Row Replacement Algorithms for Screen Editors

EUGENE W. MYERS
University of Arizona
and
WEBB MILLER
The Pennsylvania State University

Interactive screen editors repeatedly determine terminal command sequences to update a screen row. Computing an optimal command sequence differs from the traditional sequence comparison problem in that there is a cost for moving the cursor over unedited characters and the cost of an n-character command is not always the cost of n one-character commands. For example, on an ANSI-standard terminal, it takes nine bytes to insert one character, ten to insert two, eleven to insert three, and so on. This paper presents an $O(MN)$ dynamic programming algorithm for row replacement where an n-character command costs $\alpha n + \beta$ for constants α and β. M is the length of the original row and N is the length of its replacement. Also given is an $O(Cost \times (M + N))$ "greedy" algorithm for optimal row replacement. Here *Cost* is the optimal cost (in bytes) of the replacement, so the algorithm is fast when the required update is small. Though the algorithm is rather complicated, it is fast enough to be useful in practice.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Program Complexity**]: Nonnumerical Algorithms and Problems—*computations on discrete structures*

General Terms: Algorithms

Additional Key Words and Phrases: Dynamic programming, greedy algorithm, row replacement, screen editor

1. THE ROW REPLACEMENT PROBLEM

Screen-oriented programs maintain a representation of an object and present a view of it on the screen. For example, screen editors keep an internal edit buffer and display a block of lines from the buffer. The screen must be updated when the object is changed. In one solution, procedures that modify the object must also update the view or at least specify how the view has changed. Optimal use of such a package [1] may require learning and calling many different routines. A cleaner approach lets an autonomous screen manager module determine how

This work was supported in part by National Science Foundation grant DCR-8511455.
Authors' current addresses: E. W. Myers, Department of Computer Science, University of Arizona, Tucson, AZ 85721; W. Miller, Department of Computer Science, The Pennsylvania State University, University Park, PA 16802.

to update the screen by comparing its record of screen contents with views of the modified object. The interface to the screen manager is then a single routine, *refresh*, that updates the screen with respect to the current object. It is given no information other than the object and screen contents. Writers of screen editors are almost unanimous in recommending the use of an autonomous screen manager [2, 6, 10, 18, 24]. The design greatly improves the editor's internal structure, so it is appropriate with both ASCII and bitmap displays.

Care is needed to achieve satisfactory efficiency with an autonomous screen manager. In many contexts, data travels from the editor to the screen at between 100 and 1,000 characters per second (e.g., 1,200 or 9,600 baud), which is orders of magnitude slower than the editor's computing speed. Under such circumstances, it is worthwhile for the editor to compute a minimal, or near minimal, set of row-updating commands, as long as the time to do so does not outweigh the savings in character-transmission time. Although these considerations are far less important with high-speed communication and bitmap displays, the results of this paper will be useful as long as people want to access remote processors over slow transmission media, such as telephone lines.

For autonomous screen managers, a basic problem is to optimally update a screen row. An algorithm, presented with the row currently on the screen and the desired row contents, must produce a shortest-possible sequence of terminal commands that replaces the existing row with the desired row. This *row replacement problem* is complicated by varying terminal capabilities and command encoding schemes. This paper assumes the following terminal operations, which are provided by most modern terminals.

—*Clear.* The characters at, and to the right of, the cursor are deleted. The cursor does not move.

—*Delete.* The character at the cursor is deleted, causing later characters to be shifted left. The cursor does not move, so it ends up on the character that followed the deleted character.

—*Insert x.* The character x is inserted at the cursor's location, causing the characters at and right of the cursor to be shifted to the right. The cursor moves one position to the right, so it stays with the same character.

—*Move to k.* The cursor is moved to the kth column in the current row.

—*Replace by x.* The character x is displayed at the cursor's location, overwriting the previously displayed character. The cursor moves one position to the right.

Command encodings vary widely among terminal brands. Typically, *replace* is performed by simply sending the character to the terminal. Other commands contain escape sequences that encode the operator. Table I gives escape sequences for ANSI-standard terminals and for the IBM 3101 where ⟨*esc*⟩ denotes the "escape character". The *clear* and *delete* commands are three bytes long in one case, two in the other. With ANSI-standard terminals, a string of characters is inserted by sending a 4-byte sequence to put the terminal in insert mode, then sending the desired characters, and finally sending a 4-byte sequence to exit from insert mode. For the 3101, each character must be inserted individually using a 3-byte escape sequence. Thus, inserting n consecutive characters costs $n + 8$ bytes in one case and $3n$ bytes in the other.

Table I. Two Command Encoding Schemes

Instruction	ANSI standard	IBM 3101
clear	⟨*esc*⟩[*K*	⟨*esc*⟩*I*
delete	⟨*esc*⟩[*P*	⟨*esc*⟩*Q*
enter insert mode	⟨*esc*⟩[4*h*	—
insert character	—	⟨*esc*⟩*P*⟨*char*⟩
leave insert mode	⟨*esc*⟩[4*l*	—

On ANSI-standard terminals, the cursor is moved by the command '⟨*esc*⟩[α;βH' where α is the decimal representation of the row number and β is the column number. For example the 7-byte command '⟨*esc*⟩[5;20H' moves to row 5, column 20. For these terminals, the *move* command is 6, 7, or 8 bytes long, depending on the destination (and assuming less than 100 rows and columns). The 3101 uses the 4-byte *move* command '⟨*esc*⟩Y$\rho\gamma$'. Here ρ is a byte containing 31 plus the row number, and γ is 31 plus the column number. The designers add 31 because ρ and γ are then printable characters in the ASCII code.

Implementation of a row-replacement strategy requires that a number of additional details be addressed. With a line whose length exceeds the screen width, some editors fold the line onto several screen rows to make all characters visible. Other editors add a special character at the right margin of the screen to indicate an overlength line. In either case, a side effect of inserting a character in a long line is to delete the rightmost character in the row by pushing it off the screen. The implementation of a screen-update procedure must also be concerned about swamping the terminal by sending control sequences too quickly; "pad characters" or "handshaking" may be required. Such implementation details are not considered in this paper.

The commands discussed above are used for intraline editing or, as we call it, row replacement. Other commands, like *delete row* and *insert row*, manipulate entire screen rows. The methods used by screen editors to recycle entire rows are often surprisingly different from those for reusing parts of a row. A complete description of one design is given in [10, pp. 280–316], where the row-replacement strategy simply avoids retransmission of a common prefix and, if it saves time, of a common suffix. This paper is concerned only with row replacement, though methods for interline editing are mentioned briefly in the remainder of this section and under "Open Problems" in Section 5.

Gosling [6] proposes a rigorous approach to screen updating based on an off-the-shelf application of a sequence-comparison algorithm for both intra- and interline edits. However his comparison model ignores cursor-movement costs and charges an n-symbol operation like n one-symbol operations. Moreover efficiency considerations lead him to use the length of B as an approximation to the cost of replacing row A by B when evaluating possible interline editing scripts. With these *approximate* costs, the classic dynamic programming algorithm is quite practical for interline edits. Indeed Gosling used the approach in the UNIX EMACS editor, though row replacement is nonoptimally performed by simply retaining common prefixes and suffixes of the old and new lines. The resulting screen manager is quite independent of the remainder of the editor, as illustrated by the use of Gosling's code in the Maryland Window System [23].

Myers [15] uses sequence comparison techniques to solve the "window positioning problem": find a screen-sized block of buffer lines that contains the current line and reuses the maximum number of currently visible lines. If R is the number of screen rows, then there are generally R choices for the window position. A straightforward solution requires time $O(R^3)$ when only interline edits are considered, but Myers' algorithm needs only time $O(R^2)$. Miller and Myers [13] give another optimal row-replacement algorithm with worst-cost complexity $O(M^2N^2)$, but which works efficiently for many practical screen-updating problems.

This paper presents two worst-case efficient algorithms for optimal row replacement. That is, under certain assumptions about the terminal and the permissible scripts of screen-update commands, these algorithms minimize the number of bytes sent to the screen. Section 2 places these algorithms in historical perspective by sketching earlier work on the related sequence-comparison problem. Although sequence-comparison algorithms are not immediately applicable to row replacement, as shown in Section 2, they contain the seeds of our row-replacement algorithms. Sections 3 and 4 give $O(MN)$ and $O(Cost \times (M + N))$ row-replacement algorithms, respectively. Section 5 concludes with further refinements, empirical results, and open problems. The experiments cited in Section 5 show that the $O(Cost \times (M + N))$ row-replacement algorithm is fast enough to be of practical value.

