A redisplay algorithm

Article *in* ACM SIGPLAN Notices · April 1981

DOI: 10.1145/872730.806463

CITATIONS
13

READS
146

1 author:

James Gosling
Amazon
69 PUBLICATIONS **9,817** CITATIONS

SEE PROFILE

A Redisplay Algorithm

James Gosling
Carnegie-Mellon University

Abstract

This paper presents an algorithm for updating the image displayed on a conventional video terminal. It assumes that the terminal is capable of doing the usual insert/delete line and insert/delete character operations. It takes as input a description of the image currently on the screen and a description of the new image desired and produces a series of operations to do the desired transformation in a near-optimal manner. The algorithm is interesting because it applies results from the theoretical string-to-string correction problem (a generalization of the problem of finding a longest common subsequence), to a problem that is usually approached with crude ad-hoc techniques.

1. Introduction

Redisplay algorithms are an important part of many modern video editors. It is the responsibility of the redisplay to maintain the correspondence between the image on the screen and the text being edited. When a change is made to the text, the image on the screen must be updated to reflect the fact. An example of such an editor is *Emacs* [8].

Communication bandwidth limitations make it necessary to attempt to transmit to the screen only information about changes that have been made. The process is complicated when the insert/delete line and insert/delete character operations available on many commercial video terminals are used. These operations allow lines of text to be moved up and down on the screen by deleting lines or inserting blank lines; and they allow text to be moved left or right on a line by inserting and deleting individual characters.

Redisplay algorithms can be grouped into two categories: those that intermix the display update with the data base update, and those that separate them.

The first approach interweaves display changes with data base changes. Any time a change is made to the data base that change is reflected immediatly on the screen. This approach is easy to implement since there is usually a close correspondence between changes on the screen and in the data base. For example if the data base is a text file then inserting a line into the data base should cause a line to be inserted on the screen. But this approach has disadvantages: complicated compound operations can cause unnecessary and confusing manipulation of the screen; when the data base does not correspond closely to the image on the screen the technique breaks down completely (as in a structure editor); and this interweaving of display and data base code can make the program difficult to debug and modify and is generally poor programming practice. An example of this approach can be found in [3].

The second approach separates display and data base changes. The data base is changed without considering the effects on the display, an update procedure is called periodically to analyze the new data base and update the display. The advantages of this approach parallel the disadvantages of the first: compound operations are handled gracefully and the separation of the data base and the display yield a more reliable, maintainable and clean program. The principle disadvantage is poor performance, which can nearly be eliminated by using good algorithms and a good implementation.

Most examples of this second approach use straightforward, unsophisticated algorithms [4]. The algorithm presented in this paper employs a sophisticated but simple algorithm which is based on an algorithm for the string-to-string correction problem.

2. The String-to-String Correction Problem

In [9] the *string-to-string correction problem* along with an $O(n^2)$ dynamic programming solution is presented. This problem is concerned with determining the *edit distance* between two strings, which is defined as the shortest sequence of *edit operations* needed to transform one string to another. An edit operation is the insertion, deletion, or alteration of an element of a string. The intended applications of the solution were in automatic spelling correction, and in the solution of the longest common subsequence problem [5, 1]. Faster algorithms exist, but they are more complicated and their speed advantages are only realized for large problem sizes [6, 7].

Wagner and Fischer[1] define a *trace* to be a description of how an edit sequence S transforms string A into string B, ignoring order and redundancy in S. For example:

[1] The text of much of this section borrows heavily from the presentation in [9]. They said it well and I doubt that I could improve on it.

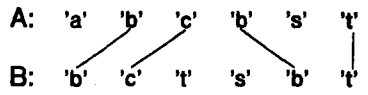

Figure 2-1: An example of a trace

A line in this diagram joining element i of A to element j of B means that B_j is derived from A_i, either directly if $A_i = B_j$ or indirectly otherwise. Certain elements of A are untouched by lines; these elements are deleted by the transformation. Certain elements of B are untouched by lines; these elements are inserted by the transformation. It is important to note that no two lines ever cross.

Formally, they define a *trace from A to B* as a triple (T,A,B) where A and B are strings and T is a set of ordered pairs satisfying:

1. $1 \le i \le |A|$ and $1 \le j \le |B|$
2. for any two distinct pairs (i_1,j_1) and (i_2,j_2) in T:
   a. $i_1 \ne i_2$ and $j_1 \ne j_2$
   b. $i_1 < i_2$ iff $j_1 < j_2$

Ordered pairs in the trace correspond to lines in the diagram. $(i,j) \in T \Rightarrow A_i$ gets transformed to B_j.

