New Applications of Failure Functions

D. S. HIRSCHBERG

University of California at Irvine, Irvine, California

AND

L. L. LARMORE

California State University, Dominguez Hills, California

Abstract. Presented are several algorithms whose operations are governed by a principle of failure functions: When searching for an extremal value within a sequence, it suffices to consider only the subsequence of items each of which is the first possible improvement of its predecessor. These algorithms are more efficient than their more traditional counterparts.

Categories and Subject Descriptors: E.1 [**Data**]: Data Structures—*arrays*; *lists*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*computations on discrete structures*

General Terms: Algorithms, Theory, Verification

Additional Key Words and Phrases: Dynamic programming, failure functions

1. *Introduction*

The notion of failure functions is often associated with the linear-time substring recognition algorithm [1, 4]. The principle of failure functions is disarmingly simple: When searching for an extremal value within a sequence, it suffices to consider only the subsequence of items, each of which is the first feasible alternative of its predecessor. The value of the failure function is a pointer to that first feasible alternative. Implementing this function in isolation will not yield any advantage since the effort required to determine the first feasible alternative is equal to the hoped for savings, which is not having to consider many losing alternatives. The preprocessing costs negate the run-time savings. (In practice, the preprocessing may be chronologically interspersed with the processing.) However, if many such searches are contemplated and they are closely related, the preprocessing costs may be spread over the multiple searches with some additional intersearch fix-up costs. The net effect may be some real savings. This was the case for the pattern-matching algorithm, and is also the case for the algorithms given in this paper.

This research was supported in part by National Science Foundation grant MCS 82-00362.

Authors' addresses: D. Hirschberg, Department of Information and Computer Science, University of California, Irvine, CA 92717; L. Larmore, Department of Computer Science, California State University, Dominguez Hills, CA 94707.

We first consider the problem of determining the optimum way to break a paragraph (scroll of words) into lines, provided the penalty function (for a line being too long or too short) is linear. Our algorithm for this problem is linear time. We then exhibit an algorithm for more general penalty functions that is linear time in the case of a piecewise quadratic function.

We also consider the problem of finding the minimum sum of key length pagination of a scroll of n items. We present a linear-time algorithm, improving on the $O(n \log n)$ result of Diehr and Faaland [2].

2. *Breaking a Paragraph into Lines*

We are given a paragraph consisting of a scroll of n words, where the ith word has length $w_i > 0$, and a nonnegative-valued function $penalty(x)$, which is defined over the closed interval $[lmin, lmax]$, where $0 < lmin < lmax$. We assume that there is an optimum line length $lopt \in [lmin, lmax]$ for which $penalty(lopt) = 0$. We define a *break sequence* of the paragraph to be a monotone increasing sequence $1 = b_1, b_2, \ldots, b_m \leq n$ of integers. The break sequence defines lines, where the kth line begins with the b_kth word, and its length is $length_k = w_{b_k} + \cdots + w_{b_{k+1}-1}$ (where b_{m+1} is taken to be $n + 1$). We say that a break sequence is *admissible* if $length_k \in [lmin, lmax]$ for all $k < m$, and $length_m \leq lmax$.

The *total penalty* of a given break sequence is defined to be the sum of the penalties of the lines, but where the last line is not penalized for being too short. The problem is to find an admissible break sequence with minimum total penalty.

In practice, text editors use penalty functions more general than the ones given in this paper. For example, there could be a special penalty for hyphenation, or the penalty for a line being the wrong length could depend on the number of words in the line. Additionally, there could be a penalty that is a function of the number of lines required. Our methods can be easily extended to cover these cases.

2.1 THE TRADITIONAL LINE-BREAKING ALGORITHM. For each $i \leq n + 1$, define $f[i]$ to be the lowest total penalty of any break sequence of the subscroll $w_i \cdots w_n$. We let $f[n + 1] = 0$ by default. The traditional algorithm (see, e.g., [3]) uses dynamic programming.