2. SEQUENCE COMPARISON ALGORITHMS

Sequence-comparison problems arise in a variety of disciplines, including computer science and molecular biology [21]. The basic problem can be phrased in terms of editing operations as follows. Given sequences $A = a_1a_2 \cdots a_M$ and $B = b_1b_2 \cdots b_N$, determine a sequence of "basic operations" that converts A to B and minimizes the sum of the operations' costs (or "weights"). Traditionally, the only operations permitted are *Delete*, *Insert*, and *Replace*. There is no notion of a cursor; an operation may be applied at any point in the string. Moreover the cost of an operation does not depend on its context in the edit script.

In the *point indel* model, the basic operations insert a single symbol, delete a symbol, or replace one symbol by another, and an operation's cost may depend on the affected symbol or symbols:

$d\_cost(a)$	cost of deleting a
$i\_cost(b)$	cost of inserting b
$r\_cost(a, b)$	cost of replacing a with b when $a \neq b$

In the *block indel* model, an indel (i.e., insertion or deletion) of $k > 0$ consecutive symbols is treated as an atomic operation, rather than k operations on individual symbols. The operation is assigned a weight, or *gap penalty*, w_k that depends only on the number of symbols inserted or deleted. A variation of the model allows insertion costs to differ from deletion costs. The block indel model also includes a single-symbol *Replace* operation of cost $r\_cost(a, b)$.

Simple dynamic programming algorithms compute optimal edit scripts for the point indel problem in time $O(MN)$ [20] and for the block indel problem in time $O(MN(M + N))$ [22], where M and N are the sequence lengths. Let A_k denote

Table II. $C(i, j)$ for $A = abcabba$, $B = cbabac$

			c	b	a	b	a	c	
		0	1	2	3	4	5	6	
	0	0	1	2	3	4	5	6	
a	1	1	1	2	2	3	4	5	
b	2	2	2	1	2	2	3	4	
c	3	3	2	2	2	3	3	3	
a	4	4	3	3	2	3	3	4	...
b	5	5	4	3	3	2	3	4	diagonal 2
b	6	6	5	4	4	3	3	4	diagonal 1
a	7	7	6	5	4	4	3	4	diagonal 0
									diagonal −1
									diagonal −2
									...

the k-symbol prefix, $a_1a_2 \cdots a_k$, of A. A dynamic programming approach for finding the minimum cost of editing A into B requires computing $C(i, j)$, the minimum cost of editing A_i into B_j for all $i \in [0, M]$ and $j \in [0, N]$. For example, if A is *abcabba*, B is *cbabac*, and we allow single-symbol *Delete*, *Insert*, and *Replace* operations of cost 1, then C is shown in Table II. Observe that $C(4, 4) = 3$ since $A_4 = abca$ can be edited to $B_4 = cbab$ with either the sequence '*Replace* a_1 with b_1, *Delete* a_3, *Insert* b_4 after a_4' or the sequence '*Replace* a_1 with b_1, *Replace* a_3 with b_3, *Replace* a_4 with b_4'. Notice there is no cost associated with shifting attention from one part of the string to another and commands may be applied in any order.

The following $O(MN)$ dynamic programming algorithm [20] computes C for the point indel model.

Algorithm 1a. *Computing* $C(i, j)$.

```
C(0, 0) ← 0
for j ← 1, 2, ..., N do
  C(0, j) ← C(0, j − 1) + i_cost(b_j)
for i ← 1, 2, ..., M do
  { C(i, 0) ← C(i − 1, 0) + d_cost(a_i)
    for j ← 1, 2, ..., N do
      { c1 ← C(i − 1, j) + d_cost(a_i)
        c2 ← C(i, j − 1) + i_cost(b_j)
        if a_i = b_j then
          c3 ← C(i − 1, j − 1)
        else
          c3 ← C(i − 1, j − 1) + r_cost(a_i, b_j)
        C(i, j) ← min(c1, c2, c3)
      }
  }
```

The correctness of the algorithm can be seen by considering the computation of $C(i, j)$ in the innermost loop. Any edit script converting A_i into B_j must either (1) delete a_i, (2) insert b_j, or (3) replace a_i by b_j if $a_i \neq b_j$. If the script is optimal, then the remainder of the script optimally solves a subproblem. In case 1, the script consists of an optimal script converting A_{i-1} to B_j plus the *delete* command. In case 2, it consists of an optimal script converting A_i to B_{j-1} plus the *insert*

command. In case 3, it consists of an optimal script converting A_{i-1} to B_{j-1} plus the *replace* command if necessary. The algorithm considers the three forms of edit scripts, and selects the best.

Notice that extending an edit script by adding a *delete* operation corresponds to a vertical move in the table from the $(i-1, j)$ grid points to (i, j). Extensions by *insert* and *replace* operations correspond to horizontal and diagonal moves, respectively. The alignment of a_i with b_j when $a_i = b_j$ corresponds to a diagonal *zero-cost* move to grid point (i, j). A script for editing A to B corresponds to a sequence or path of moves from point (0, 0) to point (M, N). The cost of a script is the sum of the costs of the individual moves. Thus finding a minimum-cost script corresponds to finding a minimum-cost or shortest path from (0, 0) to (M, N).

The insight that string editing is a special instance of the shortest paths problem led to the design of faster "greedy" algorithms for the point indel problem where all operations cost 1 [11, 14, 19]. Let diagonal k consist of those grid points (i, j) for which $i + k = j$, as shown in Table II. The diagonals of the C matrix range from $-M$ to N. Let $L(c, k)$ be the length, i, of the longest prefix A_i that can be converted to B_{i+k} with a script of cost c. If no such prefix exists, define $L(c, k)$ to be -1. Because all costs are 1, it follows that the values in the C matrix are nondecreasing along diagonals and increase by at most 1 with each diagonal move. Thus, for this special problem, one can prove that $L(c, k)$ gives the row containing the last c in diagonal k if such a row exists. For example, $L(3, 1) = 4$ and $L(1, 3) = -1$. The reader should note that this characterization of L values in terms of C values will not suffice for the more general greedy algorithm of Section 4.

Algorithm 1b. *Computing* $L(c, k)$.

```
L(−1, 0) ← −1
for c ← 0, 1, 2, . . . do
  for k ← −c, −c + 1, . . . , c do
    {last ← L(c − 1, k) + 1
     if k < c then
       last ← max(last, L(c − 1, k + 1) + 1)
     if k > −c then
       last ← max(last, L(c − 1, k − 1))
     if last ≥ 0 then
       while last < M and last + k < N and a_{last+1} = b_{last+k+1} do
           last ← last + 1
     L(c, k) ← last
     if k = N − M and last = M then
       return "Cost is c"
    }
```

Greedy algorithms fill in L for $c = 0, 1, 2, \ldots$, until the lower, right-hand value of C is known, i.e., they find the value c for which $L(c, N - M) = M$. Informally, the logic for computing $L(c, k)$ goes as follows. We are looking for a c that is as far down diagonal k as possible. A script that costs c consists of a script of cost $c - 1$ plus another operation. If this final operation is *delete*, then the best we can do is to make a vertical move from the last $c - 1$ on diagonal $k + 1$; this reaches row $L(c - 1, k + 1) + 1$. If the final operation is *insert*, row $L(c - 1, k - 1)$ can be reached by a horizontal move from the last $c - 1$ on diagonal $k - 1$. If

the final operation is *replace*, row $L(c-1, k)+1$ can be reached with a diagonal move. From whichever move places us in the greatest row, *last*, we then follow zero-cost diagonal moves ($a_{last+1} = b_{last+k+1}$) until the last c entry is reached.

The central advantage of the greedy approach is that for a given c, L-values need to be computed only for k between $-c$ and c. Any script leading to diagonal k must involve k *insert* operations if k is positive or k *delete* operations if k is negative, and hence the script must have cost k or greater. Thus, for $k \notin [-c, c]$, we know that $L(c, k) = -1$ and hence may ignore such entries.

