SIAM J. COMPUT.
Vol. 16, No. 4, August 1987

THE LEAST WEIGHT SUBSEQUENCE PROBLEM*

D. S. HIRSCHBERG† AND L. L. LARMORE†

Abstract. The least weight subsequence (LWS) problem is introduced, and is shown to be equivalent to the classic minimum path problem for directed graphs. A special case of the LWS problem is shown to be solvable in $O(n \log n)$ time generally and, for certain weight functions, in linear time. A number of applications are given, including an optimum paragraph formation problem and the problem of finding a minimum height B-tree, whose solutions realize improvement in asymptotic time complexity.

Key words. algorithms, dynamic programming

AMS(MOS) subject classifications. 68P05, 68Q25

1. Introduction. We define an instance of the *Least Weight Subsequence* problem (hereinafter referred to as the LWS problem) on $[a, b]$ as follows:

We are given a real-valued function *weight* (i, j), defined for all $a \leqq i < j \leqq b$. If $i_0, i_1, \cdots, i_t$ is a strictly monotone increasing sequence of integers in the range $[a, b]$, we define the *weight* of the sequence to be $\sum_{1 \leqq s \leqq t}$ *weight* (i_{s-1}, i_s).

The *single pair* LWS problem (or simply, LWS problem) is to find that monotone sequence of integers starting with a and ending with b which has minimum weight. The *all pairs* LWS problem is to solve the LWS problem for all pairs (a_0, b_0) such that $a \leqq a_0 \leqq b_0 \leqq b$.

We say that the weight function *weight* is *concave* if, for all $i_0 \leqq i_1 < j_0 \leqq j_1$,

$$\textit{weight}\ (i_0, j_0) + \textit{weight}\ (i_1, j_1) \leqq \textit{weight}\ (i_0, j_1) + \textit{weight}\ (i_1, j_0).$$

This concavity condition is exactly the *quadrangle inequality* introduced by Yao [6]. If *weight* is concave, we say it defines an instance of the *concave* LWS problem.

The LWS problem is equivalent to a minimum path problem for a weighted directed graph. Given an instance of the LWS problem, let $G = (V, E)$ be the directed graph with $V = \{0 \cdots n\}$ and $E = \{(i, j) \mid i < j\}$, and let *weight* be a weight function on the edges of G. Then the minimum path problem is to find a minimum weight path from 0 to n.

Conversely, given any instance P of the min path problem on directed acyclic graph $G = (V, E)$ with a weight function *weight*, choose a topological order on V. A solution to the LWS problem, where V has been numbered 0 to n in accord with the topological ordering, yields a solution to the minimum weight path problem on G.

Aside from theoretical interest, the LWS problem (and its solution) are useful in bringing to light improved solutions to a number of practical problems. In the ensuing sections, we discuss three algorithms for the LWS problem and their applications.

In § 2, we discuss the Traditional Algorithm, which solves the general LWS problem in quadratic time. In § 3, we introduce our Basic Algorithm, which solves the general concave LWS problem in $O(n \log n)$ time. In § 4, we describe an algorithm which we call the Scale-Assisted Algorithm, which solves the concave LWS problem in linear time, provided that the weight function is supported by a suitable scale function, and

* Received by the editors March 25, 1985; accepted for publication (in revised form) October 7, 1986. A preliminary version of this paper was presented at the 26th Annual Symposium on Foundations of Computer Science held in Portland, Oregon, October 21-23, 1985. An extended abstract of the paper appears in the Proceedings, pp. 137-143.

† Department of Information and Computer Science, University of California, Irvine, California 92717.

that a certain inequality can be solved in constant time. In § 5, we give some applications, including the paragraph breaking problem with quadratic penalties.

2. The Traditional Algorithm. In this section we describe a straightforward algorithm for solving the LWS problem, using dynamic programming. A variation of this Traditional Algorithm is used in TEX to solve the paragraph breaking problem [3].

