldots, 7]$, and find the longest one at $S[2]$, length 2. Now T is exhausted and the algorithm stops. Note that in each step we start at the left end of S in order to consider all possible matches.

The algorithm is presented below. Let us assume that the source string is stored in an array $S[0, \ldots, m]$, and the target string in $T[0, \ldots, n]$. $T[q]$ is the first symbol of the unmatched tail of T; q is initially zero. The first refinement of the algorithm is now as follows.

```
q := 0;
while q<=n do
begin
        L: find p and l such that (p,q,l) is a maximal block move
        if l>0 then print(p,q,l);
        q := q+Max(1,l)
end
```

Implementing the statement labelled L is simple. Search S from left to right for a longest possible prefix of $T[q, \ldots, n]$. Note that the search can terminate as soon as there are fewer than $l+1$ symbols left in S, assuming that l is the length of the maximal block move found in the current iteration. Similarly, there is no possibility of finding a longer block move if the last one included $T[n]$. (We use **and then** as the conditional logical AND operator.)

```
L:
        l := 0; p := 0; pCur := 0;
        while pCur+l <= m and q+l <= n do
        begin { Determine length of match between S[pCur,...] and T[q,...] }
                lCur := 0;
                while (pCur+lCur <= m) and (q+lCur<=n)
                    and then (S[pCur+lCur] = T[q+lCur])
                    do lCur := lCur+1;
                if lCur > l then
                begin { new maximum found }
                    l := lCur; p := pCur
                end;
                pCur := pCur+1
        end
```

The runtime of this algorithm is bounded by mn, and the space requirements are $m+n$. We now show that this algorithm finds a $\Delta_S(T)$. Clearly, the set of block moves printed is a covering set, because each symbol in T that is not included in some block move is (unsuccessfully) matched against each symbol in S. To see that the covering set is minimal, consider T below, with the matching produced by our algorithm denoted as follows. Substrings included in a block move are bracketed by "(" and ")". Substrings of symbols excluded from any block move are denoted by X.

$$\cdots X(\cdots)X(\cdots)(\cdots)X(\cdots)(\cdots)(\cdots)X\cdots$$

Suppose there is a $\delta'_S(T)$ with fewer block moves than the set generated by our algorithm. Clearly, the substrings denoted by X cannot be part of $\delta'_S(T)$, because our algorithm does produce a covering set. We can therefore exclude all unmatched substrings from consideration, and concentrate on individual sequences of contiguous block moves.

Now consider block moves that are contiguous in T. The only way to obtain a smaller covering set is to find a sequence of $k>1$ contiguous block moves and to "reparcel" them into a covering set of fewer moves. We will show by induction on the number of contiguous block moves that the set produced by our algorithm is minimal.

Suppose we have $k \geq 1$ contiguous block moves generated by our algorithm. This means that we have k triples (p_i, q_i, l_i), $(1 \leq i \leq k)$ satisfying the following conditions.

$$\mathrm{A}i{:}1 \leq i \leq k \;\; T[q_i, \ldots, q_i+l_i-1] = S[p_i, \ldots, p_i+l_i-1] \qquad (*)$$

$$\mathrm{A}i{:}1 \leq i \leq k,\ \mathrm{A}p{:}0 \leq p \leq m-l_i,\ T[q_i, \ldots, q_i+l_i] \neq S[p, \ldots, p+l_i] \qquad (**)$$

$$\mathrm{A}i{:}1 \leq i < k \;\; T[q_i+l_i] = T[q_{i+1}] \qquad (***)$$

The first condition is just the definition of a block move. The second condition assures that each block move starting at $T[q_i]$ is maximal. The third condition means that the block moves are contiguous in T.