Suppose the minimum cost of editing A to B is *Cost*. The code within the two **for** loops is repeated $O(Cost^2)$ times since the outer loop is repeated $Cost + 1$ times and the inner loop is repeated $2c + 1$ times. Thus the algorithm consumes $O(Cost^2)$ time with the exception of the **while** loop, which must be accounted for separately. Charge each iteration of this loop to the grid point $(last + 1, last + k + 1)$ for which the character alignment occurs. Each grid point is charged at most once and all grid points processed are between diagonals $-Cost$ and $Cost$. Thus $O(Cost \times (M + N))$ time is spent in this loop, the dominant cost of the algorithm.

The $O(Cost \times (M + N))$ worst-case time bound tells only part of the story behind the superiority of the greedy approach. Indeed, there are other sequence comparison algorithms that achieve this performance [4, 19] and work for arbitrary (noninteger) costs. However, the greedy algorithm's expected performance is often far better than this bound indicates since not all entires of a given cost c are explicitly found. For example, when $r\_cost(a, b) = \infty$ (i.e., the longest common subsequence problem) the greedy algorithm has expected running time $O(N + Cost^2)$ [14].

Algorithms 1a and 1b determine only the minimum cost; they do not explicitly produce a shortest-possible edit script. Construction of a script is covered elsewhere [11, 14, 20]. It is easy to economize the use of space in both the dynamic programming and the greedy algorithms for the minimum cost; both require $O(M + N)$ space. It is possible [9, 14], though more difficult, to construct edit scripts in $O(M + N)$ space.

The dynamic programming and greedy algorithms developed above work only for the point indel model. For applications where the block indel model is appropriate, it is critical to find natural conditions on the gap weights w_k that permit the problem to be solved in time close to $O(MN)$, since the $O(MN(M + N))$ time for general weights is often prohibitive. A natural restriction is that w_k be *concave*:

$$\Delta w_k \geq \Delta w_{k+1} \quad \text{for all} \quad k \geq 1, \quad \text{where} \quad \Delta w_k = w_{k+1} - w_k$$

In words, w is concave if the cost of inserting or deleting an additional symbol decreases with the size of the affected substring. For concave weighting functions, the dynamic programming approach can be modified to run in time $O(MN \log(M + N))$ [12]. *Affine gap penalties*, i.e., the further restriction that $w_k = \alpha + \beta k$ for constants α, β, permit a solution in time $O(MN)$ [7]. Affine gap costs are currently preferred in biological applications [5], though more general concave weights have also been advocated.

The block indel model with affine indel costs is appropriate for screen update costs, such as the $(8 + n)$-byte cost of inserting n characters on an ANSI-standard

terminal. To see why the point indel would produce anomalous results, consider replacing row '*qbx* ···' with row '*yababz* ···'. If inserting n characters costs In for some constant I, then the command sequence, '*Insert y, Move* to 4, *Insert ab, Replace* by *z*', costs the same as '*Insert yab, Move* to 6, *Replace* by *z*'. However, with ANSI-standard terminals, the first command set costs 8 bytes more than the second (assuming that *move* is not permitted in insert mode).

While the algorithmic paradigms of traditional sequence comparison algorithms are relevant to row replacement, these algorithms do not immediately solve the row-replacement problem. A major shortcoming is that the string edit models ignore the cost of cursor movement. For example, consider the problem of replacing row '*xaxaxaxax*' with '*yayayayay*'. On typical terminals, repeatedly moving the cursor to the next *x* and replacing it by *y* is inferior to simply replacing every character. The *clear* operation is also omitted from the edit models. For updating screen row 'repeated repeated' to the string 'repeated', moving the cursor to the space between the two words and clearing the row is less expensive on an ANSI-standard terminal than nine *delete* commands.

3. A DYNAMIC PROGRAMMING ALGORITHM FOR ROW REPLACEMENT

Let *OPS* be the set {*delete, insert, move, replace*} and let *OPS*+ be the full operation repertoire, *OPS* ∪ {*clear*}. In the row-replacement framework, the cost of n consecutive *op* commands is assumed to be $startup(op) + n \times perchar(op)$ where $startup(op)$ and $perchar(op)$ are nonnegative integers modeling the startup and per-character costs for *op*. For example, Table III gives cost parameters for ANSI-standard terminals. The startup cost is paid (1) for the first operation of an edit script, and (2) whenever an operation differs from the previous operation. Thus, '*Replace* by *a*, *Move* to 4, *Insert xy*' costs $startup(replace) + 1 \times perchar(replace) + startup(move) + 2 \times perchar(move) + startup(insert) + 2 \times perchar(insert) = 19$ for ANSI-standard terminals. The coefficient of $perchar(move)$ is 2 because the cursor is moved *two* columns to the right.

This cost accounting is a simplification. For example, it does not model the variable cost of 6–8 bytes for an ANSI-standard *move* command. (We assume conservatively that *move* costs 8 bytes.) Moreover some terminal brands allow cursor movement, deleting of characters, etc., while in "insert mode," so this approach may overestimate attainable costs by charging for unnecessary startups. However, the astute reader will observe that our row-replacement algorithms can be adapted to cover such variations.

The algorithms developed in this section and the next compute a minimum-cost left-to-right conversion of A to B. A conversion is *left-to-right* if *move* operations always move the cursor one or more columns to the right of its current position. If the cursor starts in column 1 and characters are not "spilled" off the end of the row, then a minimum-cost left-to-right script is guaranteed to be optimal over all row-updating scripts. Otherwise near-optimal scripts can be obtained from our left-to-right scripts as follows. If the cursor is not initially in position 1, then preface the script with '*Move 1*'. If a spill occurs, then append commands that move to the end of the line and restore the spilled characters. Alternatively our algorithms can be modified to consider only left-to-right scripts that avoid spills. To do this, entries of the cost matrix above diagonal $W - M$ are ignored, where W is the screen width and M is A's length.

Table III. ANSI Standard Command Costs

	startup	*perchar*
clear	3	0
delete	0	3
insert	8	1
move	8	0
replace	0	1

Table IV. $C(i, j, op)$ for $A = abcabba$, $B = cbabac$

		0		c 1		b 2		a 3		b 4		a 5		c 6	
0		8	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
		8	0	9	∞	10	∞	11	∞	12	∞	13	∞	14	∞
a 1		∞	3	∞	12	∞	13	18	14	∞	15	20	16	∞	17
		∞	∞	12	1	10	10	11	11	12	12	13	13	14	14
b 2		∞	6	∞	4	9	13	∞	14	18	15	∞	16	∞	17
		∞	∞	15	4	13	2	11	11	12	12	13	13	14	14
c 3		∞	9	14	7	∞	5	∞	14	∞	15	∞	16	21	17
		∞	∞	18	7	16	5	14	3	12	12	13	13	14	14
a 4		∞	12	∞	10	∞	8	13	6	∞	15	20	16	∞	17
		∞	∞	21	10	19	8	17	6	15	4	13	13	14	14
b 5		∞	15	∞	13	18	11	∞	9	13	7	∞	16	∞	17
		∞	∞	24	13	22	11	20	9	18	7	16	5	14	14
b 6		∞	18	∞	16	21	14	∞	12	17	10	∞	8	∞	17
		∞	∞	27	16	25	14	23	12	21	10	19	8	17	6
a 7		∞	21	∞	19	∞	17	21	15	∞	13	17	11	∞	9
		∞	∞	30	19	28	17	26	15	24	13	22	11	20	9

Legend:

		j
i	$C(i, j, \text{move})$	$C(i, j, \text{delete})$
	$C(i, j, \text{insert})$	$C(i, j, \text{replace})$

The basic row-replacement algorithm operates in two steps. First a dynamic-programming algorithm fills in a table giving minimum costs for row replacement using certain restricted kinds of edit scripts. In a second pass, boundary values in the table are inspected to account for operations omitted from the first step.

A *restricted replacement script* for $A_i = a_1 a_2 \cdots a_i$ and $B_j = b_1 b_2 \cdots b_j$ is a left-to-right script that satisfies:

(1) The cursor ends in column $j + 1$,

(2) *clear* is not used, and

(3) *replace* commands are not permitted once the cursor is beyond the end of the current string.

One should think of $a_1 a_2 \cdots a_i$ as terminated by a "null character" that is carried along as the edit operations are performed. The cursor ends up positioned on it. With restricted replacement scripts, one may insert at the null character or move to it, but it may not be deleted or replaced.