For any $1 \leq i \leq j \leq n + 1$, let $Line(i, j) = w_i + \cdots + w_{j-1}$, and let $Legal(i, j)$ be the Boolean function which is true if and if $Line(i, j) \in [lmin, lmax]$.

Algorithm 1: *Traditional Algorithm*

```
         $f[n + 1] \leftarrow 0$
Loop:    for $i$ from $n$ downto 1 do
           if $Line(i, n + 1) \leq lopt$ then
             begin
               $f[i] \leftarrow 0$
               $nextbreak[i] \leftarrow n + 1$
             end
           else if $Legal(i, j)$ for some $j$ then
             begin
Choose:        Choose $r$ such that $Legal(i, r)$ and $f[r] + penalty(Line(i, r))$ is minimized
               $f[i] \leftarrow f[r] + penalty(Line(i, r))$
               $nextbreak[i] \leftarrow r$
             end
           else
             $f[i] \leftarrow \infty$
         if $f[1] < \infty$ then Define_break_sequence
```

The subroutine Define_break_sequence recovers the breakpoint vector b from the array *nextbreak*.

```
Subroutine Define_break_sequence
$s \leftarrow 1$
$t \leftarrow 1$
while $t \le n$ do
  begin
    $b[s] \leftarrow t$
    $s \leftarrow s + 1$
    $t \leftarrow nextbreak[t]$
  end
```

The bottleneck in the Traditional Algorithm is the Choose step, since all other steps can be done in time $O(n)$. The total time for all executions of the Choose step is $O(nM)$, where M is an upper bound on the number of words that could possibly occur in a line (we could set $M = lmax/W$, where W is the minimum value of w_i). If M is considered to be bounded, then the Traditional Algorithm is linear. However, if we consider a class of problems in which M, as well as n, grows then the Traditional Algorithm is no longer linear.

2.2 Linear Penalty Function. We consider the case that *penalty* is linear and the optimum line length is either the minimum or maximum permissible. That is, $lopt$ = either $lmin$ or $lmax$, and for all $x \in [lmin, lmax]$, $penalty(x) = C(x - lopt)$, for some constant C that may be positive or negative. We define $penalty(x) = \infty$ if $x \notin [lmin, lmax]$.

We use dynamic arrays *leftlow* and *rightlow*, which have pointer (actually index) values, and dynamic arrays f and g, which have penalty values. The array f is identical to the f in the traditional algorithm, and g is a modified array that always satisfies the equation $g[k] = f[k] + C\,Line(1, k)$. At any given time, $rightlow[k]$ is the smallest $l > k$ such that $g[l] \le g[k]$, and $leftlow[k]$ is the largest $l < k$ such that $g[l] < g[k]$. *leftlow* is used as a failure function for choosing the previous breakpoint (beginning of a line) corresponding to a current end of line, and *rightlow* is used as a failure function for updating the *leftlow* values.

```
Algorithm 2: Linear Penalty Algorithm
          $g[n + 2] \leftarrow -\infty$
          $f[n + 1] \leftarrow 0$
          $g[n + 1] \leftarrow C\,Line(1, n + 1)$
          $r \leftarrow n + 1$
          $rightlow[n + 1] \leftarrow n + 2$
Loop:     for $i$ from $n$ downto 1 do
            begin
              if $Line(i, n + 1) \le lopt$ then
                begin
                  $f[i] \leftarrow 0$
                  $nextbreak[i] \leftarrow n + 1$
                end
              else
                begin
Choose1:          while $Line(i, r) > lmax$ do   $r \leftarrow r - 1$
Choose2:          while $leftflow[r]$ defined and $Legal(i, leftflow[r])$ do   $r \leftarrow leftlow[r]$
                  $f[i] \leftarrow f[r] + penalty(Line(i, r))$
                  if $f[i] < \infty$ then $nextbreak[i] \leftarrow r$
                end
              $g[i] \leftarrow f[i] + C\,Line(1, i)$
              $k \leftarrow i + 1$
```

```
Update:     while $g[k] > g[i]$ do
              begin
                $leftlow[k] \leftarrow i$
                $k \leftarrow rightlow[k]$
              end
            $rightlow[i] \leftarrow k$
          end (of Loop)
        if $f[1] < \infty$ then Define_break_sequence
```

We can prove the correctness of the Linear Penalty Algorithm by showing that it simulates the Traditional Algorithm. We need three lemmas.