Three cost functions are used:

$C_t(A_i,B_j)$ is the cost of transforming A_i to B_j.

$C_i(B_j)$ is the cost of inserting B_j.

$C_d(A_i)$ is the cost of deleting A_i.

Let T be a trace from A to B. Let I and J be the sets of positions in A and B respectively not touched by any line in T. The total cost of applying T is then defined as:

$$C(T) = \sum_{(i,j)\in T} C_t(A_i,B_j) + \sum_{i\in I} C_d(A_i) + \sum_{j\in J} C_i(B_j)$$

That is, the total cost is just the sum of the costs for all the transformations, deletions, and insertions.

Now return to the diagrammatic representation of a trace T from A to B. Let A=A'A" and B=B'B", and suppose no line of T connects an element of A' to B" or A" to B'; that is the two strings A and B can each be split into two strings without having a line of T cross the split.

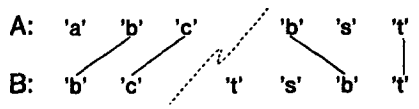

Figure 2-2: Splitting a trace

Corresponding to this split in A and B, T can be split into two traces, T' and T", in the obvious way. Furthermore, C(T) = C(T') + C(T"), so if T is a least cost trace from A to B, then so is T' a least cost trace from A' to B', and so is T" a least cost trace from A" to B".

Every trace T from A to B can be split into two traces T' and T" as above such that the lengths of A" and B" are each at most one but they are not both zero. This is the key idea for the following theorem, upon which the edit distance algorithm is based.

Notation: Let A and B be strings. Define $A(i)$ = a string composed of the first i elements of A, define $B(j)$ similarly. Define $A[i]$ = the ith element of A, and $B[j]$ similarly. Define $D_{i,j}$ = the minimum cost of a trace from $A(i)$ to $B(j)$. For convenience in handling boundary conditions: define $D_{i,-1} = \infty$, $D_{-1,j} = \infty$, and $\forall k \le 0$, $A[k]$, $B[k]$ = some unique string element that occurs in neither A nor B and that can be transformed at no cost to itself.

Theorem 1:

$D_{0,0} = 0$

$\forall i,j, i>0$ or $j>0, 0 \le i \le |A|, 0 \le j \le |B|$:

$D_{i,j} =$

$$\min\Big(D_{i-1,j-1} + C_t(A[i],B[j]),\ D_{i-1,j} + C_d(A[i]),\ D_{i,j-1} + C_i(B[j])\Big)$$

Proof: The first part of the proof ($D_{0,0} = 0$) is trivial, it is simply the cost of transforming the null string to the null string.

Let T be a least cost trace from $A(i)$ to $B(j)$. If $A[i]$ and $B[j]$ are both touched by lines in T, they must both be touched by the same line, since otherwise these lines in T would cross. Then at least one of the following three cases must hold:

1. $A[i]$ and $B[j]$ are joined by a line of T (i.e. $(i,j) \in T$). Then the cost of T is
   $$m_1 = D_{i-1,j-1} + C_t(A[i],B[j])$$
   corresponding to the cost of transforming $A(i-1)$ to $B(j-1)$ plus the cost of changing $A[i]$ to $B[j]$.
2. $A[i]$ is not touched by any line in T (and $B[i]$ may or may not be touched by a line in T). Then the cost of T is
   $$m_2 = D_{i-1,j} + C_d(A[i])$$
   corresponding to the cost of transforming $A(i-1)$ to $B(j)$ plus the cost of deleting $A[i]$.
3. $B[j]$ is not touched by any line in T. Then the cost of T is
   $$m_3 = D_{i,j-1} + C_i(B[j])$$
   corresponding to the cost of transforming $A(i)$ to $B(j-1)$ plus the cost of inserting $B[j]$.