For each $i \leq M$, $j \leq N$, and $op \in OPS$, define $C(i, j, op)$ to be the minimum cost over all restricted replacement scripts for A_i and B_j whose last operation is op. Define $C(i, j, op)$ to be ∞ if no such script exists. Table IV gives $C(i, j, op)$ for

the problem $A = abcabba$ and $B = cbabac$ under the ANSI-standard cost model. For example, $C(3, 2, delete) = 5$ because '*Replace* by *cb*, *Delete*' is a restricted replacement script for *abc* and *cb* that is optimal among those ending with *delete*. Moreover, since a_3 differs from b_2, no restricted replacement script converting *abc* to *cb* ends with *move*; thus, $C(3, 2, move) = \infty$.

Reasoning similar to that for the dynamic programming algorithm of Section 2 leads to the following $O(MN)$ algorithm for computing $C(i, j, op)$ for all choices of i, j, and op. This algorithm and the next use the definition:

$$onechar(op_1, op_2) = \begin{cases} perchar(op_2) & \text{if } op_1 = op_2 \\ startup(op_2) + perchar(op_2) & \text{if } op_1 \neq op_2 \end{cases}$$

$Onechar(op_1, op_2)$ is the cost of applying op_2 to a character if the previous operation was op_1.

Algorithm 2a. *Computing* $C(i, j, op)$.

```
for op ∈ OPS do
  C(0, 0, op) ← startup(op)
for j ← 1, 2, ..., N do
  { C(0, j, insert) ← C(0, j − 1, insert) + perchar(insert);
    C(0, j, delete) ← C(0, j, move) ← C(0, j, replace) ← ∞
  }
for i ← 1, 2, ..., M do
  { C(i, 0, delete) ← C(i − 1, 0, delete) + perchar(delete)
    C(i, 0, insert) ← C(i, 0, move) ← C(i, 0, replace) ← ∞
    for j ← 1, 2, ..., N do
      for op ∈ OPS do
        { C(i, j, op) ← ∞
          if not (op = move and a_i ≠ b_j) then
            { if op = insert then
                I ← i
              else
                I ← i − 1
              if op = delete then
                J ← j
              else
                J ← j − 1
              for θ ∈ OPS do
                C(i, j, op) ← min(C(i, j, op), C(I, J, θ) + onechar(θ, op))
            }
        }
  }
```

$C(0, 0, op)$ is initialized to $startup(op)$ so that the first operation in a script is charged a startup cost. Values in row 0 give costs for converting the empty string to B_j. With restricted replacement scripts this conversion requires j *insert* operations, so for $j > 0$:

$$C(0, j, insert) = startup(insert) + j \times perchar(insert)$$
$$C(0, j, op) = \infty \text{ for } op \in \{delete, replace, move\}$$

Similarly, converting A_i for $i > 0$ to the empty string requires i *delete* operations, so:

$$C(i, 0, delete) = startup(delete) + i \times perchar(delete)$$
$$C(i, 0, op) = \infty \text{ for } op \in \{insert, replace, move\}$$

Consider the computation of $C(i, j, delete)$ where $i > 0$ and $j > 0$. We want the optimal cost of a restricted replacement script converting A_i to B_j that ends with *delete*. If the *delete* is removed, then the reduced script optimally converts A_{i-1} to B_j and its length is one of the values $C(i-1, j, \theta)$ where $\theta \in$ OPS. It then follows that

$$C(i, j, delete) = \min_{\theta \in OPS} (C(i-1, j, \theta) + onechar(\theta, delete))$$

Similar reasoning verifies the remaining recurrences that justify the algorithm:

$$C(i, j, insert) = \min_{\theta \in OPS} (C(i, j-1, \theta) + onechar(\theta, insert))$$

$$C(i, j, replace) = \min_{\theta \in OPS} (C(i-1, j-1, \theta) + onechar(\theta, replace))$$

$$C(i, j, move) = \text{if } a_i \neq b_j \text{ then } \infty \text{ else } \min_{\theta \in OPS} (C(i-1, j-1, \theta) + onechar(\theta, move))$$

Once C has been computed, more general row-replacement scripts can be considered. First, *move* operations can be removed from the ends of scripts since they do not help transform A to B. Second, *clear* commands can be utilized. Third, instead of inserting characters at the right end of B, the *replace* operation can be used to simply write them. Additional improvements are possible, but are not considered here. For example it might be preferable to delete the last character of A by replacing it by a blank.

The following $O(M + N)$ second pass examines the array C produced in the first step in order to treat more general replacement scripts. $Min4(i, j)$ is assumed to return the minimum of $C(i, j, op)$ over $op \in OPS$.

Algorithm 2b. *Optimizations after C has been computed.*

```
optimal ← min4(M, N)
/* account for the common suffix */
i ← M
j ← N
while i > 0 and j > 0 and a_i = b_j do
    { i ← i − 1
      j ← j − 1
      optimal ← min(optimal, min4(i, j))
    }
/* account for the clear operation */
for i ← 0, 1, . . . , M − 1 do
  for op ∈ OPS do
    optimal ←
      min(optimal, C(i, N, op)+onechar(op,clear)+(M−i−1)×perchar(clear))
/* account for overwriting beyond A */
for j ← 0, 1, . . . , N − 1 do
  for op ∈ OPS do
    optimal ←
      min(optimal, C(M, j, op)+onechar(op,replace)+(N−j−1)×perchar(replace))
return optimal
```

These optimizations are based on the following observations:

(1) Suppose $a_{i+1}a_{i+2} \cdots a_M = b_{j+1}b_{j+2} \cdots b_N$ and S is a restricted replacement script converting A_i to B_j. Then S converts A to B (and leaves the cursor

on b_{j+1}). Note that a common suffix is not necessarily preserved by an optimal script, and hence may not be automatically stripped from the original problem.

(2) It can be assumed without loss of generality that *clear*, if it appears in an optimal replacement script, is the final operation. This is because (a) a *clear* or *delete* after a *clear* is meaningless; (b) *move* (to the right) after a *clear* is also useless since blanks are not treated specially; and (c) a *clear-insert* or *clear-replace* pair is equally well performed by an *insert-clear* or a *replace-clear*, respectively. Suppose S is a restricted replacement script converting A_i to B. Appending a *clear* operation to S gives a script that converts A to B at cost $cost(S) + startup(clear) + (M - i) \times perchar(clear)$.

(3) Suppose A is converted to B_j at cost $C(M, j, op)$ by a restricted replacement script that ends with the operation *op*. The script positions the cursor just after b_j, so simply replacing by $b_{j+1}b_{j+2} \cdots b_n$ produces B and the augmented script costs $C(M, j, op) + onechar(op, replace) + (N - j - 1) \times perchar(replace)$.

It takes only $O(M + N)$ additional time to produce a minimum-cost replacement script once the cost matrix C is computed. First determine the entry $C(i, j, op)$ from which *optimal*'s value was attained in the post-pass and output as the last command '*clear*', '*Replace* by $b_{j+1} \cdots b_N$', or nothing, as is appropriate. Then trace a path of vertical, horizontal, and diagonal moves backwards from $C(i, j, op)$ to $C(0, 0, op')$ such that the value at the end of each move is the value at the start plus the *onechar* cost of the move. Each move to entry $C(i, j, op)$ corresponds to a command: $C(i, j, insert)$ to '*Insert* a_i', $C(i, j, delete)$ to '*Delete*', $C(i, j, replace)$ to '*Replace* by b_j', and $C(i, j, move)$ to '*Move* to $j + 1$'. Listing these commands as the path is traversed in the forward direction gives the restricted portion of the script.

4. A GREEDY ALGORITHM FOR ROW REPLACEMENT

The greedy algorithm considers, for successive values of c, prefixes of row A that can be edited into a prefix of row B at cost c. Suppose the minimum cost of replacing A with B is *Cost*. The greedy algorithm examines only $O(Cost \times (M + N))$ grid points of the cost matrix C of Section 3, so it is faster than the dynamic programming algorithm when *Cost* is small.

The greedy algorithm's complexity is reduced by permitting left-to-right scripts to include *move* operations that leave the cursor in its current position. However no optimal script contains such commands, so reported answers are unaffected. This relaxation was not considered in Section 3 because it unnecessarily complicates the computation of C.

