SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 18(7), 597–611 (JULY 1988)

A Simple Row-replacement Method

WEBB MILLER
Department of Computer Science, The Pennsylvania State University, University Park, PA 16802, U.S.A.

AND

EUGENE W. MYERS
Department of Computer Science, University of Arizona, Tucson, AZ 85721, U.S.A.

SUMMARY

Updating a video screen involves row replacement, i.e. the task of updating an existing screen row to produce the desired row. In many environments, screen operations require transmitting characters to the terminal by a process that is painfully slow compared to computing speeds. Thus, it is worth while to compute a minimal set of row updating commands, as long as the time to do so does not outweigh the savings in character transmission time. This paper presents a simple and practical algorithm for optimal row replacement and describes experience with its use in a screen editor.

KEY WORDS Row replacement Video terminal Screen editor

INTRODUCTION

At the heart of many modern screen editors and other screen-oriented programs, lies the 'screen manager' module, which is responsible for keeping the video display up to date. In one design, procedures that modify the buffer (the editor's copy of the edited file) must also inform the screen manager how to update the display. A cleaner approach lets an autonomous screen manager determine how to update the screen by comparing its record of the screen contents with the current buffer contents. The interface to the screen manager is then a single routine, refresh, that synchronizes the screen with the buffer. It is given no information other than the contents of the screen and the buffer. An autonomous screen manager gives the editor an orderly internal structure, and writers of screen editors are almost unanimous in recommending this approach.[1–3]

Compared to the other parts of an editor, the screen manager may well involve both the messiest implementation details and the deepest algorithmic issues. The implementation details involve idiosyncrasies of various terminal brands.[4] Such difficulties are not considered here.

This paper is concerned with one aspect of the following algorithmic problem: given the current screen contents and the desired screen contents, determine a set of screen updating commands having minimum cost. Often, data travels from the editor to the

0038–0644/88/070597–15$07.50

Received 23 December 1986
Revised 16 December 1987

screen at between 100 and 1000 characters per second (e.g. 1200 or 9600 baud), which is orders of magnitude slower than the editor's computing speed. Such conditions justify the use of a relatively complex scheme to minimize the number of bytes sent to the screen, so long as the savings in character transmission time are not outweighed by the execution time. Execution time is a critical issue because the desirable properties of an independent screen manager are achieved at a cost in execution efficiency. The screen manager operates without knowledge of the user's edit command, so it cannot take short-cuts tailored to particular commands. For example, when typed characters are being inserted in the buffer, an autonomous screen manager applies the updating algorithm for every keystroke, and a character-by-character comparison of two 24×80 character arrays (the current screen contents and the desired screen contents) may be too slow.

Video terminals treat rows and columns differently. Commands usually exist for inserting or deleting rows, but not columns, and printing or inserting a character moves the cursor along a row, but not down a column. It is not surprising, then, that screen-update procedures often begin by partitioning the problem into two levels. At the top level is the decision of which rows to delete, insert, replace or leave unchanged; at the bottom level, a row-replacement algorithm evaluates the cost of each replacement and may then, under control of the top level, perform the replacements. The top-level techniques for reusing entire screen rows are often quite different from the bottom-level methods for reusing parts of a row. For the top-level problem, both simple[1, 3] and algorithmically sophisticated[2, 5] approaches have been used.

In the traditional sequence-comparison model,[6] which has been used for solving the toplevel update problem,[2] the only editing operations permitted are Delete, Insert and Replace, and the cost of a script of edit operations is the sum of the individual operations' costs. The low-level row-replacement problem is not captured by the traditional sequence-comparison model because of:

1. *Non-additive costs*. Screen-update operations commonly violate the assumption that the cost of operating on k symbols is k times the cost of operating on one symbol. For example, on ANSI-standard terminals it costs nine bytes to insert one character, ten to insert two, eleven to insert three, and so on. The eight-byte fixed cost comes from a four-byte command to put the terminal in 'insert mode' and another four-byte command to return to normal mode.
2. *Cursor movement costs*. With intra-line editing, cursor movement commands often dominate the editing cost, and traditional models, which ignore cursor position, can be misleading. For example, consider the problem of replacing the row *xaxaxaxax* with *yayayayay*. On typical terminals, repeatedly moving the cursor to the next x and replacing it with y is decidedly inferior to simply replacing the entire row.
3. *The richer command repertoire*. Intra-line editing can use what is here called the Clear command (see next section). Also, a row's last character can be deleted by printing a blank over it. Other such possibilities exist, and classical string edit models fail to encompass them.