LEMMA 1. *For a fixed value of i, among the set of j for which $Legal(i, j)$, that j which minimizes $g[j]$ also minimizes $f[j] + penalty(Line(i, j))$.*

PROOF. We show that the difference between $g[j]$ and $f[j] + penalty(Line(i, j))$ is constant.

$$\begin{aligned} f[j] + penalty(Line(i, j)) - g[j] &= penalty(Line(i, j)) - C\,Line(1, j) \\ &= -C(lopt + Line(1, i)), \end{aligned}$$

which is constant for fixed i. □

LEMMA 2. *The following holds after each iteration of the main loop of the Linear Penalty Algorithm, where i is the value of the loop variable.*

LB1: *For all $i \le s \le n + 1$, $rightlow[s] = t$, where $t > s$ is the smallest value such that $g[t] \le g[s]$.*

LB2: *For all $i < s \le n + 1$, $leftlow[s] = t$, where $i \le t < s$ is the largest value such that $g[t] < g[s]$, provided such a t exists. Thus, in this case, for all $leftlow[s] < j < s$, $g[j] \ge g[s]$. Otherwise, $leftlow[s]$ is undefined.*

PROOF. We prove Lemma 2 by induction on i, the loop variable of the main loop. Initially (i.e., before the main loop iterates at all), we can take $i = n + 1$. Then, LB2 holds vacuously, while LB1 holds by initial assignment.

Our inductive hypothesis is that LB1 and LB2 are true for all values of the loop variable that are greater than i. Thus, before execution of the Update loop, for all $i + 1 \le s \le n + 1$:

(a) $rightlow[s] = t$, where $t > s$ is the smallest value such that $g[t] \le g[s]$.

(b) $leftlow[s] = t$, where $i + 1 \le t < s$ is the largest value such that $g[t] < g[s]$, provided such a t exists. Otherwise, $leftlow[s]$ is undefined.

Thus, we need to prove only that, at the end of an iteration of the main loop, $rightlow[i]$ has the correct value, and that $leftlow[s] = i$ if $i < s \le n + 1$ and $g[i] < g[s]$ and $leftlow[s]$ was undefined before the Update loop.

Let $k_0 \cdots k_m$ be the sequence of values of k produced in the Update loop, that is, $k_0 = i + 1$, and $k_{l+1} = rightlow[k_l]$ for $0 \le l < m$. Note that $g[k_m] \le g[k_i]$ and $g[k_l] > g[k_i]$ for all $0 \le l < m$.

SUBLEMMA. *For all $0 \le l \le m$, and for all $i < s \le k_l$, $g[s] \ge g[k_l]$. Furthermore, if $l \ge 1$ and $s < k_l$, $g[s] \ge g[k_{l-1}]$.*

PROOF. By induction on l. For $l = 0$, the sublemma holds since we must have $s = k_0 = i + 1$. For the inductive step, assume the sublemma holds for $l - 1$. If $i < s \le k_l$, then, because $k_l = rightlow[k_{l-1}] > k_{l-1}$ by LB1, either (i) $s \le k_{l-1}$, or (ii) $k_{l-1} < s < k_l$, or (iii) $s = k_l$.

If case (i) is true, then by the inductive hypothesis, $g[s] \geq g[k_{l-1}]$.

If case (ii) is true, then $k_l = rightlow[k_{l-1}]$, by the second assignment of the Update loop. By LB1, $g[k_l] \leq g[k_{l-1}]$ and, for any s between k_{l-1} and k_l, $g[s] > g[k_{l-1}]$.