For given integers c and k and $op \in OPS$, let $L(c, k, op)$ be the length, i, of the longest prefix A_i that can be converted to B_{i+k} by a restricted replacement script of cost c that ends with operation *op*. If no such prefix exists, then let $L(c, k, op)$ equal *NONE*, a negative number, say -1. The relationship between L and the matrix C of Section 3 can be captured using the correspondence between edit scripts and paths sketched in Section 2. Computing C corresponds to finding, for each grid point (i, j) and operation *op*, the cost of a shortest path from (0, 0) to

the grid point ending with an *op* move. Computing L corresponds to finding, for each c, k, and *op*, the furthest position along diagonal k that can be reached by a path of cost c ending with *op*. For the example giving rise to Table IV, $L(13, -1, move) = 5$ since the last 13 in a *move* position on diagonal -1 occurs in row 5 of the C matrix. Also, $L(7, -1, move) = NONE$ because a script containing a *move* must cost at least 8.

With the general cost functions of the row-replacement problem, there can be paths of some cost c reaching some diagonal k and ending with some operation *op* even if no c appears in an *op* entry of diagonal k of the C matrix. For such a c, k, and *op*, $L(c, k, op)$ is defined but (unlike the case where all costs are 1) cannot be characterized as the last row where a c occurs in a diagonal of C. For example suppose deleting or inserting n symbols costs $2n$ and replacing n symbols costs $3n$. The *replace* entries on diagonal 0 are 0, 3, 6, . . . , but $L(7, 0, replace) = 2$ because the script '*Delete*, *Insert* b_1, *Replace* by b_2' edits A_2 to B_2. However if c occurs in an *op* position in diagonal k of C, then $L(c, k, op)$ is the row containing the last such c.

For values of c in the order 1, 2, . . . , the greedy algorithm computes $L(c, k, op)$ for a relevant range of diagonals k and all choices of *op*. The algorithm can stop when it reaches the first value of c for which $L(c, N - M, op) = M$ for some *op* because it has encountered the smallest cost, c, for which $A_M = A$ can be edited into $B_{M+(N-M)} = B$. We first show how to compute $L(c, k, op)$ from previously computed values. General scripts are then discussed. Finally the complete algorithm is presented and its complexity analyzed.

The greedy algorithm presented below rests on three additional but realistic assumptions about the cost parameters:

(A1) $perchar(delete) > 0$ and $perchar(insert) > 0$

(A2) $perchar(replace) > 0$

(A3) $perchar(move) = 0$ and $startup(move) > 0$

These assumptions are needed for two reasons. First they permit the formulation of computable recurrences for the L values of cost c from the L values of strictly smaller cost. Specifically, Assumptions A1 and A2 guarantee that $best(c, k, op)$ (defined below) is a function of L values of cost strictly less than c and A3 guarantees the same property for $mbest(c, k)$. The second reason is to ensure that the band of diagonals considered has width not exceeding $2c + 1$. Specifically, A1 guarantees that neither $upper(c)$ nor $lower(c)$ (defined below) exceeds c.

For the greedy method to be efficient, Assumption A1 is essential. Without it the approach produces an $O(MN)$ algorithm, no better than dynamic programming, because every diagonal must be considered. However Assumptions A2 and A3 are not critical. A3 can be changed to $perchar(move) > 0$ and A2 can be changed to $perchar(replace) = 0$ and $startup(replace) > 0$, without changing the feasibility or time complexity of the greedy approach. Nonetheless either the assumptions or their alternates must be chosen *a priori* because the choice affects the logic of the algorithm. The most questionable of the assumptions is $perchar(delete) > 0$, since some terminals support a fixed length "delete n symbols" command. However many of these devices disguise a greater-than-zero

perchar cost by requiring the transmission of a number of "pad characters" proportional to n.

Fix $op \in OPS$ and integers $c > 0$ and k, and suppose $L(C, K, \theta)$ has been determined for all $C < c$ and all K and θ. Let $firstop(op)$ be the cost of a first operation in an edit script:

$$firstop(op) = startup(op) + perchar(op)$$

Then $L(c, k, op)$ is computed by:

Algorithm 3a. *Computing* $L(c, k, op)$.

```
long ← NONE
if op = delete then
  K ← k + 1
else if op = insert then
  K ← k − 1
else
  K ← k
if c = firstop(op) and K = 0 then
  long ← 0
else if c > firstop(op) then
  for θ ∈ OPS do
    if not θ = op = move then
      long ← max(long, L(c − onechar(θ, op), K, θ))
if long ≠ NONE then
  if op = move then
    while long < M and long + k < N and a_{long+1} = b_{long+k+1} do
      long ← long + 1
  else if op ≠ insert then
    long ← long + 1
L(c, k, op) ← long
```

The only script of cost $firstop(op)$ ending with op consists of a single op command. A single *delete* command edits A_1 to B_0, a single *replace* command edits A_1 to B_1, and a single *insert* command edits A_0 to B_1. The best single *move* command edits the longest common prefix of A and B into itself. These observations lead to the equations:

$$L(firstop(delete), -1, delete) = 1$$
$$L(firstop(insert), 1, insert) = 0$$
$$L(firstop(replace), 0, replace) = 1$$
$$L(firstop(move), 0, move) = max(i \parallel A_i = B_i)$$

For any other case with $c \leq firstop(op)$, $L(c, k, op) = NONE$. These observations verify the correctness of Algorithm 3a for small values of c.

For $c > firstop(delete)$, consider $long = L(c, k, delete)$, the length of the longest prefix A_{long} transformable to B_{long+k} by a cost-c script ending with *delete*. Define:

$$best(c, k, op) = \max_{\theta \in OPS} (L(c - onechar(\theta, op), k, \theta))$$

Suppose that $long \neq NONE$ and that stripping the final *delete* from a corresponding script leaves a script ending with operation θ. The truncated script costs $c\text{-}onechar(\theta, delete)$ and converts A_{long-1} to B_{long+k}. Thus $long - 1 \leq L(c\text{-}onechar(\theta, delete), k + 1, \theta) \leq best(c, k + 1, delete)$. We claim that $long - 1 = best(c, k + 1,$

delete). Suppose there were a longer prefix, $A_{long'-1}$ with $long' > long$, that could be transformed to $B_{long'+k}$ with a script costing $c - onechar(\theta', delete)$ and ending with operation θ'. Then appending a *delete* command gives a cost-c script that transforms $A_{long'}$ to $B_{long'+k}$, contradicting the maximality of *long*. Thus $L(c, k, delete) = best(c, k + 1, delete) + 1$ if it exists. Intuitively, to reach as far as possible down diagonal k at cost c, "greedily" append a *delete* move to a furthest-reaching path of cost $c - onechar(\theta, delete)$ ending in diagonal $k + 1$. Finally observe that $L(c, k, delete) = NONE$ if and only if $L(c - onechar(\theta, delete), k + 1, \theta) = NONE$ for all $\theta \in OPS$. Similar reasoning for *replace* and *insert* leads to the following recurrences.

$$
\begin{aligned}
L(c, k, delete) &= \text{if } best(c, k + 1, delete) \neq NONE \\
&\quad \text{then } best(c, k + 1, delete) + 1 \text{ else } NONE \\
L(c, k, replace) &= \text{if } best(c, k, replace) \neq NONE \\
&\quad \text{then } best(c, k, replace) + 1 \text{ else } NONE \\
L(c, k, insert) &= \text{if } best(c, k - 1, insert) \neq NONE \\
&\quad \text{then } best(c, k - 1, insert) \text{ else } NONE
\end{aligned}
$$

For a concrete example, consider computing $L(19, -1, insert)$ for the ANSI-standard model, $A = abcabba$ and $B = cbabac$ (Table IV of Section 3). We seek the longest prefix A_i that can be transformed into B_{i-2} at cost $19 - onechar(\theta, insert)$ for some θ, since appending an *insert* command to a corresponding script gives a script that transforms A_i to B_{i-1} at cost 19. $L(10, -2, delete) = L(10, -2, replace) = 6$, so taking θ to be *delete* or *replace* shows that A_6 can be edited to B_5 at cost 19. Trying $\theta = move$ is fruitless, since $L(10, -2, move) = NONE$. Extending the script for $L(18, -2, insert) = 3$ obtains a script for transforming A_3 to B_2. Thus, $L(19, -1, insert)$ is 6.