Since one of the three cases above must hold and $D_{i,j}$ is to be a minimum:

$$D_{i,j} = \min(m_1, m_2, m_3)$$

This theorem leads directly to the following implementation of the minimum cost computation:

Algorithm 2:

```
D[-1,-1] = 0;
for i := 0 to |A| loop
    D[i,-1] := ∞;
end loop;
for j := 0 to |B| loop
    D[-1,j] := ∞;
end loop;
for i := 0 to |A| loop
    for j := 0 to |B| loop
        D[i,j] :=
            min (D[i-1,j-1] + C_t(A[i],B[j]),
                 D[i-1,j] + C_d(A[i]),
                 D[i,j-1] + C_i(B[j]));
    end loop;
end loop;
mincost = D[|A|,|B|];
```

Clearly this algorithm requires O(|A|·|B|) time and space. This time bound has been proved optimal in [10] for the restricted case of the longest common subsequence problem where only equal/unequal comparisons are allowed between elements of strings. By clever bookkeeping, the algorithm can be altered to only require O(min(|A|,|B|)) space [5].

Only the cost of the minimum cost trace is returned by this algorithm. To recover the trace itself, we have to traverse matrix D from $D_{|A|,|B|}$ back to the beginning, $D_{0,0}$. To aid in this traversal we define two functions: $W_{i,j}$ tells us which of the three operations (insertion, deletion or transformation) led to the optimal solution at *i,j*; it simply tells us which of the three operands of *min* in the calculation of $D_{i,j}$ led to the minumum. $P_{i,j}$ is an ordered pair which gives the subscripts of the subproblem in D of which $D_{i,j}$ is an extension.

$W_{i,j}$ =

`transformation`	if $D_{i,j} = D_{i-1,j-1} + C_t(A[i],B[j])$
`deletion`	if $D_{i,j} = D_{i-1,j} + C_d(A[i])$
`insertion`	if $D_{i,j} = D_{i,j-1} + C_i(B[j])$

$P_{i,j}$ =

(i-1,j-1)	if $W_{i,j}$ = `transformation`
(i-1,j)	if $W_{i,j}$ = `deletion`
(i,j-1)	if $W_{i,j}$ = `insertion`

An ambiguity exists if two of the operands of *min* in the calculation of $D_{i,j}$ are equal and minimum. If this occurs it means that there are multiple optimal solutions, and one can be chosen arbitrarily.

The following algorithm prints out the trace in reverse order:

Algorithm 3:

```
i := |A|;
j := |B|;
while i>0 or j>0 loop
    print(W[i,j], " at ", i, ",", j);
    (i,j) := P[i,j];
end loop;
```

3. An Example

Figure 3-1 shows the contents of matrix D after the execution of algorithm 2 on the data of figure 2-1 using the following cost functions:

$$C_t(A_i,B_j) = 0 \text{ iff } A_i = B_j \text{ and } \infty \text{ otherwise}$$
$$C_i(B_j) = 1$$
$$C_d(A_i) = 0$$

The two dashed lines represent the two minimum cost traces. These cost functions cause the algorithm to find the longest common subsequences of "abcbst" and "bctsbt"; namely "bcbt" and "bcst".

A \ B	-1	0	1 'b'	2 'c'	3 't'	4 's'	5 'b'	6 't'
-1	0	~	~	~	~	~	~	~
0	~	0	1	2	3	4	5	6
1 'a'	~	0	1	2	3	4	5	6
2 'b'	~	0	0	1	2	3	4	5
3 'c'	~	0	0	0	1	2	3	4
4 'b'	~	0	0	0	1	2	2	3
5 's'	~	0	0	0	1	1	2	3
6 't'	~	0	0	0	0	1	2	2

Figure 3-1: Sample cost matrix

4. The Redisplay Algorithm

This redisplay algorithm takes as input the description of two images. One describes the desired appearance of the display after the redisplay is complete. The other describes the appearance of the display before the redisplay is invoked. The algorithm depends on the display having the following properties:

1. The ability to rewrite in place a character on a given line in a given position.
2. The ability to delete a character from a given line and position. All following characters on that line are moved left one position, with a blank character entering at the right margin.
3. The ability to insert a character on a given line at a given position. The character originally at that position and all following characters are moved right one position. Characters that go past the right margin disappear.
4. The ability to delete a given line on the screen. All following lines on the screen are moved up one line, with a blank line entering at the bottom of the screen.

5. The ability to insert a line before a given line on the screen. The line originally given, and all following lines are moved down one line. Lines that go below the bottom of the screen disappear.

These capabilities exist in many commercially available video terminals. For example, The Concept-100, Heathkit H19, Infoton-400, and DEC VT-100. This algorithm was motivated by the desire to efficiently and effectivly exploit these common capabilities.

Consider the subproblem of transforming one line of a display given two strings that describe the appearance of the display both before and after the transformation. For now, we will use the simple cost functions of the preceeding example.