Sequence-comparison techniques have been adapted to give two 'optimal' row-replacement algorithms.[7] 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 to convert one row to another. Both algorithms

account for non-additive costs, the cost of cursor movement, and the use of such 'non-classical' operations as Clear. The first algorithm is an $O(MN)$ dynamic programming algorithm, where M and N are the lengths of the original and final rows. The second algorithm runs in time $O(C\times(M+N))$. Here C is the optimal cost of replacement, so the algorithm is fast when the required update is small.

This paper develops a simple row replacement algorithm and shows it to be optimal under certain assumptions. The algorithm's asymptotic worst-case time complexity is $O(M^2N^2)$. However, in practice, the method is often very fast and its *average* time complexity is superior to the earlier methods.[7] Its use in a screen editor is described. The method is intended for screen updating when terminal control sequences are transmitted over slow (e.g. 9600 baud) communication lines, not for memory-mapped video terminals. It is appropriate for updating a display region that extends to the right margin of the screen; it may be inappropriate for a 'window' lying to the left of another window, since inserting and deleting characters affects the remainder of the screen row. The method is closely related to sequence-comparison techniques used in molecular genetics, as outlined at the end of the paper.

THE ROW-REPLACEMENT PROBLEM

This paper assumes the following row-modification commands, which are provided by most modern terminals. Each operation is performed by sending an appropriate stream of characters to the terminal.

1. Clear. The characters at, and to the right of, the cursor are deleted. The cursor does not move.
2. Delete k. The k characters beginning at the cursor are deleted, causing later characters to shift left. The cursor does not move, so it ends up on the character that follows the deleted characters.
3. Insert ⟨string⟩. The given character string is inserted at the cursor's location, causing later characters to shift right. The cursor moves right the length of the string, so it stays on the same character.
4. Move k. The cursor is moved k columns from its current position. This paper limits use of the Move k command to $k > 0$, i.e. movement to the right.
5. Print ⟨string⟩. The given character string is displayed beginning at the cursor's location, overwriting previously displayed characters. The cursor moves to the position immediately following the printed string.

A set of such commands is called an *LR script* because the commands move the cursor from left to right. This paper is concerned with generating an LR script of minimum cost that converts a given string, A, of length M to another given string, B, of length N.

The cost of applying the edit operation op to k characters is assumed to be cost(op k) = startup(op) + k ×perchar (op), where startup(op) and perchar(op) are non-negative real numbers. The start-up cost is paid for the first operation in a script and whenever an operation differs from the previous operation. For example, Move 2, Print ab, Print cd, Delete 1 costs cost(Move,2) + cost(Print,4) + cost(Delete,1). In practice, startup and perchar are usually integers indicating the number of bytes that must be transmitted to the screen. However, our discussion does not depend on these values being whole numbers.

The following table gives cost parameters for ANSI-standard terminals and the IBM 3101. Entries are based on the number of characters sent to the screen. For instance, the ANSI Clear command is the three-character sequence ⟨escape⟩[K, so cost(Clear, k) = 3 for all k or, equivalently, startup(Clear) = 3 and perchar(Clear) = 0.

Table I simplifies cost accounting somewhat. On ANSI-standard terminals, a Move instruction sends between 6 and 8 bytes to the screen, depending on the target address. Moreover, it is often possible to delete characters or move the cursor in insert mode, so our cost model may overestimate costs by charging for unnecessary mode changes.

The algorithm presented below rests on the following additional, but realistic, assumptions.

A1: perchar(Move) < perchar(Print)

A2: If Move commands are not allowed, then the conversion of $a_1a_2 \ldots a_K$ to an equal-length string $b_1b_2 \ldots b_{K\_}$ is performed optimally by printing $b_1b_2 \ldots b_K$. (The underscore indicates cursor position.)

We do not know of any terminals that violate either assumption. If A1 did not hold, the Move command would not be useful for cursor motion to the right; printing characters over themselves would work at least as well. Assumption A2 is made for reasons of visual aesthetics. We would rather print B over A instead of, say, deleting A and inserting B. Our algorithm relies on A2 being true, but unlike A1 it is not essential. Simple additions to the procedures nontail and overwrite (see the next section) would remove the algorithm's reliance on A2.

ROW REPLACEMENT WITHOUT Move COMMANDS

As shown in the next section, the key to determining an optimal replacement strategy is deciding when to use Move commands. However, before the first Move, between Move commands, or after the final Move, a subproblem of the original row-replacement problem must be optimally solved without a Move. This section shows that an optimal LR script is easy to determine if only Clear, Delete, Insert and Print are allowed. The next section treats Move operations.