We need to show that for any set of of k block moves satisfying (*) to (***), any equivalent set has at least k block moves. Actually, it is convenient to prove something slightly more general: For any set of k block moves satisfying (*) to (***). Any set which covers the first $k-1$ bock moves and a non-empty prefix of block move k has at least k block moves. First, assume $k=1$. Clearly, we cannot split any non-empty prefix of a single block move into less than 1 covering block move. Now assume that $k>1$, and that all sets covering the first $k-2$ block moves and any non-empty prefix of block move $k-1$ consist of at least $k-1$ block moves. Consider what we can do with non-empty prefixes of the k'th block move. There are two cases. The first case applies to sets that cover the original block move $k-1$ with a single move B. In this case, let $B = (p_b, q_b, l_b)$, where $p_b \leq p_{k-1}$, and $p_b + l_b = p_{k-1} + l_{k-1}$. By the induction hypothesis, B is at least the $k-1st$ move in the equivalent set. It is impossible to append a non-empty prefix of move k to B since that would contradict (**). Thus we need at least k moves for covering the original $k-1$ moves and a non-empty prefix of original move k. The second case applies to sets that split the original block move $k-1$ into at least 2 non-empty moves (see the diagram below).

orig. block move no.	k-2	k-1	k
orig. set	......)	(... ...)	(... ...)
$\delta'_S(T)$ covering k-1	......	...) (...)	
$\delta''_S(T)$ covering k	......	...) (...	...) (...)

The only choice to reduce the number of block moves below k is to coalesce the suffix of the original move $k-1$ with a non-empty prefix of move k. This new parcelling leaves us with (a) a set covering the original $k-2$ block moves and a non-empty prefix of block move $k-1$, (b) a new coalesced move covering a suffix of move $k-1$ and a prefix of k, and (c) another block move if the suffix of move k is not empty. By the induction hypothesis, we know that (a) has at least $k-1$ moves. Add to that the (non-empty) coalesced move, and we end up with at least k moves for covering the first $k-1$ block moves and any non-empty prefix of move k. Thus, any set equivalent to the block moves generated by our algorithm has at least k elements. QED.

First Improvement of the Basic Algorithm

Consider a situation where the source string S has few replicated symbols. That is, α, the size of the alphabet of S, is approximately equal to m. In this case, a significant improvement of the basic algorithm is possible. During a single scan of S, we prepare an index that, for each symbol s in the alphabet, lists the positions of all occurrences of s in S. In the basic algorithm, we replace the statement labelled F with the following. Assume $T[q] = s$ is the first symbol of the unmatched tail of T. Look up the list L of occurrences of symbol s in S, using the above index. If the list is empty, no match is possible. Otherwise, find the maximal block move among those starting with the elements of L in S.

The performance of this algorithm is as follows. Assume the average length of a block move is l. Then the maximal block move must be selected among m/α alternatives, at a cost of not more than $l+1$ comparisons each. Thus, the runtime of the algorithm is $O(l*(m/alpha)*(n/l)) = O(mn/\alpha)$. If $m \approx \alpha$, we obtain a nearly linear algorithm.

Program text and prose have the property of few repeated lines. In program text, the only repeated lines should be empty or consist of bracketing symbols like **begin** and **end**; for all other repetitions one would normally write a subprogram. In prose text, the only repeated lines should be empty or contain formatting commands. In applying our algorithm to prose or program text, it is therefore appropriate to choose lines as the atomic symbols. To speed up comparisons, the program should use hashcodes for lines of text rather than performing character-by-character comparisons.

We implemented a program incorporating these ideas, called ***bdiff***, and compared it with ***diff***[6], which uses an LCS algorithm. We executed both programs on 1400 pairs of files. Each pair consisted of 2 successive revisions of text, deposited in a data bases maintained by the Revision Control System[12]. This system stores multiple revisions of text files as differences. Almost all of the sample files contained program text. We observed that ***diff*** and ***bdiff*** execute with similar speeds, but that ***bdiff*** produces deltas that are, on the average, only about 7% smaller. Apparently, block moves and duplicate lines in program text are not frequent enough to obtain significant space savings over LCS algorithms. We expect that the situation is more advantageous for block moves in the other applications mentioned in the introduction.

Second Improvement of the Basic Algorithm

A different improvement speeds up our basic algorithm even if the source string contains numerous duplicated symbols. The improvement involves an adaptation of the Knuth-Morris-Pratt string matching algorithm[8], which allows a pattern of length l to be found in a string of length m in $O(m+l)$ steps. Thus, if S is of length m, T is of length n, and the average block move is of length l, our algorithm should operate in $O((m+l)*(n/l)) = O(mn/l)$ steps. Note that the ratio m/l is a measure of the "difference" of S and T, and that the runtime of the algorithm is proportional to that ratio. Note also that this measure is independent of the permutation of the common substrings in T with respect to S.