Suppose we are given an instance of the LWS problem on $[0, n]$, defined by a weight function *weight*. Let $f(i)$ be the least weight of any subsequence which begins with 0 and ends with i. (Thus $f(0)=0$, and $f(n)$ is the weight of the solution to the LWS problem.) The function f exists abstractly. The Traditional Algorithm uses dynamic programming to construct an array $f[*]$ which contains the values of this abstract function.

We define an abstract function $g(i,j)=f(i)+weight\ (i,j)$, for $i<j$ integer in $[0, n]$. We may consider $g(i,j)$ to be a candidate value for $f(j)$, i.e., $f(j)$ is the minimum, over all $i<j$, of $g(i,j)$. During execution of the Traditional Algorithm, the value of $g(i,j)$ can be computed in constant time, provided $f[i]$ has already been evaluated.

Define *bestleft* (j) to be the second from the last element in a solution to the LWS problem restricted to $[0,j]$, i.e., *bestleft* $(j)=i$, where $f(j)=f(i)+weight\ (i,j)$. Since there need not be a unique least weight subsequence, there may be more than one equally good choice of *bestleft*, in which case, we allow one of them to be chosen arbitrarily. During execution of the algorithm, an array *bestleft* is defined which contains the values of the abstract function *bestleft*.

THE TRADITIONAL ALGORITHM

```
f[0] ← 0
for m from 1 to n do
  begin
    f[m] ← g(0, m)
    bestleft [m] ← 0
    for i from 1 to m − 1 do
      if g(i, m) < f[m] then
        begin
          f[m] ← g(i, m)
          bestleft [m] ← i
        end
  end
//  Comment: the array f has now been defined.
//  The remainder of the algorithm recovers the LWS using bestleft.
L ← (n)
m ← n
while m > 0 do
  begin
    m ← bestleft [m]
    prepend m to L
  end
return (f[n], L)
```

Analysis of the Traditional Algorithm. During each iteration of the main loop of the Traditional Algorithm, the value of $f[m]$ is computed, under the assumption that

the values of $f[i]$, for all $i < m$, have already been evaluated. The value of *bestleft* $[m]$ is simultaneously computed.

The computation of each $f[m]$ requires examining $f[i]$ for all $i < m$. Thus the Traditional Algorithm requires quadratic time.

Application of the Traditional Algorithm to the All Pairs LWS Problem. Only a slight modification is necessary to solve the all pairs problem. If the entire solution were given in output, it would consist of the approximately $n^2/2$ separate subsequences. Instead, the solution to the LWS problem for a pair (a, b) such that $0 \leqq a \leqq b \leqq n$ is implicitly stored in the values of $bestleft_a$. For any fixed a and b, the solution to the LWS problem for the pair (a, b) can be recovered in linear time using the pointer array $bestleft_a$. Thus, $\Theta(n^3)$ time and only $\Theta(n^2)$ space is required.

3. The Basic Algorithm. In this section, we introduce a new algorithm, which we call the *Basic* Algorithm since it forms the basis for the more complicated algorithms which are developed later.

We shall assume that we are given an instance of the concave LWS problem on $[0, n]$, defined by a weight function *weight*.

The Basic Algorithm makes use of an input-restricted deque D. At all times, D will be a subsequence of the integers $0, \cdots, n$. We refer to the least and greatest elements of the subsequence D as the *front* and *rear*, respectively. Three updating procedures are permitted on D: *Hire* (m), which appends a new element m to the rear of D; *Fire*, which removes the rear element of D; and *Retire*, which removes the front element of D.

In addition, we will need to access the second from the front element of D, which we call *front*2, and the second from the rear element of D, which we call *rear*2. These will, of course, be defined only when $|D| > 1$.

The deque D contains all current candidates for *bestleft* $[m]$, for all m which are yet to be considered. A new element joins D when it might be the value of *bestleft* in the future. The front element is removed (i.e., *Retire* is executed) when it is no longer possible for it to be *bestleft* for any future m, and an element is removed from the rear (i.e., *Fire* is executed) when it can be determined that it will *never* be the correct value of *bestleft* for any m. It is the determination of whether *Fire* should be executed that takes most of the time of the Basic Algorithm.