An operation of an LR script begins with the cursor and partly-edited string in a configuration such as

$$b_1b_2 \ldots b_j\underline{a_{i+1}}a_{i+2} \ldots a_M$$

Table I

	ANSI startup	ANSI perchar	3101 startup	3101 perchar
Clear	3	0	2	0
Delete	0	3	0	2
Insert	8	1	0	3
Move	8	0	4	0
Print	0	1	0	1

To the left of the cursor is a prefix of the desired row, B; to the right is a suffix of the original row, A. Thus, the previous operations edit $a_1a_2 \ldots a_i$ into $b_1b_2 \ldots b_j$ and leave the cursor positioned just after b_j. We say that the previous operations *edit to* configuration (i,j) and that the command *begins at configuration* (i,j). An LR script *edits from configuration* (i,j) *to configuration* $(I,\mathcal{J})$ if it transforms

$$b_1b_2 \ldots b_j a_{i+1}a_{i+2} \ldots a_M$$

to

$$b_1b_2 \ldots b_j a_{I+1}a_{I+2} \ldots a_M$$

Case 1: nontail replacement

Consider editing from configuration (i,j) to configuration $(I,\mathcal{J})$ using an LR script without a Move operation, where $i \leq I < M$ and $j \leq \mathcal{J} < N$. Since a Clear command would annihilate $a_{I+1}a_{I+2} \ldots$, none can be used. The following procedure determines the cost of an optimal script and, if the procedure argument script is true, it generates an optimal script. It prints as many characters of $b_{j+1}b_{j+2} \ldots b_{\mathcal{J}}$ as will fit over $a_{i+1}a_{i+2} \ldots a_I$, then performs a Delete or Insert operation, as appropriate.

```
nontail(i,j,I,J, script)
  {  total ← 0
     k ← min(I–i,J–j)
     if k > 0 then
       {  if script then transmit 'Print b[j+1]b[j+2]...b[j+k]'
          total ← total + cost(Print, k)
          i ← i + k
          j ← j + k
       }
     if I–i > 0 then
       {  if script then transmit 'Delete I–i'
          total ← total + cost(Delete, I–i)
       }
     else if J–j > 0 then
       {  if script then transmit 'Insert b[j+1]b[j+2]...b[J]'
          total ← total + cost(Insert, J–j)
       }
     return total
  }
```

Case 2: tail replacement

Consider editing from configuration (i,j) to a 'final' configuration, i.e. $b_1b_2 \ldots b_N$ where the cursor position is immaterial. To generate an LR script that is optimal when Move operations are not allowed, it is adequate to evaluate the following two strategies and pick the more economical.

The first strategy applies the optimal non-tail replacement approach where $(I,\mathcal{J})$

corresponds to the longest common suffix of $a_{i+1}a_{i+2}. \ . \ .a_M$ and $b_{j+1}b_{j+2}. \ . \ .b_N$. This strategy is at least as economical as any replacement method that leaves some of the characters $a_{i+1}a_{i+2}. \ . \ .a_M$ untouched.

The second strategy simply overwrites $a_{i+1}a_{i+2}. \ . \ .$ by printing $b_{j+1}b_{j+2}. \ . \ .b_M$ and, if necessary, removes excess characters with Clear or Delete, whichever is more economical. No replacement method that touches (with Clear, Delete or Print) every character $a_{i+1}a_{i+2}. \ . \ .a_M$ can be more economical. (If the model allowed deleting a final character by printing a blank over it, then the procedure would have to be modified slightly.)

```
tail(i,j,script)
  {  K ← length of the longest common suffix of a_{i+1}a_{i+2}. . .a_M and b_{j+1}b_{j+2}. . .b_N
     nont ← nontail(i, j, M–K, N–K, false)
     over ← overwrite(i, j, false)
     if script then
       if nont ≤ over then nont ← nontail(i, j, M–K, N–K, true)
       else over ← overwrite(i, j, true)
     return min (nont, over)
  }

overwrite(i,j,script)
  {  total ← 0
     if j < N then
       {  if script then transmit 'Print b_{j+1}b_{j+2}. . .b_N'
          total ← total + cost(Print, N–j)
       }
     k ← (M–i) – (N–j)
     if k > 0 then
       if cost(Clear, k) ≤ cost(Delete, k) then
        {  if script then transmit 'Clear'
           total ← total + cost(Clear, k)
        }
       else
         {  if script then transmit 'Delete k'
            total ← total + cost(Delete, k)
         }
     return total
  }
```

USING Move COMMANDS

A Move k command can begin at configuration (i,j) if

$$a_{i+1} = b_{j+1}, a_{i+2} = b_{j+2}, \ldots, a_{i+k} = b_{j+k}$$

i.e. if A and B have a common substring of length k beginning at position $i+1$ of A and position $j+1$ of B. The *jump potential* of configuration (i,j) is the smallest $k \geq 0$ such that either $i+k \geq M$, $j+k \geq N$, or $a_{i+k+1} \neq b_{j+k+1}$. Thus, Move k is permissible if and only if k does not exceed (i,j)'s jump potential. A longer jump would pass $a_{i+\text{jump\_potential}+1}$, which could then not be changed to $b_{j+\text{jump\_potential}+1}$ by an LR script.