To do this transformation and maximize the number of characters that are not redrawn, we simply run algorithm 2 with these cost functions on the two strings. Then the trace gives us the characters that are to be preserved and how they map from the old to the new image. By the property of traces that no two lines cross, it is possible to do this operation using the allowed primitives: namely the simple insertion and deletion of characters, with the attendant left and right sliding of the rest of the line. If character i is to be moved to position j, then if $i=j$, then nothing need be done. If $i<j$ then move to position i and insert $j-i$ characters. Otherwise, if $i>j$, move to position j and delete $i-j$ characters.

This is, of course, overly simplified. One has to compensate for the fact that doing the transformation for one pair in the trace affects all pairs to the right of it. One must also compensate for the usual property of these displays that characters that move off the right edge of the screen are lost (lines that move off the bottom of the screen are also lost). If you do an insertion, then a deletion, the rightmost character of the line will be turned into a blank. This can be handled by doing all deletions first: simply traverse the trace twice. On the first traversal, if two adjacent pairs are found that require two characters to be moved closer together, then do a deletion. On the second traversal, if two adjacent pairs are found that require two characters to be moved farther apart, then do an insertion.

After these insertions and deletions have been done, all that remains is to redraw those characters that are not preserved by the trace.

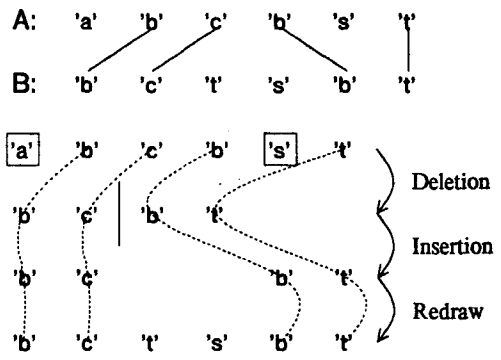

Figure 4-1: Execution of the simple redisplay

Figure 4-1 is an example of the insertions, deletions and writes that need to be done in order to perform the transformation indicated by the example in figure 2-1

To formalize this, look at $D_{i,j}$ and consider how the transformation represented by it is achieved given the ability to achieve the three neighbouring transformations.

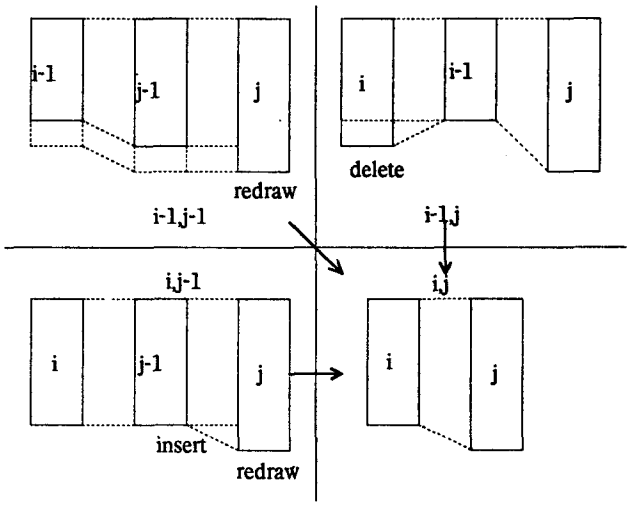

Figure 4-2: A single cell of D

To achieve our target transformation, the conversion of the first i elements of A to the first j elements of B, there are three cases, which correspond to to the three cases in the proof of theorem 1.

1. The optimal transformation can be derived from $D_{i-1,j-1}$ by first transforming the first i-1 elements of A to the first j-1 elements of B, then doing the transformation of the element A_i to B_j. It is important to note that element A_i will be in the same string position in the intermediate string as B_j in the target string after the i-1,j-1 transformation has been done because of the way that insert/delete operations behave: namely that all following elements get moved.