THE BASIC ALGORITHM

```
f[0] ← 0
D ← (0)
for m from 1 to n − 1 do
  begin
    f[m] ← g(front, m)
    bestleft [m] ← front
    while |D| > 1 and g(front2, m + 1) ≦ g(front, m + 1) do
      Retire
    while |D| > 1 and Bridge (rear2, rear, m) do
      Fire
    if g(m, n) < g(rear, n) then
      Hire (m)
  end
f[n] ← g(front, n)
bestleft [n] ← front
```

```
// Comment: the array f has now been defined.
// The remainder of the algorithm recovers the LWS using bestleft.
L ← (n)
m ← n
while m > 0 do
  begin
    m ← bestleft [m]
    prepend m to L
  end
return (f[n], L)
```

The key to the Basic Algorithm is the Boolean function *Bridge*. Intuitively, *Bridge* (i_0, i_1, i_2) if and only if i_1 can be ignored in the future because either i_0 or i_2 is always at least as good a choice for *bestleft*. Formally, for any $i_0 < i_1 < i_2$, *Bridge* (i_0, i_1, i_2) is false if and only if there exists some $k > i_2$ such that $g(i_1, k) < g(i_0, k)$ and $g(i_1, k) < g(i_2, k)$. Since *weight* is concave, *Bridge* can be evaluated by a binary search algorithm. We give an example of one such algorithm below.

```
BRIDGE (a, b, c)

if c = n then
    return (true)
else if g(a, n) ≦ g(b, n) then
    return (true)
else
  begin
    low ← c
    high ← n
    while high − low ≧ 2 do
      begin
        mid ← ⌊(low + high)/2⌋
        if g(a, mid) ≦ g(b, mid) then
            low ← mid
        else
            high ← mid
      end
    if g(c, high) ≦ g(b, high) then
      return (true)
    else
      return (false)
  end
```

The above *Bridge* Algorithm works as follows. Let *Right* be the set of k in $[c+1, n]$ for which $g(b, k) < g(a, k)$ and, similarly, let *Left* be the set of k in $[c+1, n]$ for which $g(b, k) < g(c, k)$. By definition, *Bridge* (a, b, c) is true iff *Right* and *Left* are disjoint. By the concavity condition, *Left* is a left subinterval of $[c+1, n]$ and *Right* is a right subinterval. Thus, their intersection is nonempty iff *Right* is nonempty and its minimum element is also in *Left*.

The first two steps of *Bridge* determine whether *Right* is empty. If so, *true* is returned. The next step uses binary search to find the minimum value of *Right* (stored in *high*), as it is known that this is a right subinterval. Finally, the correct value of the function is determined by checking whether *high* is in *Left*.

Analysis of the Basic Algorithm.

The Boolean Function Bridge. The value of *Bridge* (a, b, c) is false if and only if there exists $c < k \leqq n$ such that $g(b, k)$ is smaller than is either $g(a, k)$ or $g(c, k)$, i.e., if there exists some k where b is a strictly better candidate for *bestleft* $[k]$ than either a or c. We are interested in evaluating *Bridge* because if *Bridge* (*rear*2, *rear*, m) then *rear* must be removed from D (i.e., *Fire* must be executed) since it will not be the correct choice of *bestleft* $[k]$ for any future k.

Time Analysis of the Basic Algorithm. Each integer from 0 to n is entered onto the deque D exactly once. Since each execution of *Retire* or *Fire* decreases $|D|$ by one, the total number of executions of these two procedures cannot exceed $n+1$. The function *Bridge* must be evaluated once during each iteration of the main loop of the algorithm, plus one additional time after each execution of *Fire*, a total of not more than $2n$ times altogether. Each execution of *Bridge* requires $O(\log n)$ time. (We assume the reader is familiar with the technique of binary search.) All other parts of the basic algorithm require at most linear time, so the time requirement of the Basic Algorithm is $O(n \log n)$.

All Pairs Version. The all pairs LWS problem can be solved by iterating the basic algorithm n times.

Correctness of the Basic Algorithm. The proof of correctness is given in the Appendix.