Let *min_jump* be the smallest k such that cost(Move, k) < cost(Print, k). Assumption A1 guarantees that *min_jump* exists. More formally,

$$min\_jump = \min\left[k : k \text{ is an integer and } k > \frac{\text{startup(Move)}-\text{startup(Print)}}{\text{perchar(Print)}-\text{perchar(Move)}}\right]$$

For example, with the ANSI and 3101 cost parameters given in Table I, *min_jump* equals 9 and 5, respectively. If k < *min_jump*, then a Move k operation can be performed at no more cost by simply printing the k characters over themselves, and hence need not be considered.

A *jump-off configuration* for strings A and B is a configuration (i,j) where the jump potential is at least *min_jump* and either $i=0$, $j=0$ or $a_i \neq b_j$. Reasoning as in the previous section solves the problem of optimally editing from one jump-off configuration to another with a Move followed by non-Move operations. The general strategy reduces to Move as far as possible, then Print as much as possible, and then perform the appropriate Clear, Delete or Insert operation (if any).

The problem of computing an optimal LR script can be formulated in graph-theoretical terms as follows. The nodes of the *jump-off graph* consist of the jump-off configuration together with special *initial* and *final* nodes. Edges and their weights are determined by the following rules.

1. There is an edge from the initial node to every other node, but no edge into the initial node. The edge from the initial node to jump-off configuration $(I,\mathcal{J})$ has weight nontail(0,0,I,$\mathcal{J}$). The edge from the initial node to the final node has weight tail(0,0).
2. If (i,j) and $(I,\mathcal{J})$ are jump-off configurations, then there is an edge from (i,j) to $(I,\mathcal{J})$ whenever $I \geq i$ + *min_jump* and $\mathcal{J} \geq j$ + *min_jump*. The edge weight is computed by the procedure

```
jump_to_nontail(i, j, I, J, script)
  { k ← min(I–i, J–j, jump potential of (i, j))
    if script then transmit 'Move k'
    return cost(Move, k) + nontail(i+k, j+k, I, J, script)
  }
```

3. No edge begins at the final node, but there is an edge from every other node to the final node. The weight of an edge from jump-off configuration (i,j) to the final node is computed by the procedure:

```
jump_to_tail(i, j, script)
  { K ← length of the longest common suffix of a_{i+1}a_{i+2}...a_M and
    b_{j+1}b_{j+2}...b_N
    nont ← jump_to_nontail(i, j, M–K, N–K, false)
    over ← jump_to_overwrite(i, j, false)
    if script then
      if nont ≤ over then nont ← jump_to_nontail(i, j, M–K, N–K, true)
      else over ← jump_to_overwrite(i, j, true)
    return min(nont, over)
  }
```

```
jump_to_overwrite(i, j, script)
  { k ← jump potential of (i,j)
    if script then transmit 'Move k'
    return cost(Move, k) + overwrite(i+k, j+k, script)
  }
```

The row replacement problem for strings A and B is equivalent to finding a shortest path through the jump-off graph, so an algorithm such as Dijkstra's[8] can be applied. The set of all paths from *initial* to *final* models the set of LR scripts converting A into B consisting of Move commands separated by non-Move command subsequences such that (a) the non-Move portions are optimal for the substrings they edit, (b) all Moves are of length at least *min_jump*, and (c) all Moves begin at jump-off configurations.

As argued earlier, the search for an optimal LR script can be limited to scripts that satisfy (a) and (b). As discussed at the end of this section (and rigorously verified in Reference 9), for any strings A and B there exists an optimal LR script in which every Move operation begins at a jump-off configuration. Thus, at least one optimal LR script is modelled by the jump-off graph, and clearly corresponds to a path of minimum length.

In summary, our row-replacement algorithm is as follows:

1. Determine the nodes of the jump-off graph. (See the next section for an efficient approach.)
2. Find a shortest path from *initial* to *final* using Dijkstra's algorithm, terminating as soon as the minimum distance to *final* is known. Calls to nontail, tail, jump_to_nontail and jump_to_tail compute edge weights but do not generate update instructions, i.e. the procedure argument script is false. Each edge is inspected at most once, so the edge weights can be computed just as needed, then discarded.
3. 'Walk' the shortest path from *initial* to *final* by performing the appropriate sequence of calls to nontail, tail, jump_to_nontail and jump_to_tail. In this pass, the calls generate screen-update instructions, i.e. script is true.