In either of these two cases, $g[s] \geq g[k_l]$, since $g[k_l] \leq g[k_{l-1}]$. Case (iii) is trivial. □

PROOF OF LEMMA 2, CONTINUED. By the sublemma, all $i < s \leq k_l$ are unsuitable values for $leftlow[k_l]$ since $g[s] \geq g[k_l]$. Therefore, for any $l < m$, $leftlow[k_l]$ should be assigned the value i if $g[i] < g[k_l]$. The first assignment of the Update loop does exactly that. We now show that $rightlow[i]$ should be assigned the value k_m. That is, we need to show that $g[s] > g[i]$ and that $g[k_m] \leq g[i]$, for $i < s < k_m$.

Since the Update loop no longer iterates when $k = k_m$, $g[k_m] \leq g[i]$. We now show that $g[s] > g[i]$ for all $i < s < k_m$. If $m = 0$, this is vacuously true since $k_0 = i + 1$. Otherwise, $g[s] \geq g[k_{m-1}]$ by the sublemma, and $g[k_{m-1}] > g[i]$ because the Update loop continues to iterate when $k = k_{m-1}$.

Thus, the Update loop makes the correct assignment to $rightlow[i]$. □

LEMMA 3. *After execution of the Choose loops, one of the following two conditions holds.*

(*I*) *There is no $j \leq n + 1$ such that $Legal(i, j)$, and r is the largest possible value of $j \leq n + 1$ such that $Line(i, j) \leq lmax$.*

(*II*) *Among all j such that $Legal(i, j)$, r is the choice of j that minimizes $g[j]$.*

PROOF. We prove Lemma 3 by induction on the loop variable i. Condition (I) holds before the first iteration of the main loop, letting $i = n + 1$. We define loop conditions, LC, and we show that they always are true.

LC1: For all $r < j \leq n + 1$ such that $Line(i, j) \leq lmax$, $g[j] \geq g[r]$.
LC2: If $r < n + 1$, $Line(i, r + 1) \geq lmin$.

We establish that LC holds initially (i.e., before the main loop, consider $i = n + 1$), and that LC is preserved by each execution of the **while** loops of the choose block, as well as when i is decremented in the main loop.

Initially, LC holds vacuously. Decrementing i cannot cause LC to fail, because $Line(i, j)$ is monotone decreasing on the first argument. An iteration of Choose1 preserves LC2 because, immediately after any such iteration, $Line(i, r + 1) > lmax \geq lmin$. Also, if Choose1 iterated one or more times, LC1 holds vacuously.

We now show that an iteration of Choose2 preserves LC. r' denotes the value of r after the iteration.

If LC holds before an iteration then, by LC1, $g[j] \geq g[r]$ for all $r < j \leq n + 1$ such that $Line(i, j) \leq lmax$. In order for Choose2 to iterate, $leftlow[r]$ is defined and $Line(i, leftlow[r]) \geq lmin$.

LC2 will hold after the iteration, since $Line(i, leftlow[r] + 1) > Line(i, leftlow[r]) \geq lmin$ and thus $Line(i, r' + 1) \geq lmin$.

To prove that LC1 is preserved, consider j such that $r' = leftlow[r] < j \leq n + 1$ and $Line(i, j) \leq lmax$. We need to show that $g[j] \geq g[leftlow[r]] = g[r']$. If $r < j \leq n + 1$, then $g[j] \geq g[r]$ by LC1 and $g[r] \geq g[r']$ by LB2. If $r' = leftlow[r] < j \leq r$, then $g[j] \geq g[r] > g[leftlow[r]]$ by LB2.

Thus, LC is loop invariant.

Consider the case when $Legal(i, r)$ after Choose1 has executed. By LC1, $g[j] \geq g[r]$ for any $j > r$ such that $Legal(i, j)$. If $j < r$ and $Legal(i, j)$, then, since Choose2 has ceased iterating, either *leftlow*[r] is undefined, or *Line*(i, *leftlow*[r]) $<$ *lmin*, which implies by monotonicity of *Line* that *leftlow*[r] $< j < r$. In either case, by LB2, $g[j] \geq g[r]$. Thus case (II) of Lemma 3 holds.