4. The Scale-Assisted Algorithm. We now consider a method for speeding up the basic algorithm, that makes use of a *scale function* and a *difference-of-weight* function. We give an algorithm, called the Scale-Assisted Algorithm, a modification of the Basic Algorithm, which runs in linear time, provided a zero of the difference-of-weight function can be found in constant time.

More generally, the Scale-Assisted Algorithm runs in time $O(n \cdot \mathit{difftime})$, where *difftime* is the time required for finding a zero of the difference-of-weight function. Thus, the scale-assisted algorithm is asymptotically faster than the basic algorithm, if $\mathit{difftime} = o(\log n)$.

Let us consider an instance of the concave LWS problem on the sequence $0, \cdots, n$, where *weight* is the weight function. Let us suppose that we are given two real-valued functions

Scale (i)	strictly monotone increasing, defined for all integers $0 \leqq i \leqq n$,
diff (i, j, x)	defined for integers $0 \leqq i < j < n$ and real $x >$ *Scale* (j), monotone increasing in x for fixed i and j,

such that the following condition holds:

$$\mathit{diff}\,(i, j, \mathit{Scale}\,(k)) + \mathit{weight}\,(j, k) = \mathit{weight}\,(i, k) \quad \text{for all integers } 0 \leqq i < j < k \leqq n.$$

We then say that the weight function *weight* is *supported* by the scale function *Scale* and the difference-of-weight function *diff*.

The reader may easily verify that, in any instance of the concave LWS problem, the weight function is supported by some choice of scale and difference-of-weight functions. However, we do not start with a weight function and then find choices of *Scale* and *diff* which support it. Rather, we only apply the Scale-Assisted Algorithm in problems where *Scale* is given *prior* to *weight*, and *weight* depends on *Scale* in some simple way.

One such problem is the paragraph breaking problem, with quadratic penalty function, where hyphenation is allowed. We outline this application in § 5.

The only difference between the Basic Algorithm and the Scale-Assisted Algorithm is the meaning (and hence implementation) of the Boolean function *Bridge*. If $a < b < c$ are integers in $[0, n]$, *Bridge* (a, b, c) will be defined to be *true* if and only if, for all $x >$ *Scale* (c), at least one of the following two conditions holds:

(1) $f(a) - f(b) + diff(a, b, x) \leqq 0$,

(2) $f(b) - f(c) + diff(b, c, x) \geqq 0$.

In the Basic Algorithm, *Bridge* (a, b, c) is said to be true if there is no integer $d > c$ such that b is a strictly better candidate for *bestleft* $[d]$ than is either a or c. In the Scale-Assisted Algorithm, *Bridge* (a, b, c) is said to be true if there is no real number x which could be the value of *Scale* (d) for any possible $d > c$ for which b could be a better choice of *bestleft* than could either a or c.

Thus, it is not necessary to look at the various d, which is what the binary search implementation of *Bridge* does. Rather, it is only necessary to look at various choices of real x. In principal, there need be no time saving. But in practice (such as the paragraph breaking example in § 5), the function *diff* is frequently simple enough to allow evaluation of *Bridge* in constant time.

The Scale-Assisted Algorithm is identical to the basic algorithm. Only the *Bridge* function is implemented differently.

FUNCTION BRIDGE (a, b, c)—SCALE-ASSISTED ALGORITHM

```
if f(a) − f(b) + diff (a, b, Scale (n)) ≦ 0 then
    return (true)
else if f(b) − f(c) + diff (b, c, Scale (n)) < 0 then
    return (false)
else
  begin
    x0 ← lub {x | diff (a, b, x) ≦ 0}
    x1 ← glb {x | diff (b, c, x) ≧ 0}
    if x0 < x1 then
        return (false)
    else if x0 > x1 then
        return (true)
    else if diff (a, b, x0) > 0 and diff (b, c, x0) < 0 then
        return (true)
    else
        return (false)
  end
```

5. Some applications.

Example 1. *Airplane refueling.* Suppose that an airplane needs to fly between two given airports, which are distance R apart. Suppose there are $n-1$ optional refueling stops, at distances $x_1, \cdots, x_{n-1}$ from the departure point. For simplicity, assume that all these stops lie on the interval connecting the departure point and the destination. We can let $0 = x_0 < x_1 < \cdots < x_{n-1} < x_n = R$.