Consider, for example, editing A = *abcdefaabcdef* into B = *bcdefabcde* given the cost parameters

cost(Clear, k)	3
cost(Delete, k)	$2k$
cost(Insert, k)	$2 + k$
cost(Move, k)	3
cost(Print, k)	k
min_jump	4

Figure 1 gives the jump-off graph. Nodes are represented by ○. The initial and final nodes are at the extreme upper left and extreme lower right, respectively. The jump potential of a node is indicated by the •s trailing down and to the right from the node; the •s show the configurations that can be reached from the node by a single Move operation.

The edge from the initial node to the jump-off configuration (1,0) has weight 2, corresponding to the script generated by nontail(0,0,1,0), i.e.

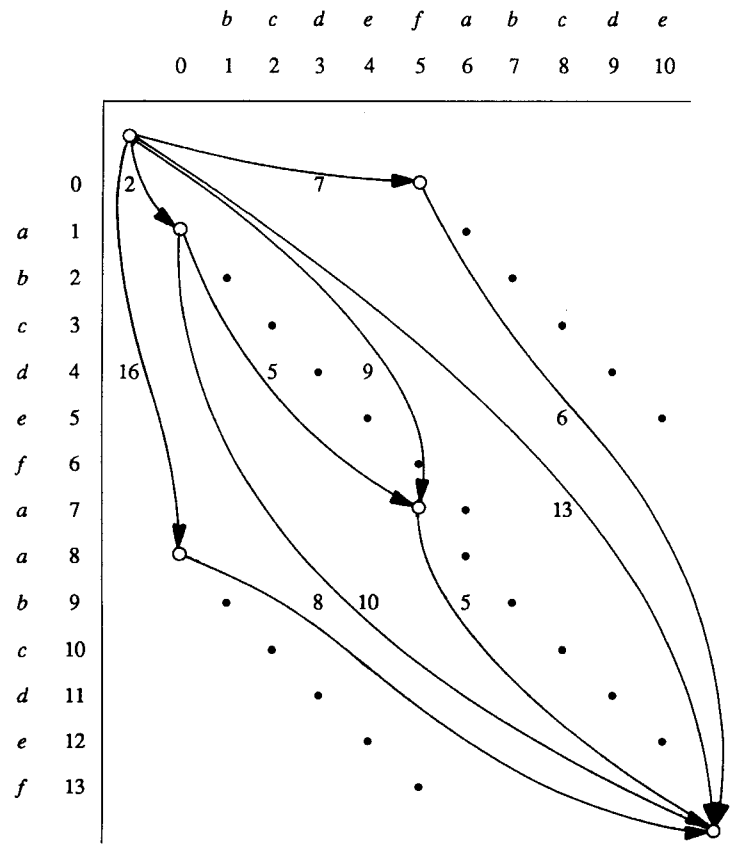

Figure 1. Jump-off graph for A = *abcdefaabcdef and* B = *bcdefabcde*

Delete 1

The edge from (1,0) to (7,5) is weighted by the cost of editing the jump-off configuration

bcdefaabcdef

into the jump-off configuration

bcdefabcdef

using jump_to_nontail(1, 0, 7, 5) = 5. The corresponding LR script is

Move 5
Delete 1

Similarly, the edge from (7,5) to the final node has weight jump_to_tail(7,5) = 5, corresponding to the script

```
Move 5
Delete 1
```

In the above example, shortest paths from the initial node to the final node have cost 12. One such path passes through nodes (1,0) and (7,5), so concatenating the above three scripts gives an optimal solution. Longer paths from the initial node to the final node correspond to more expensive ways of editing A to B. For example, taking the single edge from the initial node to (7,5) and stepping to the final node corresponds to the script

```
Print bcdef
Delete 2
Move 5
Delete 1
```

which costs 14.

It takes some effort to see why there must always be an optimal LR script in which every Move operation begins at a jump-off configuration. An arbitrary optimal LR script editing A to B may apply Move operations at other configurations. However, such a script can be transformed to one conforming to the principle without increasing its cost. A rigorous proof of this fact is given in Reference 9, but the following example, based on Figure 1, gives the basic idea. The script Delete 1, Move 6, Delete 1, Move 4, Delete 1 costs 12, but the second Move begins from configuration (8,6), which is not a jump-off configuration. In terms of the picture, the first Move jumps as far as possible, (i.e. to the last • in the chain trailing down and to the right from (1,0)). Then the second Delete drops down to another chain, but it reaches a • and not the leading ○. However, at no extra cost we can shorten the first Move by one and lengthen the second Move by one, so that the Delete will reach the ○.

EXPERIENCE

The screen editor s^3 uses the row-replacement strategy of positioning the cursor on the first mismatched character (just past the longest common prefix of the old and new rows), then applying the tail procedure given above. The only use of Move is to reach the first mismatched character, if necessary.