Otherwise, not *Legal*(i, r) after Choose1 has executed. In this case, *Line*(i, r) $\leq$ *lmax* since Choose1 did not iterate again, and therefore *Line*(i, r) $\leq$ *lmin* since otherwise *Legal*(i, r). Thus, Choose2 cannot iterate by the monotonicity of *Line* and the fact that if *leftlow*[r] is defined then *leftlow*[r] $< r$. Now, either Choose1 iterated once or more, or it did not iterate. If Choose1 iterated, Case (I) of Lemma 3 holds, since then *Line*(i, $r + 1$) $>$ *lmax* (and Choose2 did not iterate). If Choose1 did not iterate, r has the same value as it did after execution of Choose2 of the previous iteration of the main loop. By the inductive hypothesis, Lemma 3 held for $i + 1$. Case (II) cannot have held, because then *Line*(i, r) $>$ *Line*($i + 1$, r) $\geq$ *lmin*, which would imply that *Legal*(i, r). Thus Case (I) of Lemma 3 held for $i + 1$. Since *Line* is monotone decreasing on its first parameter, Case (I) of Lemma 3 then holds for i. □

We now prove the correctness of the Linear Penalty Algorithm, assuming that the traditional algorithm is correct. We need to show that the Choose steps of the Linear Penalty Algorithm are equivalent to the Choose step of the traditional algorithm. By Lemmas 1 and 3 (only case 3(II) applies since we are within the *Else if* clause), the value of r after execution of Choose2 is the same as the value of r chosen in the Choose step of the traditional algorithm, provided such a legal r exists. □

Time Complexity. We can use $O(n)$ preprocessing time to compute *Line*(1, i) for all i. Then *Line*(i, j) (and hence *Legal*(i, j)) can be computed in $O(1)$ time by the formula *Line*(i, j) = *Line*(1, j) − *Line*(1, i).

The Choose loops appear to iterate $O(n)$ times within each iteration of the main loop. However, r decreases with each iteration of each of those loops. Thus, the total number of such iterations cannot exceed n.

The Update loop also appears to iterate $O(n)$ times within each iteration of the main loop. Note, by LB2, the value of *leftlow*[k], once defined, is never redefined. It follows that the total number of iterations of the Update loop, over all iterations of the main loop, is at most n.

Piecewise Linear Penalty Function. The algorithm may be easily generalized to cover the case of a piecewise linear penalty function. For example, if *lmin* $<$ *lopt* $<$ *lmax*, the penalty function *penalty*(x) = $C \cdot |x -$ *lopt*$|$ is piecewise linear. The method uses independent copies of the Linear Penalty algorithm, one for each linear piece of the penalty function. These procedures are synchronized and exchange information once during each iteration of the main loop to decide on the best value for *nextbreak*[i].

2.3 General Concave Penalty Function.

We say that a function $p(x)$ is *concave* if, for any $x < y < z$ in its domain, $(z - x)p(y) \leq (y - x)p(z) + (z - y)p(x)$. For example, any quadratic function with nonnegative leading coefficient is concave.

We now consider the breaksequence problem where *penalty*(x) is nonnegative and concave for $x \in$ [*lmin*, *lmax*]. As before, *penalty*(x) = ∞ for $x \notin$ [*lmin*, *lmax*], and there is no penalty for the last line if its length does not exceed *lopt*.

The time bottleneck in the General Concave Algorithm (GCA) given below is the evaluation of the Boolean function *Bridge*. All other parts of the algorithm run in linear time, and *Bridge* needs to be evaluated $O(n)$ times. If *penalty* is quadratic, *Bridge* can be evaluated in $O(1)$ time, and hence the entire algorithm is linear. Generally, *Bridge* can be evaluated in $O(\log M)$ time by binary search, making the entire algorithm $O(n \log M)$. We leave open the possibility that a faster general algorithm exists.

Notation. For convenience, we let $F(i, j) = f[j] + penalty(Line(i, j))$, the least cost of a paragraph beginning at the ith word whose *second* line begins at the jth word.