We suppose that the fuel consumption of the airplane is proportional to the weight of the plane, including unused fuel, and also that there is a fixed fuel cost of landing, a cost of taking off, and a landing fee which varies at the different stops. The problem of minimizing cost of the flight is then an instance of the concave LWS problem, where $weight(i, j) = e^{\alpha + \beta(x_j - x_i)} + L_j$, where L_j is the landing fee at the jth stop, and α and β are constants which absorb the other parameters of the problem.

The airplane refueling problem can be solved in linear time, using the Scale-Assisted Algorithm, by choosing the scale function to be simply the distance from the departure point, i.e., $Scale(i) = x_i$. For any $i < j$, we then have $diff(i, j, x) = C_{i,j} e^{\beta x} + L_i - L_j$, where $C_{i,j}$ does not depend on x. For fixed i and j, the solution to the equation $diff(i, j, x) = 0$ can thus be found in the amount of time it takes to compute a logarithm (which we take to be constant). Thus *Bridge* takes constant time, and the problem can be solved in linear time.

Example 2. *Optimum paragraph formation.* We are given a scroll of words, and each word consists of one or more syllables. Let n be the number of syllables, and let w_i be the length of the ith syllable. The space between two words will be considered to be attached to the last syllable in the earlier word. We wish to form this scroll into a paragraph, minimizing the total penalty. A paragraph may be defined by a break-sequence, consisting of a subsequence of $0, \cdots, n$, where each element of the break-sequence is the index of the last syllable on a line. (0 shall always be the first element of the breaksequence, being the last syllable on the fictitious 0th line.)

We assume that penalties are assigned as follows. There is an optimum length for a line, namely *lineopt*. There are also minimum and maximum lengths for a line, i.e., *linemin* and *linemax*. We assume that the penalty for a line being too short or too long is proportional to the square of the difference between the length of that line and the optimal length, except that the last line cannot be penalized for being too short. We also assume that there is fixed penalty for breaking any word.

A weight function *weight*, along with supporting scale function *Scale* and difference-of-weight function *diff*, can be assigned as follows:

Let w_i be the length of the ith syllable.

Let *spacelen* be the length of a space (between words).

Let *hyphenlen* be the length of a hyphen, assumed never to exceed the length of the rest of the word.

Let $Scale(i) = w_1 + \cdots + w_i - spacelen$, if the ith syllable is the last syllable of its word. Remember that w_i includes the length of a space.

Let $Scale(i) = w_1 + \cdots + w_i + hyphenlen$, if the ith syllable is not the last syllable of its word.

For real x, let $penalty(x) = A \cdot (x - lineopt)^2$ if $linemin \leqq x \leqq linemax$, where A is some positive constant; otherwise, let $penalty(x) = \infty$.

For $0 \leqq i < j \leqq n$, define $weight(i, j) = 0$, if $j = n$ and $Scale(n) - Scale(i) \leqq lineopt$, otherwise let $weight(i, j) = penalty(Scale(j) - Scale(i)) + \delta \cdot B$, where $\delta = 0$ if the jth syllable is at the end of its word, and $\delta = 1$ if the jth syllable is not at the end of its word. B is the hyphen penalty, a positive constant.

There is a problem in defining the function $diff(i, j, x)$ in some cases, since $\infty - \infty$ is an indeterminate form. There are two ways to resolve this problem. One way is to force the definition of *diff* in those cases in the unique way which causes the algorithm to work. Another, more elegant way, is to eliminate infinite penalties entirely, replacing them with very large penalties which maintain the concavity condition on *weight*. Accordingly, we define, in the following, a weight function which agrees with the previous definition in all finite cases, which is very large in cases where the previous definition requires ∞, and which is concave.