The new row-replacement algorithm was incorporated into s without perceptibly degrading its performance. The algorithm's poor worst-case behaviour (i.e. for rows of length N the jump-off graph can have $O(N^4)$ edges) has not been a problem. If s ever attempts a row replacement with more than 100 jump-off configurations (this has yet to happen), then only the first 100 constructed jump-off configurations are entered into the jump-off graph and a suboptimal update may result. The following crude probabilistic argument illustrates why the number of jump-off configurations is small. Suppose A is the result of M Bernoulli trials, where ρ is the probability that two successive characters are equal (ρ = 1/(the alphabet size) if characters are equally likely). Suppose B is obtained from A by deleting and inserting blocks of symbols at

k distinct locations. Then, there are at most $k+1$ jump-off configurations induced by the transformation and at most $MN(\rho^{min\_jump})(1-\rho))$ that occur by chance. This latter quantity is negligible for reasonable values of *min_jump* and alphabet size.

The following algorithm fragment shows how we use the initial cursor position, init_pos. If the cursor is not initially on the row (e.g. when other rows have been modified in the current screen update), then init_pos = ∞.

```
len ← length of the longest common prefix of A and B
if len ≥ init_pos – 1 then
  offset ← init_pos – 1
else
  { issue a preliminary 'Move to column len+1' command (a non-LR move)
    offset ← len
  }
remove the first offset characters from A and B
solve the row-replacement problem with the truncated A and B
```

If init_pos were ignored, then *aax* might be transformed to *ax* by moving the cursor to the second *a* (the first mismatched character), deleting the *a*, and moving the cursor back.

The implementation process was complicated by the fact that *s* folds long lines of the edited file onto several screen rows to make all characters visible. The row-replacement algorithm is applied to entire lines and the generated update commands are interpreted in light of line-folding. With 80-character screen rows, for example, Move to column 100 would be taken as referring to column 20 of the line's second segment. We have discovered row-replacement algorithms that directly model fixed row length, generate optimal command sequences and run in polynomial time, but their practical value remains to be demonstrated.

To guarantee that minor row updates are handled as efficiently as with *s*'s original row-replacement technique, the following approach was adopted. Define

$$\text{thresh} = \text{cost(Move, min\_jump)} + \min\{\text{cost(op,1)} : \text{op} \neq \text{Move}\}$$

That is, thresh gives the minimum cost of a useful Move plus another command. When replacing a row, *s* first computes C = tail(0,0), the cost of optimal row replacement without Move commands. If C $\leq$ thresh, then the general replacement procedure is not applied since any update sequence with a Move command costs at least C. Hence, the script generated by *s* is optimal in this case. In particular since thresh $\geq$ cost(Insert,1) and thresh $\geq$ cost(Print,1), jump-off configurations are not computed while text is simply being inserted directly into the buffer.

A considerable portion of our effort to develop a practical row-replacement technique was devoted to efficient computation of the jump-off configurations and their jump potentials. Construction of the jump-off graph dominates the execution time of the row-replacement algorithm, so the efficiency of this part of computation is essential. Let $\mathcal{J}$, M and N denote the number of jump-off configurations and the two row lengths, respectively. Jump-off configurations can be computed in time $O(\mathcal{J} + M + N)$ by a straightforward application of suffix trees.[10] With some effort, we refined the method to also compute jump potentials within the same time bound. However, the method's

performance in practice was unsatisfactory.

Although not asymptotically as efficient as suffix trees, a simple hashing scheme proved superior for a wide range of uses. With hashing, the row-replacement algorithm runs in time $O(\mathcal{J}P + M + N)$ in expectation where $\mathcal{J}P$ is the sum of the jump potentials of all jump-off configurations. The average 80 by 80 problem takes 0·8ms on a VAX 8600 running Berkeley 4.3bsd UNIX. In the worst case when $\mathcal{J}P$ is of order MN (e.g. a string of 80 xs versus itself), 2·6ms are required. Since the row replacement algorithm's execution is dominated by this phase, execution time roughly equals the time for transmitting one character to the screen at 9600 baud. The Appendix lists the code for constructing jump-off graphs, and Reference 9 gives a further discussion and analysis.

CONCLUSION AND AN OPEN PROBLEM

In some respects, the subject of screen updating is reminiscent of code optimization, i.e. the generation of optimal — or at least good — object code by a compiler. In both areas there are theoretical results giving algorithms for generating optimal instruction sequences. (For code optimization there are many such results; for screen updating, only a few.) Although these theorems may fail to model all intricacies of actual hardware, they can provide insights that are quite useful in practice.[11] However, the engineering trade-offs faced during implementation may mean that the most cost-effective optimizations are some of the simplest.[12] The method proposed in this paper is one such simple method of practical value.