2. The optimal transformation can be derived from $D_{i,j}$ by first deleting A_i and then transforming the first i-1 elements of A to the first j elements of B. In the other two cases we solve the subproblem first, then extend that solution to a solution of the full problem. Here we reverse the order (delete *then* solve the subproblem) because we want to avoid intermediate strings longer than B. If the deletion was done first, then after the solution of the subproblem we could have an intermediate string of length > |B|, which because of the constraints of the display would be truncated to the first |B| elements.
3. The optimal transformation can be derived from $D_{i,j-1}$ by first transforming the first i elements of A to the first j-1 elements of B, then doing an insert operation at position j and the transformation of the null element that resulted from the insertion to B_j

This leads to the following algorithm:

Algorithm 4:

```
procedure Redisplay (i, j) is
    begin
        if (i,j) = (0,0) then
            return;
        end if;
        case W_{i,j} in
    when transformation =>
            Redisplay (i-1, j-1);
            TransformElement (j, A_i, B_j);
    when deletion =>
            DeleteElement (i);
            Redisplay (i-1, j);
    when insertion =>
            Redisplay (i, j-1);
            InsertElement (j);
            TransformElement (j, null, B_j);
        end case;
    end
```

Referring back to the set of display properties given at the beginning of this section, a strong symmetry between line and character operations is apparent. A line can be viewed as a string of characters, and a screen as a string of lines. Algorithm 2 can be employed in three ways in a complete redisplay algorithm:

1. To update individual lines, as has already been described.
2. To move lines, rather than characters, using the line insertion/deletion operations; minimizing the amount of work done.
3. As a cost function to determine the similarity between two lines.

This suggests the following algorithm to do the total redisplay:

Algorithm 5:

```
call Algorithm2 (
    A = string of lines from old image,
    B = string of lines from new image,
    C_d(A_i) = cost of deleting a line at i
    C_i(B_j) = cost of inserting a line at j+ C_t(null, B_j)
    C_t(A_i, B_j) = call Algorithm2 (
        A = A_i,
        B = A_j,
        C_d, C_i, C_t = those for the one-line case.
        { this invocation merely returns a cost,
          and does not touch the screen. }
    )
);
call Algorithm4 (
    A = string of lines from the old image,
    B = string of lines from the new image,
    The procedures InsertElement and DeleteElement
        do line oriented operations,
        TransformElement(p,old,new) is
    begin
        call Algorithm2 (
            A = old,
            B = new,
            C_d, C_i, C_t = those for the one-line case.
        );
        call Algorithm4 (
            A = old,
            B = new,
            The procedures InsertElement, DeleteElement,
                and TransformElement do character
                oriented operations on line p
        );
    end;
);
```

If l is the length of a line, and s is the number of lines on the screen, then this algorithm takes $O(s^2l^2)$ time. While this algorithm does an excellent job of minimising the number of characters transmitted, its runtime is unacceptable, even given a clever implementation [2]. So some compromises have to be made. In computer science, compromises are usually called "heuristics".

5. Performance heuristics

Most of the time used by algorithm 5 is consumed by the inner invocation of algorithm 2 as the cost function. Other, cheaper, cost functions can be used, but optimality is lost. Since C_i and C_d already take constant time, no improvements can be made there. Large improvements can be made by speeding up the evaluation of C_t, the cost of transforming one line to another.

One possibility for C_t is:

$$C_t(A_i,B_j) = 0 \text{ iff } A_i = B_j \text{ and } |B_j| \text{ otherwise.}$$

which still takes $O(l)$ worst case time since A_i and B_j are strings of length l. This can be speeded up at the cost of some accuracy by preprocessing the two arrays of strings; computing a hash value for each string. Then two strings can be compared in constant time.

$$C_t(A_i,B_j) = 0 \text{ iff } A_i^h = B_j^h \text{ and } |B_j| \text{ otherwise.}$$

Doing a string comparison only takes $O(l)$ time in the worst case. The expected time for a comparison is actually a constant which depends on the size of the alphabet, assuming that when strings are compared the comparison stops when two differing characters are encountered. For an alphabet size of 2 and a uniform distribution of characters the expected number of comparisons is 2, for larger alphabet sizes it is less. It the process of doing a string comparison a count of the number of matching characters may be kept, resulting in a measure of the two lines similarity, rather than just a match/nomatch indication. However, the overhead may still make the hash comparison method preferable.

The choice of C_t can be influenced by the intended application. For example, the project that originally motivated the research described by this paper was a structure editor; ie. an editor which manipulates a tree representation of the program, and the tree representation is reflected back on the screen with the appearance of a conventional program. A common operation is to embed a series of statements within a **begin-end** pair. The motion that one would like to see on the screen is for two lines to be inserted, in which the strings "begin" and "end" are written, and for all intervening lines to be moved right on the screen. This was done by using a hash of the contents of each line that ignored leading and trailing blanks. It has been satisfactory.

This gives us an $O(s^2)$ time line permutation phase, preceeded by $O(sl)$ time for preprocessing. This is followed by an $O(sl^2)$ time phase to update individual lines.