First, a value $\mathbf{M}$ is calculated which is larger than the total penalty of any actual paragraph. For example, we could let $\mathbf{M} = B \cdot n + A \cdot n\,(linemax - linemin)^2$. Let $\varepsilon > 0$ be less than the difference of the scale values of any two syllables, i.e.,

$\varepsilon < Scale\,(i) - Scale\,(i-1)$ for all i. Now, redefine

$$penalty\,(x) = A \cdot (x - lineopt)^2 \quad \text{if } linemin \leqq x \leqq linemax,$$
$$\mathbf{M} \cdot 2^{|x - lineopt|/\varepsilon} \quad \text{otherwise.}$$

Finally, define $weight\,(i, j)$ as before, but with the new formula for $penalty\,(Scale\,(j) - Scale\,(i))$. Then, for any $0 \leqq i < j < n$, and any real $Scale\,(j) < x \leqq Scale\,(n)$, $diff\,(i, j, x) = weight\,(Scale\,(i), x) - weight\,(j, x)$, and we observe that

$$\begin{aligned}
diff\,(i, j, x) &> 0 \quad \text{if } x - Scale\,(i) > linemax,\\
&< 0 \quad \text{if } x - Scale\,(j) < linemin \quad \text{and} \quad x < Scale\,(n),\\
&> 0 \quad \text{if } x = Scale\,(n) \text{ and } x - Scale\,(i) > lineopt,\\
&= 0 \quad \text{if } x = Scale\,(n) \text{ and } x - Scale\,(i) \leqq lineopt,\\
&= A \cdot [Scale\,(i)^2 - Scale\,(j)^2 - (Scale\,(i) - Scale\,(j)) \cdot (lineopt + 2x)]\\
&\qquad \text{otherwise.}
\end{aligned}$$

For fixed i and j, $diff\,(i, j, x)$ is a monotone increasing function of x, and is linear in the interval where it may be zero. Thus, it takes only constant time to compute the values of the least upper bound and greatest lower bound needed in the evaluation of *Bridge*. It follows that the optimum paragraph may be found in linear time.

There are additional recent discussions of the paragraph problem [2], [5].

Example 3. *Minimal Height* B-*Trees.* Diehr and Faaland [1] give an algorithm for finding a minimum height B-tree structure on a scroll (i.e., list) of words, in $O(n^3 \log n)$ time. We show how to use the all pairs version of the Basic Algorithm to solve the problem in $O(n^2 \log^2 n)$ time, and also how to use the all pairs version of the Scale-Assisted Algorithm to solve the problem in $O(n^2 \log n)$ time.

Given a scroll of n words, where the ith word has length w_i, we give a recursive definition of a B-tree structure of height h on the scroll.

A B-tree structure of height 0 is defined to be the empty subsequence of the scroll. A B-tree structure of height $h > 0$ consists of:

(1) A subsequence of the scroll, called the *boundary sequence.*

(2) A B-tree structure of height $h-1$ on the *boundary sequence* (itself a scroll).

We define a *page* of the B-tree structure to be either the substring consisting of all words between consecutive elements of the breaksequence (such a page is called a *leaf*), or a page of the B-tree structure on the *boundary sequence.* The *length* of a page is defined to be the sum of the lengths of all the words which constitute that page.

The *height* of each page can be recursively defined. A leaf has height 0. A page of height h is a page of height $h-1$ of the B-tree structure on the *boundary sequence.* Each word of the scroll lies in just one page; we define the *height* of a word to be the height of the page which contains that word.

The pages of a B-tree of height structure form a tree. A leaf consists of a substring of the scroll. If the leaf is the entire scroll, that leaf is the root, and is the unique page. Otherwise, the leaf is bounded on at least one end by a word of height 1. We then define the parent of the leaf to be the page of height 1 which contains that word. (Note that if both ends are bounded by words of height 1, they belong to the same page.) Parents of pages of height greater than 0 are defined recursively.

It is important to note that a page of a B-tree can be empty, and that an empty page can have either zero or one child.

In applications, B-trees normally have a restriction on the total length of any page.