An important element in the Knuth-Morris-Pratt algorithm is an auxiliary array N which indicates how far to shift a partially matched pattern or block move after a mismatch. The array N is as long as the pattern, and is precomputed before the match. Precomputing N poses a problem for our algorithm. Since we do not know how long a block move is going to be, we would have to precompute N for the entire unprocessed tail of T, although we would normally use only a small portion of it. Fortunately, N can also be computed incrementally. The outline of the adapted pattern matching algorithm is as follows.

Assume the next unmatched symbol is $T[q]$. Start by initializing $N[q]$ and apply the Knuth-Morris-Pratt algorithm to find the first occurrence of $T[q]$. (Note that this is a pattern of length 1.) If this pattern cannot be found, there is no block move including $T[q]$. Otherwise, expand the pattern by 1, compute the next entry in N, and reapply the Knuth-Morris-Pratt algorithm to find the first occurrence of the expanded pattern. Start the search with the previous match. Continue this process, until the pattern reaches a length for which there is no match. At that point, the previous match is the maximal block move.

Suppose the maximal common block move starting with $T[q]$ is l. The last attempted pattern match is therefore of length $l+1$, and fails. The incremental computation of the entries $N[q, \ldots, q+l+1]$ at a total cost proportional to l assures that the cost of the average match remains $O(m+l)$.

The detailed program is given in the appendix. It is useful for applications (3) and (4) mentioned in the introduction. The idea of incrementally computing auxiliary data structures can also be applied to the Boyer-Moore pattern matching algorithm[1], resulting in a program that runs even faster on the average.

Reconstructing the Target String

An edit script that reconstructs target string T from source string S is a sequence of *move* and *add* commands. The commands build a string T' left-to-right. Each block move (p,q,l) in $\Delta_S(T)$ is represented by a command of the form $M\ p,l$, which copies the string $S[p, \ldots, p+l-1]$ to the end of the string T'. For any substring $T[u, \ldots, v]$ consisting entirely of symbols that do not occur in S, the edit script contains the command $A\ T[u,...,v]$, which simply appends the unmatchable substring to T. After completion of all edit commands, $T = T$.

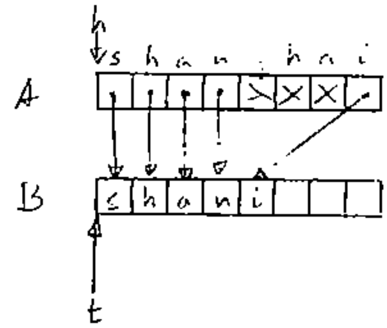
In general, T cannot be constructed in a single pass over S, because block moves may cross (cf. examples in Sect. 3). If S is a sequential file, one can minimize the number of rewind operations caused by crossing block moves as follows. During the generation of the edit script, it does not matter which one of 2 or more equivalent block moves is chosen. For example, suppose we have the following equivalent, maximal block moves starting with $T[q]$: $B1 = (p_1, q, l)$ and $B2 = (p_2, q, l)$, with $p_1 < p_2$. If the previous block move emitted had its S-endpoint between $S[p_1]$ and $S[p_2]$, choosing the block move $B2$ saves one rewind operation for S. Our algorithms are easily modified to accommodate this idea. Rather than starting at the left end of S while searching for the longest possible match, they must start with the endpoint of the previous match and "wrap around" at the end of S.

So far, we have presented our edit scripts as constructing T separately from S. It is also possible to transform S "in place". The following paragraphs discusses the algorithm in some detail.

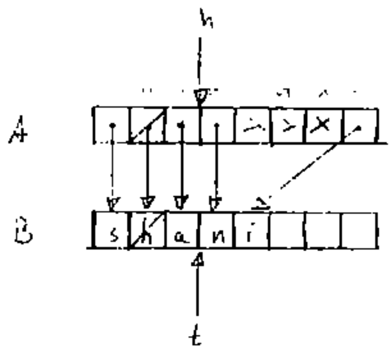
Suppose we have a buffer $B[0, \ldots, Max(m,n)]$ initialized to S, i.e., $B[i] = S[i]$ for $0 \le i \le n$. The goal is to transform the contents of B to T. The key to this algorithm is an auxiliary array $A[0, \ldots, n]$, which keeps track of the positions of the original symbols $S[i]$ in B. Initially, $A[i] = i$ for $0 \le i \le n$. A marker h moves through A from left to right, giving the index of the rightmost symbol involved in a block move so far. Thus, for the k'th *move* command $M\ p_k, l_k$, $h = Max(p_j + l_j, 0 \le j \le k)$. There is also a marker t indicating the index of the last symbol processed in B.