The Boolean Function Bridge. $Bridge(j, k, l)$ is defined for $1 < j < k < l \leq n + 1$. If *true*, it means that k need not be considered as a choice for $nextbreak[i]$ for any "future" i (i.e., $i < j$), since either j or l is always (i.e., for any i) at least as good a choice as k. Formally, for the algorithm to run correctly, it suffices that *Bridge* satisfy the following two conditions:

Br1: If $1 \leq i < j < k < l \leq n + 1$ such that $Legal(i, k)$ and $Bridge(j, k, l)$, then $F(i, k) \geq \min\{F(i, j), F(i, l)\}$.
Br2: If $1 \leq i < j < k < l \leq n + 1$ such that $Legal(i, k)$ and not $Bridge(j, k, l)$, then $F(i, k) \leq \max\{F(i, j), F(i, l)\}$.

There is an allowed ambiguity in the definition of *Bridge*. Any function that satisfies Br1 and Br2 will work. For example, one possible *Bridge* function is such that it is *false* if and only if there exists some i such that $F(i, k)$ is less than $\min\{F(i, j), F(i, l)\}$. To compute this particular function, we can determine whether such an i exists by binary search since, by concavity of *penalty*, $F(i, j) \leq F(i, k)$ implies that i is too low and $F(i, l) \leq F(i, k)$ implies that i is too high. Because we can restrict our initial search domain to no more than M possible values of i, *Bridge* can be computed in $O(\log M)$ time.

Quadratic Case. Suppose that $penalty(x) = ax^2 + bx + c$ for $x \in [lmin, lmax]$, where $a \geq 0$. Then for any $j < k < l$, let $Bridge(j, k, l)$ be *true* if and only if the following two conditions hold:

Q1: $f[k] + penalty(lmax - Line(k, l)) \geq f[j] + penalty(lmax - Line(j, l))$
Q2: $Line(j, l) f[k] \geq Line(j, k) f[l] + Line(k, l) f[j] + a\, Line(j, k) Line(j, l) \cdot Line(k, l)$

Q1 and Q2 can both be computed in $O(1)$ time. Thus, we have a linear time algorithm for the case of a quadratic penalty function.

Data Structure. We make use of an input-restricted deque S of integers. Integers can be deleted from both the top and bottom ends of S, but can only be inserted to the top end. Deque S is used to choose r, similar in function to the *leftlow* pointer array in the Linear Penalty Algorithm (LPA). The chosen value of r will be at the bottom of S. Let us define time i to be the point in an algorithm when the main loop variable has value i (smaller values of i are later).

We note that the GCA is not a generalization of the LPA; the GCA constructs and traverses a failure structure in a different manner than does the LPA.

After it is completely evaluated, the *leftlow* pointer array is a failure forest that can be thought of as being rooted at 0. At any time in the LPA, the *leftlow* failure tree is only partially constructed. During each loop, the LPA progressively develops the failure tree (in Update) and eliminates from consideration some candidates by consideration of *lmax* (in Choose1) and by following a chain in the failure tree (in

Choose2). In the GCA, deque S at time i corresponds to the frontier of the developing failure tree in the LPA at the lastest time j when $Line(i, j) \geq lmin$. The following operators on S are used:

Functions:

$\mid S \mid$	= current cardinality of S
Top	= value of the top element of S
Bottom	= value of the bottom element of S
2*Top*	= value of the second from the top element of S
2*Bottom*	= value of the second from the bottom element of S

Procedures:

Pop delete the top element of S
Drop delete the bottom element of S
Push(x) insert x at the top of S