Suppose that no page can have length exceeding some positive constant *pagemax*. We can then ask two questions about a given scroll consisting of n words:

(1) Does there exist a B-tree structure on that scroll?

(2) What is the minimum height B-tree structure on the scroll?

The answer to the first question is yes, if and only if $w_i \leqq pagemax$ for all i. If so, there certainly exists a B-tree structure of height $O(\log n)$. The Basic Algorithm can be used to solve an instance of the all pairs LWS problem for each h from 0 to the actual minimum height, constructing the minimum height B-tree structure in time $O(n^2 \log^2 n)$. We outline the method below.

The Diehr-Faaland Graph. It is convenient to enlarge the scroll by introducing fictitious words in positions 0 and $n+1$, where $w_0 = w_{n+1} = 0$. For any integer $h \geqq 0$ we define the hth *Diehr-Faaland graph* G_h of the scroll to be an acyclic directed graph where

(1) the vertices of G_h are the integers $[0, n+2]$, and

(2) the edges of G_h consist of all ordered pairs (i, j) such that $i < j$ and there exists a B-tree structure of height h on the scroll consisting of all words strictly between the ith and the jth words.

We note:

(1) $(i, i+1)$ is an edge of G_h.

(2) If (i, k) is an edge of G_h, then (i, j) is also an edge of G_h for all $i < j < k$.

(3) If (i, j) is an edge of G_h, then (i, j) is also an edge of G_{h+1}.

(4) The smallest h such that $(0, n+1)$ is an edge of G_h is the height of the minimum height B-tree structure on the scroll.

In passing, we note that condition 2 allows the directed graph G_h to be represented using only linear storage. For each $0 \leqq i < n+1$, let $farthest\_j_h[i]$ be the largest j such that (i, j) is an edge of G_h. Any pair (i, j) is an edge of G_h if and only if $j \leqq farthest\_j_h[i]$, thus only the array $farthest\_j_h$ need be stored to represent G_h.

Our method is to first construct the graph G_0, then use the all pairs version of the Scale-Assisted Algorithm to construct G_{h+1} from G_h for all h. We stop when $(0, n+1)$ is an edge.

It is convenient to refer to $Sum(i) = w_1 + \cdots + w_i$. All values of *Sum* can be precomputed in linear time.

Construction of G_0. For any i, $farthest\_j_0[i]$ is the largest j such that $Sum(j-1) - Sum(i) \leqq pagemax$. It takes linear time to evaluate $farthest\_j_0$.

We now show how to construct G_{h+1} from G_h. Pick $\varepsilon > 0$, smaller than the smallest w_i. Let $\mathbf{M}$ be a sufficiently large number. For $0 \leqq i < j \leqq n+1$, let

$$weight(i, j) = \begin{cases} w_j & \text{if } (i, j) \text{ is an edge in } G_h, \\ \mathbf{M} \cdot 2^{Sum(j) - Sum(i)} & \text{otherwise.} \end{cases}$$

The value of $weight(i, j)$ when (i, j) is not an edge of G_h is a substitute for ∞. $\mathbf{M}$ must be chosen large enough that the edge (i, j) is never used in the LWS.

We now use the Basic Algorithm to compute the weight of the LWS between all pairs (i, j). Since the Basic Algorithm requires $O(n \log n)$ time, the all pairs version requires $O(n^2 \log n)$ time. Let Let $L_h(i, j)$ be the weight of this LWS. Finally, G_{h+1} may be defined as follows: (i, j) is an edge of G_{h+1} if and only if $L_h(i, j) \leqq pagemax + w_j$.

The method outlined above requires $O(n^2 \log^2 n)$ time to find the minimum height B-tree structure, and requires $O(n^2 \log n)$ storage. In fact, however, because of the special nature of the B-tree problem, the all pairs version of the Scale-Assisted Algorithm

may be used instead. Thus, the algorithm may always be sped up to require only $O(n^2 \log n)$ time. The reason is that it is unnecessary to use binary search to evaluate *Bridge*. In fact, because of the simple formula for *weight* (i, j), *Bridge* (a, b, c) is true if and only $f[b] \geqq f[c]$, and thus takes constant time to evaluate.