The row-replacement method developed above is quite similar to techniques used for comparing biological sequences. A number of biosequence programs begin by determining all common substrings of length at least k (commonly $k = 2$ for proteins and $k = 5$ for DNA sequences). The use of suffix trees has been proposed,[13] but hashing has been the method of choice.[14] To compare two sequences, several of these programs[15, 16] compute an optimal path in a graph whose nodes are common substrings. (Biologists formulate the sequence comparison problem in such a way that the problem is to compute a *longest* path.) However, in this context the computed results are not necessarily optimal. For example, matches of length less than k are generally missed and overlapping segments (e.g. jump-off configurations (1,0) and (7,5) of Figure 1) are not treated optimally. However, the method is very fast, especially when heuristic rules are applied to reduce the number of common substrings placed in the graph.[16] A popular use of this approach[17, 18] is for rapid comparison of one sequence against an extensive sequence library; promising library sequences are then compared to the probe sequence using a more sensitive technique.

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.[19] Work has been done on simple approaches for screen updating by a text editor[20] and display updating in computer animation,[21] but the problem of optimally updating a bit-map display remains open.

ACKNOWLEDGEMENTS

We thank Bonnie Lynn Webber and the referees for suggestions that improved the presentation of this paper. The work of Eugene W. Myers was supported in part by the U.S. National Science Foundation under Grant DCR–8511455.

APPENDIX

For sequences A, B and given value of *min_jump*, the following C program constructs the nodes of the jump-off graph. As presented here, triples (i, j, min_jump) are printed. In a screen editor, the code would store the triples, add an *initial* and a *final* node, and apply a shortest-path algorithm that calls nontail, tail, jump_to_nontail and jump_to_tail for edge weights.

```
/* Construct nodes of the jump_off graph.
*
* A substring of length min_jump is hashed to an integer, hsum. Hsum is a
* weighted sum of the substring's characters, where the i-th character from
* the right is multiplied by i. Hsum can be computed by a few additive
* operations from the preceding substring's hsum. Hash[hsum] points to a linked
* list of POS_LISTs, one POS_LIST for each key, where a position's key is the
* immediately preceding character. A POS_LIST is a list of positions in A with
* identical hsum and key. */

#define JMPMAX 10                    /* limit on min_jump */
#define STRMAX 100                   /* limit on |A| - min_jump + 1 */
#define HSIZE  64*JMPMAX*(JMPMAX+1) /* size of hash table */

typedef struct P { char *pos; struct P *nxt; } POSITION;
typedef struct L { int key; POSITION *val; struct L *lft;} POS_LIST;

POSITION p_pool[STRMAX];             /* pool of POSITIONs */
POS_LIST l_pool[STRMAX];             /* pool of POS_LISTs */
POS_LIST *hash[HSIZE];               /* hash buckets */
int used[STRMAX], *utop = used;      /* stack of used buckets */
int jump = -1, jmpmult[128];         /* multiples of min_jump */

/* entry point - compute jump_off configurations and their jump potentials */

jump_off(A, B, min_jump) char *A, *B; int min_jump;
{  int mults, i;

   if (min_jump != jump) {
      jump = min_jump;
      for (mults = -jump, i = 0; i < 128; i++)
         jmpmult[i] = (mults += jump);
   }
   Ascan(A);       /* build table of positions in A */
   Bscan(A, B);    /* generate jump-off configurations */
}

static Ascan(A) char *A;
{  register POSITION *ptop;
   register POS_LIST *t, **v, *ltop;
   register char *s, *p;
   int csum, hsum, prev_char;

   while (utop > used)      /* reinitialize hash[] */
      hash[*(--utop)] = 0;
   ptop = p_pool;
   ltop = l_pool;

   prev_char = hsum = csum = 0;
   p = (s = A) + jump;
   while (s < p)
```

```
      hsum += (csum += *s++);
   p = A;
   for ( ; ; ) {   /* for positions in A, find position
                      list with same (hsum, prev_char) */
      for (t = *(v=hash+hsum); t!=0 && t->key<=prev_char; t = *(v = &(t->lft)))
         if (t->key == prev_char) {
            ptop->nxt = t->val;
            ptop->pos = p;
            t->val = ptop++;
            break;
         }
      if (t == 0 || t->key > prev_char) {   /* list not found */
         *(utop++) = hsum;      /* mark bucket as used */
         ptop->nxt = 0;         /* start a new list for the */
         ptop->pos = p;         /* the pair (hsum, prev_char) */
         ltop->val = ptop++;
         ltop->key = prev_char;
         ltop->lft = t;
         *v = ltop++;
      }
      if (*s == '\0') break;
      csum += *s++ - (prev_char = *p++);
      hsum += csum - jmpmult[prev_char];
   }
}

static Bscan(A, B) char *A, *B;
{  register POSITION *h;
   register POS_LIST *t;
   register char *m, *n, *p, *s;
   int csum, hsum, prev_char, N;

   hsum = csum = 0;
   p = (s = B) + jump;
   while (s < p)
      hsum += (csum += *s++);
   p = B;
   prev_char = 1;
   N = strlen(B);
   B[N] = '\01';
   for ( ; ; ) {     /* for positions in B */
      for (t = hash[hsum]; t != 0; t = t->lft)
         if (t->key != prev_char)

            /* found a list of positions with identical
               hsum but different preceding character. */

            for (h = t->val; h != 0; h = h->nxt) {

               /* resolve hash collisions and compute jump
                  potential four characters at a time. */

               if (*((int *)(m=h->pos)) == *((int *)(n=p))) {
                  while (*((int *)(m+=4)) == *((int *)(n+=4)))
                     ;
                  if (*(m-=3) == *(n-=3))
                     while (*++m == *++n)
                        ;
               } else if (*m == *n)
                  while (*++m == *++n)
```

```
                    ;
                if (n >= s)
                    printf("(%2d,%2d,%2d)\n",h->pos-A,p-B,n-p);
            }
        if (*s == '\01') break;
        csum += *s++ - (prev_char = *p++);
        hsum += csum - jmpmult[prev_char];
    }
    B[N] = '\0';
}
```