The first step is to remove all symbols from B which are not in T. This step preprocesses the edit script to isolate the symbols to be deleted, and then actually removes them from B. It also updates the mapping array A to reflect the compression, and marks those entries of A as undefined whose counterparts in B were deleted. The second step processes the edit commands in sequence. An *add* command simply inserts the given string to the right of t, and resets t to point to the last symbol so inserted. It also updates the array A for the symbols shifted right by the insertion. For each *move* of the form $M\ p, l$, compare p and the current value of h. If $p > h$, then the current block move is to the right of the previous one. The symbols between h and p, i.e., $B[A[h+1], \ldots, A[p-1]]$, are not included in the current move, but will be moved later. Mark them as such and set h to $p+l-1$ and t to $A[h]$. Thus, the characters $S[p, \ldots, p+l-1]$ will be included in the result. Otherwise, if $p \le h$, the current block move crosses the previous one, and a substring located before t must be moved or copied forward. All symbols in that string that were marked for moving by an earlier command are now moved, the others are simply copied forward. It is conceivable that the the current block move involves symbols to the left and right of h. In that case, first handle the string to the left of h by moving or copying elements of the string $B[A[p], \ldots, A[Min(p+l-1,h)]]$ after $B[t]$. The remaining (possibly empty) string $A[h+1, \ldots, p+l-1]$ is simply included by setting h to $Max(p+l-1,h)$. Update A to reflect the moves and shifts, and set t to $A[h]$.

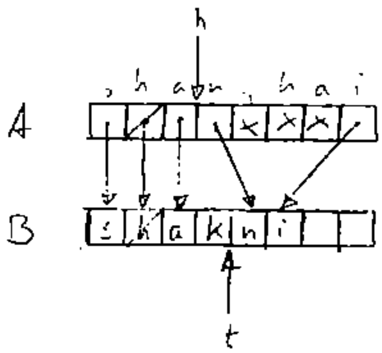
Below is a trace of the algorithm, transforming the string *shanghai* to *sakhalin* by applying the edit script $M0,1$; $M2,1$; A"k"; $M1,2$; A"l"; $M7,1$; $M3,1$. The algorithm can be applied to update display screens efficiently, provided the display offers operations for character and line insertion and deletion, as well as a *copy/move* feature. The latter feature is needed for copying and moving character strings forward in the above algorithm. The auxiliary array A is allocated in main memory.



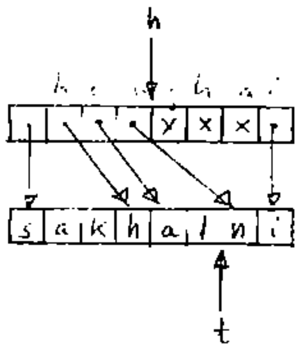
After removing unused symbols



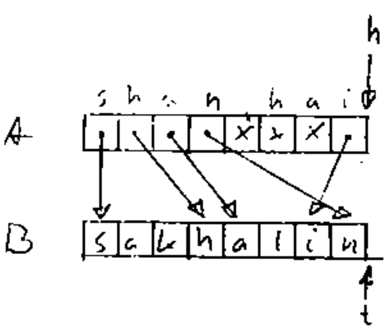
After applying
M 0,1; M 2,1
A[1] is marked for move.



After applying
I "k"



After applying
M 1,2; I "l"



After applying
M 7,1; M 3,1

Conclusions

The original string-to-string correction problem as formulated in[13] permitted the editing commands *add*, *delete*, and *change*. Clearly, a *change* command can be simulated with a *delete* followed by an *add*. Any sequence of *add* and *delete* commands can be transformed into an equivalent sequence of *add* and *move* commands. This transformation works since *delete* and *move* commands complement each other, provided no block moves cross or overlap. Our approach of extending the editing commands by permitting crossing block moves results in shorter edit sequences. We developed efficient algorithms for computing those sequences. Reconstructing the target string by applying the edit sequence is efficient if the source string can be accessed randomly.