Computing $L(c, k, move)$ requires a different approach since $perchar(move)$ equals zero. For $c > firstop(move)$, consider $long = L(c, k, move)$, the length of the longest prefix A_{long} transformable to B_{long+k} by a cost-c script ending with *move*. If there are several such scripts, consider one whose final *move* command moves right the *fewest* columns. Let:

$$mbest(c, k) = \max_{\theta \neq move} (L(c\text{-}onechar(\theta, move), k, \theta))$$

Suppose that $long \neq NONE$ and that stripping the final *move* from a corresponding script leaves a script ending with operation θ. Note that $\theta \neq move$ since two consecutive *moves* can be economized into one. The truncated script costs $c - onechar(\theta, move)$ and converts A_{long-m} to $B_{long-m+k}$, where m is as small as possible. Thus $long - m \leq L(c - onechar(\theta, move), k, \theta) \leq mbest(c, k)$ and $a_{long-m+1} \cdots a_{long} = b_{long-m+k+1} \cdots b_{long+k}$. We claim that $long - m = mbest(c, k)$. Suppose there were a longer prefix, $A_{long-m'}$ with $m' < m$, that could be transformed to $B_{long-m'+k}$ by a script costing $c\text{-}onechar(\theta', move)$ and ending with operation θ'. If $m' \geq 0$, then appending a *move* command that moves the cursor m' columns to the right gives a cost-c script that transforms $A_{long'}$ to $B_{long'+k}$, contradicting the minimality of m. If $m' < 0$, then appending a *move* command that moves zero columns to the right gives a cost-c script that transforms

$A_{long-m'}$ to $B_{long-m'+k}$, contradicting the maximality of *long*. This reasoning justifies the recurrence:

$$
\begin{aligned}
&L(c, k, move)\\
&\quad = \text{if } mbest(c, k) \neq NONE \text{ then}\\
&\qquad \max(i \parallel a_{mbest(c,k)+1} \cdots a_i = b_{mbest(c,k)+k+1} \cdots b_{i+k} \text{ or } i = mbest(c, k))\\
&\quad \text{else}\\
&\qquad NONE
\end{aligned}
$$

As the value of c is increased, computed values may satisfy $L(c, k, op) > M$ or $L(c, k, op) + k > N$. This happens because the commands *delete*, *replace*, *insert*, and *move-zero-columns-right* are effective regardless of the characters actually in A or B. Thus such an entry indicates that any supersequence of A can be converted to a prefix of B or that any prefix of A can be transformed into a supersequence of B at cost c. Such entries could be eliminated by an additional boundary check but this is unnecessary since they cannot contribute to the optimal solution.

Now turn attention to general row-replacement scripts. Since the algorithm's goal is to terminate as soon as the optimal value of c is reached, the additional features of general scripts cannot be handled in a postpass, as with the dynamic programming algorithm. Instead, checks for general cost-c scripts are "folded" into the computation of L-values.

First consider removal of a final *move* operation. Before starting the computation of L, the longest common suffix of A and B is identified by setting:

$$lim = \min(i \parallel a_{i+1} \cdots a_M = b_{i+(N-M)+1} \cdots b_N \text{ or } i = M)$$

Pictorially this step finds a line segment beginning at the lower right corner of the grid and extending toward the upper left until the row and column labels disagree. If $long = L(c, N - M, op) \geq lim$ for some operation *op* then there is a cost-c script converting A_{long} to $B_{long+(N-M)}$ and $a_{long+1} \cdots a_M = b_{long+(N-M)+1} \cdots b_N$. This script transforms A to B and leaves the cursor on column $long + 1$. For general edit scripts the algorithm terminates upon encountering a value of c for which $long \geq lim$.

As explained in Section 3, if *clear* commands are permitted, then it suffices to consider a single *clear* at the end of the script. If $L(c, k, op) = N - k$ for some $k > N - M$, then there is a cost-c script converting A_{N-k} to B where $N - k < M$. Appending a *clear* operation gives a script that edits A into B of cost $c + onechar(op, clear) + (k - (N - M) - 1) \times perchar(clear)$. The preceding argument assumes the *clear* operation is appended to a nonempty script, i.e., $c > 0$. In the special case where $N = 0$, a single *clear* edits A into B at cost $startup(clear) + M \times perchar(clear)$. As scripts editing A into B and ending with *clear* are detected, their minimum cost is recorded in a variable *Clim*. If the loop variable c reaches this value, the algorithm terminates and reports that the minimum row-replacement cost is *Clim*. If an optimal script editing A to B ends with *clear*, it will be discovered because removing the *clear* from at least one such script leaves a script that edits a longest prefix of A into B.

Handling overwrites beyond A is the analog of final *clear* operations. If $L(c, k, op) = M$ for some $k < N - M$, then there is a cost-c script converting A_M to B_{M+k} where $M + k < N$. Appending $N - (M + k)$ *replace* operations

gives a script converting A to B of cost c + *onechar*(*op*, *replace*) + (($N - M$) − k − 1) × *perchar*(*clear*). In the special case where $M = 0$, N *replace* commands edit A into B at cost *startup*(*replace*) + N × *perchar*(*replace*). It is sufficient to let the variable *Clim* above also record the minimum cost of these overwrite-terminated scripts as each is detected.

Up to this point it has been assumed that for each value c, $L(c, k, op)$ is computed for the entire range of diagonals, $k \in [-M, N]$. A central advantage of the greedy approach is that only $O(c)$ L-values need to be computed since the remainder are easily inferred to be *NONE*. Let:

$$lower(c) = \max\left(\frac{c - startup(delete)}{perchar(delete)}, 0\right)$$

$$upper(c) = \max\left(\frac{c - startup(insert)}{perchar(insert)}, 0\right)$$

If a cost-c restricted replacement script transforms A_i into B_{i+k} where $k > 0$, then it must insert at least k characters. Thus $c \geq startup(insert) + k \times perchar(insert)$ because at best the k *insert* commands are contiguous. Rearranging the inequality gives the relationship $k \leq upper(c)$. Arguing similarly about the case where $k < 0$ shows that $k \geq -lower(c)$ and leads to the fact:

$$\text{If } k \notin [-lower(c), upper(c)] \text{ then } L(c, k, op) = NONE.$$

The algorithm uses this fact in two ways. First, for each value of c it only computes $L(c, k, op)$ for $k \in [-lower(c), upper(c)]$. Second, when referencing previously computed values, $L(C, K, \theta)$, in the innermost loop, it infers that the value is *NONE* if $K \notin [-lower(C), upper(C)]$.

Algorithm 3b. *The greedy row-replacement algorithm.*

```
/* identify the longest common suffix */
lim ← M
while lim > 0 and lim + N − M > 0 and a_lim = b_lim+N−M do
  lim ← lim − 1
if lim = 0 and M = N then
  return "cost is 0"

if M = 0 then
  Clim ← startup(clear) + N × perchar(clear)
else if N = 0 then
  Clim ← startup(replace) + M × perchar(replace)
else
  Clim ← ∞
for c ← 1, 2, ..., Clim do
  for k ← −lower(c), −lower(c) + 1, ..., upper(c) do
    for op ∈ OPS do
      { long ← NONE
      if op = delete then
        K ← k + 1
      else if op = insert then
        K ← k − 1
      else
        K ← k
      if c = firstop(op) and K = 0 then
        long ← 0
```

```
        else if c > firstop(op) then
          for θ ∈ OPS do
          { C ← c − onechar(op, θ)
          if not θ = op = move and K ∈ [−lower(C), upper(C)] then
            long ← max(long, L(C, K, θ))
          }
        if long ≠ NONE then
          if op = move then
            while long < M and long + k < N and a_{long+1} = b_{long+k+1} do
              long ← long + 1
          else if op ≠ insert then
            long ← long + 1
        L(c, k, op) ← long
        /* account for the common suffix */
        if k = N − M and long ≥ lim then
          return "cost is c"
        /* account for the clear operation */
        if k > N − M and long = N − k then
          Clim ← min(Clim,c+onechar(op,clear)+(k−(N−M)−1)×perchar(clear))
        /* account for overwrite operations */
        if k < N − M and long = M then
          Clim ←
            min(Clim, c+onechar(op,replace)+((N−M)−k−1)×perchar(replace))
    }
  return "cost is Clim"
```

Let *Cost* be the optimal cost of replacing row A with row B of lengths M and N, respectively. The greedy algorithm runs in time $O(Cost \times (M + N))$. Identifying the longest common suffix takes only $O(M + N)$ time. The code within the triply nested loops, '**for** c, **for** k, **for** op', is repeated $\Theta(Cost^2)$ times because the '**for** c' loop is repeated *Cost* times, the '**for** k' loop is repeated $upper(c) + lower(c) + 1 \leq 2c + 1$ times (by Assumption A1), and the '**for** op' loop is repeated four times. With the exception of the innermost **while** loop, the code within the three **for** loops takes $O(1)$ time. Thus all portions of the algorithm except for this **while** loop consume $O(M + N + Cost^2)$ time.