Algorithm 3: *General Concave Algorithm*

```
           $f[n+1] \leftarrow 0$
           $S \leftarrow \Lambda$ (empty list)
           $eol \leftarrow n + 1$
Loop:      for $i$ from $n$ down to 1 do
             begin
Choose1:       while $S$ nonempty and $Line(i, Bottom) > lmax$ do   Drop
Update:        while $Line(i, eol) \geq lmin$ do
                 begin
                   while $S$ nonempty and $F(i, eol) \leq F(i, Top)$ do   Pop
                   while $|S| \geq 2$ and $Bridge(eol, Top, 2Top)$ do   Pop
                   if $Line(i, eol) \leq lmax$ then $Push(eol)$
                   $eol \leftarrow eol - 1$
                 end (of Update)
Choose2:       while $|S| \geq 2$ and $F(i, 2Bottom) \leq F(i, Bottom)$ do   Drop
               if $Line(i, n+1) \leq lopt$ then
                 begin
                   $nextbreak[i] \leftarrow n + 1$
                   $f[i] \leftarrow 0$
                 end
               else if $S$ nonempty then
                 begin
                   $nextbreak[i] \leftarrow Bottom$
                   $f[i] \leftarrow F(i, Bottom)$
                 end
               else (i.e., $S = \Lambda$)
                 $f[i] \leftarrow \infty$
             end (of Loop)
           if $f[1] < \infty$ then Define_break_sequence
```

Piecewise Concave Penalty Function. If the penalty function is piecewise concave, the algorithm can be generalized, using one deque for each concave piece. The running times are simply added. If there are Γ concave pieces, the running time for the combined algorithm is $O(n\Gamma(1 + \log(M/\Gamma)))$. In the case in which the function is piecewise linear or piecewise quadratic, the running time is $O(n\Gamma)$.

The method is essentially to use independent copies of the general concave algorithm, one for each concave piece of the penalty function. These procedures are synchronized and meet once during each iteration of the main loop to exchange information and decide which one has the best value for *nextbreak*$[i]$.

3. *Pagination of Scrolls*

A *boundary sequence* for a scroll is a sequence $0 = s_0 < s_1 < \cdots < s_{v+1} = n + 1$ such that $\sum_{s_{k-1}<i<s_k} w_i \in [lmin, lmax]$ for all $1 \le k \le v + 1$, where $0 \le lmin < lmax$ are fixed. The *length* of that boundary sequence is defined to be $\sum_{1\le k\le v} w_{s_k}$. McCreight [6] asks whether we can "quickly" find a boundary sequence of minimum length.

Diehr and Faaland [2] develop an algorithm that finds the minimum length boundary sequence in $O(n \log n)$ time. We present a linear-time algorithm.

For convenience, assign any positive value, say 1, to w_{n+1} and w_0.

Define $Gap(a, b)$ as the sum of the lengths of the scroll w_i, *strictly between* the ath and the bth items. Note that $Gap(a, a + 1) = 0$. Define $Gap(a, a) = -w_a$.

Define Boolean function $Page(a, b)$ to be *true* iff $Gap(a, b) \in [lmin, lmax]$.

For any $0 \le a \le b \le n + 1$, we define an *admissible path* from a to b to be a sequence $s_0, s_1, \ldots, s_v$ such that $Page(s_{k-1}, s_k)$ for each $0 < k \le v$. The *length* of that path is $\sum_{1\le k\le v} w_{s_k}$. If there exists an admissible path from j to $n + 1$, we say that j is *accessible.*

For any $0 \le i \le n + 1$, define $f(i)$ to be the minimum length of all paths from i to $n + 1$. If i is inaccessible, let $f(i) = \infty$.

For each $0 \le i < n + 1$ such that $Page(i, k)$ for some k, define $\rho(i)$ to be the unique number that satisfies the following three conditions:

(i) $Page(i, \rho(i))$
(ii) $f(\rho(i))$ is minimized subject to (i)
(iii) $\rho(i)$ is maximized subject to (i) and (ii)

If there is no k which $Page(i, k)$ is *true*, then $\rho(i)$ is undefined. Also, $\rho(n + 1)$ is undefined.

Computation of f and ρ clearly suffices to find the minimum-length boundary sequence. A boundary sequence exists if and only if $f(0) < \infty$, and the minimum-length boundary sequence can be found by using ρ.