A cheaper method of doing the intra-line update is needed that doesn't compromise effectiveness too much. Many methods are possible, but the following has proved effective: Most non-total changes to a line affect only one small subpart. For example, inserting or deleting a character, or changing an identifier. The old and new lines can each be broken into three subparts: a leading match, a trailing match, and a differing string in the middle. The leading match is the longest common prefix of the two strings, the training match is the longest common suffix of the two strings, and the differing strings are just the regions in the two strings between the leading and trailing matches. Once this partitioning has been done, all that has to be done is to move the trailing match sequence in the original line with insert/delete character operations so that it lines up with the corresponding string in the new line, and redraw the central difference. Allowance must be made for the costs of performing the various operations, but this is simply a long and tedious case analysis.

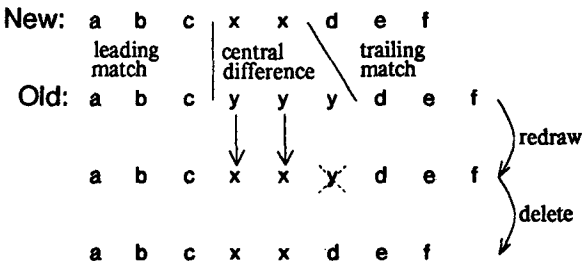

Figure 5-1: Intra-Line update

So, our final redisplay algorithm is:

Algorithm 6:
call Algorithm2 (
A = string of lines from old image,
B = string of lines from new image,
$C_d(A_i)$ = cost of deleting a line at i
$C_i(B_j)$ = cost of inserting a line at j + $C_t(null, B_j)$
$C_t(A_i, B_j)$ = 0 iff $A_i^h = B_j^h$ and $|B_j|$ otherwise.
);
call Algorithm4 (
A = string of lines from the old image,
B = string of lines from the new image,
The procedures *InsertElement* and *DeleteElement* do line oriented operations,
TransformElement(p,old,new) uses the technique just outlined.
);

The values of C_i and C_d are affected by display screens having a fixed length: when you do an insertion, the last line on the screen is deleted, and when you do a deletion, a blank line is inserted after the last line. Effectively for each insertion you get a free deletion, and for each deletion you get a free insertion at the bottom of the screen. This can be handled by setting $C_i(B[j]) = 0$ when evaluating $D_{s,j}$ and $C_d(A[i]) = 0$ when evaluating $D_{i,s}$. The corresponding insertion and deletion operations can be omitted when doing the redisplay.

6. Conclusion

The redisplay algorithm described in this paper is used in an Emacs-like editor for Unix and a structure editor. It's performance has been quite good: to redraw everything on the screen (when everything has changed) takes about 0.12 seconds CPU time on a VAX 11/780 running Unix. Using the standard file typing program, about 0.025 seconds of CPU time are needed to type one screenful of text. Emacs averages about 0.004 CPU seconds per keystroke (with one call on the redisplay per keystroke).

Although in the interests of efficency we have stripped down algorithm 5 to algorithm 6 the result is still an algorithm which has a firm theoretical basis and which is superior to the usual ad-hoc approach.

7. Acknowledgements

The people who did the real work behind this paper are Mike Kazar, Charles Lieserson and Craig Everhart; all from CMU.

Bibliography

1. Kevin Q. Brown. Dynamic Programming in Computer Science. CMU, February, 1979.

2. Craig Everhart. --. Personal communication

3. James Gosling. *Fred: a screen editor for Unix.* CMU CSD, 1979. Unpublished manual.

4. B. S. Greenberg. The Multics Emacs Redisplay Algorithm. Honeywell Inc., 1979.

5. D. S. Hirschberg. "A linear space algorithm for computing maximal common subsequences." *CACM 18* (1975), 341-343.

6. J. W. Hunt and T. G. Szymansky. "A Fast Algorithm for Computing Longest Common Subsequences." *CACM 20* (1977), 350-353.

7. W. J. Masek and M. S. Paterson. A Faster Algorithm for Computing String Edit Distances. Tech. Rept. 105, MIT, May, 1978.

8. Richard M. Stallman. *EMACS manual for TWENEX users.* MIT AI Lab, 1980.

9. H. M. Wagner and M. J. Fischer. "The string-to-string correction problem." *JACM 21*, 1 (January 1974), 168-173.

10. C. K. Wong and A. K. Chandra. "Bounds for the String Editing Problem." *JACM 23* (1976), 13-16.