To account for the time taken by the innermost **while** loop, charge each $O(1)$ iteration to the grid entry $(long + 1, long + k + 1)$ examined by the loop. We claim that a given entry is charged at most $\Delta = firstop(move) + firstop(replace)$ times. First observe that if $(i, i + k)$ is charged when computing $L(c, k, move)$, then the recurrence for $L(c, k, move)$ implies that $i \in [mbest(c, k) + 1, L(c, k, move) + 1]$. Now suppose $(i, i + k)$ is *first* charged when computing $L(d, k, move)$. Then

$$mbest(d + \Delta, k) \geq L(d + firstop(replace), k, replace) \geq L(d, k, move) + 1 \geq i$$

Thus $mbest(C, k) \geq i$ for all $C \geq d + \Delta$. But then entry $(i, i + k)$ cannot be charged when computing $L(C, k, move)$ for such C, since the converse would imply $mbest(C, k) < i$. This proves the claim. The algorithm computes L-values only over the range of diagonals $-lower(Cost)$ to $upper(Cost)$ and there are less than $(2Cost + 1)\min(M, N)$ grid points within this range. Since each grid point is charged $O(1)$ time (considering Δ to be a constant), it follows that $O(Cost \times (M + N))$ time is taken by the body of the innermost **while** loop.

Reconstructing a script from the $O(Cost^2)$ L-values computed by the greedy algorithm is analogous to the postprocess described for the dynamic programming algorithm and requires only $O(Cost)$ additional time. However, hierarchical screen updating methods [6] require just the costs of row replacements when evaluating interline editing strategies. These costs can be computed in only $O(Cost)$ space as follows. Observe that the value of $L(c, k, op)$ depends only on values $L(C, K, \theta)$ for which $C \geq c - MOP$ where $MOP = \max_{op \in OPS}(firstop(op))$. Thus the cost-only algorithm need only retain the L-values of the most recent MOP repetitions of the outer '**for** c' loop and this requires only $O(Cost)$ space (considering MOP to be a constant).

5. DISCUSSION AND OPEN PROBLEMS

Algorithmic Variations. Algorithms 2a, 2b, and 3b can be extended in several ways. Of the two, the dynamic programming approach is more 'robust' in the sense of extending easily to certain generalizations of the problem. For example, like Algorithm 1a of Wagner and Fischer, Algorithms 2a and 2b work without change for arbitrary real costs. On the other hand, Algorithm 3b, like Algorithm 1b, requires integer costs, and efficiency considerations demand that *perchar*(*delete*) and *perchar*(*insert*) be positive.

Both algorithms can be made more space-efficient. To construct an optimal script, Algorithms 2a, 2b, and 3b, as presented, require retention of $O(MN)$ and $O(Cost^2)$ arrays C and L, respectively. If Algorithm 2b's postpass checks are integrated into Algorithm 2a, as was done in Algorithm 3b, then Hirschberg's divide and conquer technique [9] computes the replacement script in only linear space. Most of the details are explained in [16], which gives a linear-space sequence comparison algorithm for block indels with affine costs. Similarly Myers' approach [14] for computing an edit script in linear space can be adapted for Algorithm 3b. However in both cases the necessary adaptations are complex and probably unnecessary, given the typical sizes of row replacement problems that arise with screen editors.

Efficiency Considerations. An editor's screen-updating module must be efficient. As text is typed, the module is invoked with every keystroke. With the algorithms developed in this paper, efficiency considerations are paramount. Tests under realistic conditions favor the greedy approach over dynamic programming, but a straightforward implementation of Algorithm 3b may be impractically slow. Fortunately both algorithmic (high-level) and implementation (low-level) efficiency gains are possible with the greedy algorithm.

At the algorithmic level, the screen operations *op* with startup cost 0 can be lumped together since *onechar*(θ, *op*) does not vary with θ. Thus, with ANSI-standard terminals, the restricted replacement scripts can be classified: (1) ends with *insert, (2)* ends with *move*, and (3) ends with *delete* or *replace*. That is, entries for *delete* and *replace* are coalesced into a single entry *delrep* for which $L(c, k, delrep) = \min(L(c, k, delete), L(c, k, replace))$. This reduces space requirements and the number of executions of the innermost '**for** op $\in$ OPS' loop by the factor 3/4.

The innermost **while** loop treating the zero *perchar* cost of *move* commands frequently visits a grid point as many as *firstop*(*move*) + *firstop*(*replace*) times. This can be reduced to one by maintaining a vector, $Far(k)$, equal to the maximum of $L(d, k, move)$ for all d less than the current c of the outermost loop. Initially set $Far(k)$ to *NONE* for all k and change Algorithm 3a as follows.

```
...
if op = move then
  if long ≤ Far(k) then
    long ← Far(k)
  else
    { while long < M and long + k < N and a_{long+1} = b_{long+k+1} do
        long ← long + 1
      Far(k) ← long
    }
else if op ≠ insert then
...
```

Essentially, $mbest(c, k)$ is nondecreasing in c and so using $Far(k)$ prevents the rediscovery of a sequence of zero-cost diagonal moves.

At the implementation level a number of code transformations are possible. The test '$K \in [-lower(C), lower(C)]$' can be avoided within the inner loop by explicitly setting $L(c, k, op)$ to *NONE* for $k \in [-lower(c + MOP), -lower(c) - 1] \cup [upper(c) + 1, upper(c + MOP)]$. The '**for** op' and '**for** θ' loops can be completely unrolled and the resulting code specialized by eliminating the now unnecessary tests on *op* and θ. Similarly the '**for** c' outer loop can be broken into a loop for values less than or equal to *MOP* and one for greater values. In the second copy of the loop, the code is specialized by eliminating all tests on c. By far the greatest low-level efficiency gains can be made by replacing the triple subscripting for $L(c, k, op)$ by pointer manipulations. In effect, this involves hand-coding of the strength-reduction optimizations that the compiler cannot detect.

Empirical Results. Nine row-replacement problems were solved by the programs:

S	implementation of the algorithm used by the *s* editor [10, pp. 290–298]
Dyna	straightforward implementation of Algorithms 2a and 2b
Greedy	straightforward implementation of Algorithm 3b
Hand	implementation of Algorithm 3b optimized as above

Separate measurements were taken for the ANSI and the IBM 3101 cost parameters, since cost parameters determine which scripts are optimal and affect the performance of the greedy method. Table V summarizes the results. Row labels have the following interpretations:

$M:N$	lengths of the input rows A and B.
$Script_s$	replacement script generated by *S*. Edit commands are indicated by letters *C* (*Clear*), *D* (*Delete*), *I* (*Insert*), *M* (*Move*) and *R* (*Replace*). For example, 'R^2DMI' denotes a script of the form '*Replace, Replace, Delete, Move, Insert*'.