REFERENCES

1. D. Barach, D. Taenzer and R. Wells, 'The design of the PEN video editor display module', *Proceedings of the ACM Symposium on Text Manipulation, SIGPLAN Notices,* **16**, (6), 130–136 (1981).
2. J. Gosling, 'A redisplay algorithm', *Proceedings of the ACM Symposium on Text Manipulation, SIGPLAN Notices,* **16**, (6), 123–129 (1981).
3. W. Miller, *A Software Tools Sampler*, Prentice-Hall, 1987.
4. H. Thimbleby, 'The design of a terminal independent package', *Software — Practice and Experience,* **17**, (5), 351–367 (1987).
5. E. Myers, 'Incremental alignment algorithms and their applications', to appear in *SIAM J. Comput.*
6. R. Wagner and M. Fischer, 'The string-to-string correction problem', *Journal ACM,* **21**, (1), 168–173 (1974).
7. E. Myers and W. Miller, 'Row replacement algorithms for screen editors', to appear in *ACM Trans. Prog. Lang. and Systems.*
8. A. Aho, J. Hopcroft and J. Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1983.
9. W. Miller and E. Myers, 'A simple row-replacement algorithm', *Tech. Rept. TR86–37*, Computer Science Department, The Pennsylvania State University, University Park, PA 16802, 1986.
10. E. McCreight, 'A space-economical suffix tree construction algorithm', *Journal ACM,* **23**, (2), 262–272 (1976).
11. S. Johnson, 'A portable compiler: theory and practice', *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, Tucson, AZ, 97–104 (1978).
12. D. Hanson, 'Simple code optimizations', *Software — Practice and Experience,* **13**, (11), 745–763 (1983).
13. M. Waterman, 'General methods of sequence comparison', *Bull. Math. Biol.,* **46**, (4), 473–500 (1984).
14. J. Dumas and J. Ninio, 'Efficient algorithms for folding and comparing nucleic acid sequences', *Nucleic Acids Research,* **10**, (1), 197–206 (1982).
15. H. Martinez, 'An efficient method for finding repeats in molecular sequences', *Nucleic Acids Research,* **11**, (13), 4629–4634 (1983).
16. W. Wilbur and D. Lipman, 'The context dependent comparison of biological sequences', *SIAM J. Appl. Math.,* **44**, (3), 557–567 (1984).
17. D. Lipman and W. Pearson, 'Rapid and sensitive protein similarity searches', *Science,* **227**, 1435–1441 (1985).
18. C. Lawrence, D. Goldman and R. Hood, 'Optimized homology searches of the gene and protein sequence data banks', *Bull. Math. Biol.,* **48**, (5/6), 569–583 (1986).
19. L. Guibas and J. Stolfi, 'A language for bitmap manipulation', *ACM Trans. Graphics,* **1**, (3), 191–214 (1982).
20. R. Pike, 'The text editor sam', *Software — Practice and Experience,* **17**, (11), 813–845 (1987).
21. M. Denber and P. Turner, 'A differential compiler for computer animation', *Computer Graphics,* **20**, (4), (SIGGRAPH '86), 21–27 (1986).