Algorithm 4: *Scroll Pagination*

```
           Compute $Sum[i] = \sum_{k\le i} w[k], 0 \le i \le n + 1$
           $leftlow[i] \leftarrow -1, 0 \le i \le n + 1$
           $f[n + 1] \leftarrow 0$
           $r \leftarrow n + 1$
Loop:      for $i$ from $n$ downto 0 do
             begin
Choose1:       while $Gap(i, r) > lmax$ do   $r \leftarrow r - 1$
Choose2:       while $r < leftlow[r]$ and $Gap(i, leftlow[r]) \ge lmin$ do   $r \leftarrow leftlow[r]$
               if $Page(i, r)$ then
                 begin
                   $f[i] \leftarrow f[r] + w[i]$
                   $\rho[i] \leftarrow r$
                 end
               else
                 $f[i] \leftarrow \infty$
               $k \leftarrow i + 1$
Update:        while $f[k] > f[i]$ do
                 begin
                   $leftlow[k] \leftarrow i$
                   $k \leftarrow rightlow[k]$
                 end
               $rightlow[i] \leftarrow k$
             end (of Loop)
```

It is important to distinguish between the functions $f(i)$ and $\rho(i)$ on the one hand, which are defined abstractly, and the arrays $f[i]$ and $\rho[i]$, whose values are assigned dynamically during execution of the algorithm. Also, we remind the reader that, for all $0 < i \leq n + 1$, either $f(i) = \infty$ or $f(i) = f(\rho(i)) + w_i$.

Intuitively, the algorithm works as follows. r is a running temporary $\rho(i)$, which never decreases. When r is too large because $Gap(i, r) > lmax$, r is decremented by 1 until *Gap* is small enough. We then need to decrease r, minimizing the f value, thus obtaining $\rho(i)$. In [2], a heap of possible values is maintained, and it takes $\Theta(\log n)$ time to find $\rho(i)$. In Algorithm 4, the pointer *leftlow* tells us where to look next. Even though it might take $\Theta(n)$ time to find $\rho(i)$ for a particular i, the total time for these searches over all i is still only $O(n)$, since r never increases. Thus, *leftlow* is a failure function. The pointer array *rightlow* is used for updating *leftlow*, and also for updating itself. It too is used as a failure function.

Loop Invariant. For any $0 \leq i \leq n + 1$, the following conditions hold after $n + 1 - i$ iterations of the loop of Main:

L1(i): If $\rho(i)$ is defined, $r = \rho(i)$. Otherwise, r is the largest j such that $Gap(i, j) \leq lmax$.

L2(i): For all $i \leq j \leq n + 1, f[j] = f(j)$.

L3(i): For all $i \leq j \leq n + 1$, if $\rho(j)$ is defined, $\rho[j] = \rho(j)$. Otherwise, $\rho[j]$ is undefined.

L4(i): For all $i \leq j \leq n + 1$, *leftlow*$[j]$ is the largest $i \leq k < j$ such that $f(k) < f(j)$, provided there is such a k. Otherwise, *leftlow*$[j] = -1$.

L5(i): For all $i \leq j < n + 1$, *rightlow*$[j]$ is the smallest $j < k \leq n + 1$ such that $f(k) \leq f(j)$.

The reader is referred to [5] for a detailed demonstration of the loop variants.

REFERENCES

1. Aho, A. V., Hopcroft, J. E., and Ullman, J. D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974, pp. 329–335.
2. Diehr, G., and Faaland, B. Optimal pagination of B-trees with variable-length items. *Commun. ACM 27,* 3 (Mar. 1984), 241–247.
3. Knuth, D. E., and Plass, M. F. Breaking paragraphs into lines. *Softw. Pract. Exper. 11,* 12 (1981), 1119–1184.
4. Knuth, D. E., Morris, J. H., and Pratt, V. R. Fast pattern matching in strings. *SIAM J. Comput. 6,* 2 (June 1977), 323–350.
5. Larmore, L. L., and Hirschberg, D. S. Efficient optimal pagination of scrolls. *Commun. ACM 28,* 8 (Aug. 1985), 854–856.
6. McCreight, E. M. Pagination of B*-trees with variable-length records. *Commun. ACM 20,* 9 (Sept. 1977), 670–674.

RECEIVED NOVEMBER 1984; REVISED MAY 1985; ACCEPTED JUNE 1986