Table V. Experimental Data for Row-Replacement Algorithms

	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8	Test 9
$M:N$	30:0	45:44	45:46	40:30	40:40	40:40	40:40	40:39	75:75
Script_S	C	MD	MI	$MR^{12}C$	R^{40}	MR^{30}	MR^{15}	$MR^{15}D$	MR^{74}
ANSI-standard:									
Script_{Opt}	C	MD	MI	$MR^{12}C$	R^{40}	R^2DMR^8	R^{18}	R^8DMR	$R^{42}MR^4$
$\text{Cost}_{Opt}:\text{Cost}_S$	3:3	11:11	17:17	23:23	40:40	21:38	18:23	20:26	53:82
Time_{Dyna}(ms)	1	198	196	89	157	157	156	152	542
Time_{Greedy}(ms)	<1	5	24	63	247	50	29	39	482
Time_{Hand}(ms)	<1	1	3	5	18	5	3	4	32
IBM 3101:									
Script_{Opt}	C	MD	MI	$MR^{12}C$	R^{40}	R^2DMR^8	R^2DMI	MDMR	$IR^8MR^3D^2MIMR^4$
$\text{Cost}_{Opt}:\text{Cost}_S$	2:2	6:6	7:7	18:18	40:40	11:34	11:19	11:21	37:78
Time_{Dyna}(ms)	1	192	192	86	154	155	156	152	544
Time_{Greedy}(ms)	<1	3	6	44	219	15	15	14	185
Time_{Hand}(ms)	<1	<1	1	3	14	1	1	1	13

$Script_{Opt}$	optimal replacement script.
$Cost_{Opt} : Cost_S$	length of the optimal script and of the script generated by S.
$Time_{Dyna}(ms)$	execution time of *Dyna* in milliseconds. The procedures were written in C and run on a DEC 8600 under the UNIX operating system.
$Time_{Greedy}(ms)$	execution time of *Greedy* in milliseconds.
$Time_{Hand}(ms)$	execution time of *Hand* in milliseconds.

A number of interesting facts can be extracted from this data, but two points deserve explicit mention. First, *Greedy* was more efficient than *Dyna*, especially for the 3101. Execution time for the greedy method depends on terminal cost parameters and improves with the 3101's shorter escape sequences. *Dyna's* execution is essentially independent of these costs. Moreover, hand optimization sped up the greedy algorithm by a factor of 12 for the ANSI model and 15 for the 3101.

Table V also indicates conditions where a hand-optimized greedy program can outperform a simple row-replacement procedure. A method's performance is determined by the time to compute the script and transmit it to the screen. Assuming that it takes a millisecond to transmit a character (e.g., at 9600 baud), *Hand* was superior to S whenever S failed to generate an optimal script. This occurred when the optimal script contained a noninitial *move* command. The superiority of *Hand* over S benefited from the 3101's economical *move*. (Table V was simplified by omitting S's execution times, which were always less than a millisecond, and by giving a single line of S's scripts, which happened to coincide for the two cost models on Tests 1–9.)

Open Problems and Future Work. The key assumptions in this paper are that the cursor begins in column 1 and that spilling characters off the end of a row is not a concern. While a script produced by our algorithms can be transformed into one that takes into account the current cursor location and avoids a spill, the transformed script is not guaranteed to be optimal. It appears very difficult, but still solvable in polynomial time, to compute an optimal script that takes into account these factors. Designing an efficient algorithm for this problem is a subject of current research.

Row replacement is just a subproblem in the ultimate goal of building an efficient, but autonomous, screen manager that optimally updates the entire screen. Optimality is sacrificed by the natural two-level approach that decides which lines to delete, insert, and replace at the top level, and that uses row replacement to appraise and perform replacements at the bottom level. Moreover, there are natural asymmetries between intra- and inter-row edits that resist a homogeneous approach. For example, unlike character insertion, the cost of a row insertion command is "symbol-dependent" since its cost is a function of the length of the row itself. This prevents the greedy algorithm of Section 4 from being applied to interline edits. It remains to be seen whether an optimal "screen replacement" algorithm of satisfactory efficiency can be developed.

The change from terminals and serial communications to workstations and high-speed networks does not make the screen-update problem disappear, though different techniques are needed to minimize the cost of *bitblt* operations [8]. Pike

[17] discusses a simple approach for screen updating by a text editor, while [3] treats display-updating problems in computer animation. However, the problem of optimally updating a bitmap display is not well understood.

What is the most general class of sequence comparison problems that can be solved with the greedy approach? Algorithm 3b solves, as a special case, the problem for block indels with affine gap penalties, which is important in molecular biology. The algorithm can be extended to handle symbol-dependent replacement costs, which are used for comparing proteins. However, the exact limits of the greedy algorithm remain to be determined.

ACKNOWLEDGMENTS

The authors thank the referees for numerous suggestions that improved this presentation.

REFERENCES

1. ARNOLD, K. C. R. C. Screen updating and cursor movement optimization: a library package. In *UNIX Programmer's Manual: Supplementary Documents (4.2BSD)*, University of California, Berkeley, 1984.
2. BARACH, D., TAENZER, D., AND WELLS, R. The design of the PEN video editor display module. In *Proceedings of the ACM Symposium on Text Manipulation* (Portland, Ore., June 8–10, 1981). *SIGPLAN Not. 16*, 6 (1981), 130–136.
3. DENBER, M. J., AND TURNER, P. M. A differential compiler for computer animation. *Comput. Graph. 20*, 4 (1986), 21–27.
4. FICKETT, J. W. Fast optimal alignment. *Nucleic Acids Res. 12*, 1 (1984), 175–179.
5. FITCH, W. M., AND SMITH, T. F. Optimal sequence alignments. In *Proceedings National Academy of Science USA 80* (1983), 1382–1386.
6. GOSLING, J. A redisplay algorithm. In *Proceedings of the ACM Symposium on Text Manipulation* (Portland, Ore., June 8–10, 1981). *SIGPLAN Not. 16*, 6 (1981), 123–129.
7. GOTOH, O. An improved algorithm for matching biological sequences. *J. Molec. Biol. 162* (1982), 705–708.
8. GUIBAS, L. J., AND STOLFI, J. A language for bitmap manipulation. *ACM Trans. Graph. 1*, 3 (1982), 191–214.
9. HIRSCHBERG, D. S. A linear space algorithm for computing maximal common subsequences. *Commun. ACM 18*, 6 (1975), 341–343.
10. MILLER, W. *A Software Tools Sampler*. Prentice-Hall, Englewood Cliffs, N.J., 1987.
11. MILLER, W., AND MYERS, E. W. A file comparison program. *Softw. Pract. Exper. 15*, 11 (1985), 1025–41.
12. MILLER, W., AND MYERS, E. W. Sequence comparison with concave weighting functions. *Bull. Math. Biol. 50*, 2 (1988), 97–120.
13. MILLER, W., AND MYERS, E. W. A simple row replacement method. *Softw. Pract. Exper. 18*, 7 (1988), 597–612.
14. MYERS, E. W. An $O(ND)$ difference algorithm and its variants. *Algorithmica 2*, 1 (1986), 251–266.
15. MYERS, E. W. Incremental alignment algorithms and their applications. To appear in *SIAM J. Comput.* See also TR86-22, Dept. of Computer Science, Univ. of Arizona, Tucson.
16. MYERS, E. W., AND MILLER, W. Optimal alignments in linear space. *Comput. App. Biosci. 4*, 1 (1988), 11–17.
17. PIKE, R. The text editor *sam*. *Softw. Pract. Exper. 17*, 11 (1987), 813–845.
18. STALLMAN, R. M. EMACS—the extensible, customizable self-documenting display editor. In *Proceedings of the ACM Symposium on Text Manipulation* (Portland, Ore., June 8–10, 1981). *SIGPLAN Not. 16*, 6 (1981), 147–157.
19. UKKONEN, E. Algorithms for approximate string matching. *Inf. Contr. 64*, 1–3 (1985), 100–118.

20. WAGNER, R. A., AND FISCHER, M. J. The string-to-string correction problem. *J. ACM 21*, 1 (1974), 168–173.
21. WATERMAN, M. S. General methods for sequence comparisons. *Bull. Math. Biol. 44*, 4 (1984), 473–500.
22. WATERMAN, M. S., SMITH, T. F., AND BEYER, W. A. Some biological sequence metrics. *Advances in Math. 20* (1976), 367–387.
23. WEISER, M. CWSH: The windowing shell of the Maryland Window System. *Softw. Pract. Exper. 15*, 5 (1985), 515–519.
24. WOOD, S. R. Z—the 95% program editor. In *Proceedings of the ACM Symposium on Text Manipulation* (Portland, Ore., June 8–10, 1981). *SIGPLAN Not. 16*, 6 (1981), 1–7.

Received July 1986; revised November 1987 and August 1988; accepted August 1988