Appendix: Using the Knuth-Morris-Pratt Pattern Matching Algorithm.

```
S: array[0..m] of symbol;
T: array[0..n] of symbol;
N: array[0..n] of symbol;

q := 0; { start at left end of T }
while q <= n do
begin { Characters left in T; find longest match starting with T[q] }
        k := 0; { start match at left end of S }
        j := q; { first symbol of pattern }
        last := q; { last symbol of pattern }
        N[q] := q-1; { initialize N[q] }
        iN := q-1; { initialize computation of N[q+1,...] }

        loop   { loop with exit from the middle }
               { try to find a match for T[q]..T[last]   }
               { T[q]..T[last-1] has already been matched }

               kOld := k; { save last point of old match, if any }

               while (j<=last) and (k<=m) do
               begin
                   while (j>=q) and (S[k] <> T[j])
                         do j := N[j];
                   k := k+1; j := j+1;
               end

        until (j<=last) || (last=n);  { exit from the middle }

               { found match; now increase last and compute N[last] }
               while (iN>=q) and (T[last] <> T[iN])
                      do iN := N[iN];
               last := last+1; iN := iN+1;
               if T[last]=T[iN]
                      then N[last] := N[iN]
                      else N[last] := iN;
        end { end of loop }

        { print match }
        if j>last then
        begin { found match for tail of T }
               print(k-(n-q+1), q, n-q+1);
               q := n+1;
        end else if q = last then
        begin { no match }
               q := q+1;
        end else
        begin { last match failed; take previous one }
               print(kOld-(last-q), q, last-q)
               q := last;
        end
end
```

References

1. BOYER, ROBERT S. AND MOORE, J. STROTHER, "A Fast String Searching Algorithm," *Communications of the ACM* **20**(10) p. 762-772 (October 1977).
2. GOSLING, JAMES, "A Redisplay Algorithm," *Proc. ACM SIGPLAN SIGOA Symposium on Text Manipulation*, p. 123-129 (June 1981).
3. HECKEL, PAUL, "A Technique for Isolating Differences Between Files," *Communications of the ACM* **21**(4) p. 264-268 (April 1978).
4. HIRSCHBERG, DANIEL S., "A Linear Space Algorithm for Computing Maximal Common Subsequences," *Communications of the ACM* **18**(6) p. 341-343 (June 1975).
5. HIRSCHBERG, DANIEL S., "Algorithms for the Longest Common Susbsequence Problem," *Journal of the ACM* **24**(4) p. 664-675 (October 1977).
6. HUNT, JAMES W. AND MCILROY, M. D., "An Algorithm for Differential File Comparison," Computing Science Technical Report No. 41, Bell Laboratories (June 1976).
7. HUNT, JAMES W. AND SZYMANSKI, THOMAS G., "A Fast Algorithm for Computing Longest Common Subsequences," *Communications of the ACM* **20**(5) p. 350-353 (May 1977).
8. KNUTH, DONALD E., MORRIS, JAMES H., AND PRATT, VAUGHAN R., "Fast Pattern Matching in Strings," *SIAM Journal of Computing* **6**(2) p. 323-350 (June 1977).
9. NAKATSU, NARAO, KAMBAYASHI, YAHIKO, AND YAJIMA, SHUZO, "A Longest Common Subsequence Algorithm for Similar Text Strings," *Acta Informatica* **18** p. 171-179 (1982).
10. ROCHKIND, MARC J., "The Source Code Control System," *IEEE Transactions on Software Engineering* **SE-1**(4) p. 364-370 (Dec. 1975).
11. SANKOFF, DAVID, "Matching Sequences under Deletion/Insertion Constraints," *Proc. Nat. Academy of Sciences, USA* **69**(1) p. 4-6 (January 1972).
12. TICHY, WALTER F., "Design, Implementation, and Evaluation of a Revision Control System," pp. 58-67 in *Proceedings of the 6th International Conference on Software Engineering*, IPS, ACM, IEEE, NBS (September 1982).
13. WAGNER, ROBERT A. AND FISHER, MICHAEL J., "The String-to-String Correction Problem," *Journal of the ACM* **21**(1) p. 168-173 (January 1974).