We note that Diehr and Faaland [1] also give a heuristic for finding a minimum height B-tree structure by optimizing the pagination of a scroll for each level of the tree. Their algorithm uses time $O(n \log n)$ per level. This has been improved to $O(n)$ per level [2], [4].

Appendix: Correctness of the Basic Algorithm. We assume the correctness of the Traditional Algorithm. The Basic Algorithm differs from the Traditional Algorithm only in the contents, but not the purpose, of the main loop. The purpose of the mth iteration of the main loop (for both algorithms) is to assign the correct values of $f[m]$ and *bestleft* $[m]$.

The Forward Property. Concavity of the weight function *weight* guarantees that, for $0 \leqq a < b < c < d \leqq n$, $g(b, c) < g(a, c) \Rightarrow g(b, d) < g(a, d)$. We call this the *forward property* of g. Intuitively, the forward property is that if b is superior to a as a candidate for *bestleft* $[c]$, it is also a superior candidate for *bestleft* $[d]$ for all $d > c$.

The Best-Candidates Condition. We say that the deque D satisfies the *Best-Candidates condition* up to m, which we denote by BC (m), if:

BC1 (m). For any $j > m$, there exists some $i \in D$ such that $g(i, j) \leqq g(k, j)$ for all $0 \leqq k \leqq m$. That is, there is some element of D which is at least as good a choice for *bestleft* as any other element in the range $[0, m]$.

BC2 (m). For any $i \in D$, there exists some $j > m$ such that $g(i, j) < g(k, j)$ for all $k \in D$, $k \neq i$. That is, there exists some element beyond m for which i is not only the best choice of *bestleft* $[j]$, but also the *strictly* best choice among the elements of D.

We show that the Best-Candidates Condition is a loop invariant, i.e., D satisfies BC (m) after m iterations of the main loop of the Basic Algorithm, for any $m < n$.

After 0 iterations of the main loop, $D = (0)$, and thus D satisfies BC (0).

Suppose that D satisfies BC $(m-1)$ before the mth iteration of the main loop. We need to show that the updating of D within that iteration causes D to satisfy BC (m). (We only consider $m < n$, since if $m = n$, no updating of D is done because it is no longer needed.)

It is helpful to realize that BC1 (m) is a positive condition, i.e., is fulfilled if D has "enough" elements, while BC2 (m) is a negative condition, i.e., is fulfilled if D has "not too many" elements. Precisely stated, BC1 (m) cannot be made false by adding elements to D, while BC2 (m) cannot be made false by removing elements from D.

When m is appended to D, D will satisfy BC1 (m), provided nothing is removed. We need to show that those items removed do not contribute to BC1 (m). When *Retire* is executed, *front* is removed from D. But this only occurs if *front*2 is at least as good a choice for *bestleft* $[m+1]$ (and hence, by the forward property of g, for all $k > m$) as *front*. Thus execution of *Retire* does not harm BC1 (m). When *Fire* is executed, *rear* is removed from D. But this only occurs if either *rear*2 or m (both of which will still be on D) is at least as good a choice for *bestleft* $[k]$ as *rear* for all $k > m$. Thus, execution of *Fire* does not harm BC1 (m).

In order to show that after the mth iteration of the main loop BC2 (m) is satisfied, we need to show that iteration removes every item which would cause BC2 (m) to fail.

For each $i \in D$ before the iteration, let $j_i > m-1$ be the element required by the condition BC2 $(m-1)$. Without loss of generality, j_i is the smallest such element. By

the forward condition on g, $i_0 < i_1 \Rightarrow j_{i_0} < j_{i_1}$. Suppose $i \in D$ before the iteration, but BC2 $(m) \Rightarrow i \notin D$. There are two possible reasons for this. Either $j_{front2} = m+1$, in which case $i = front$ and is deleted when *Retire* is executed, or m is as least as good a choice for *bestleft* $[j_i]$ as i itself. In the latter situation, by the forward condition, and by the condition BC2 $(m-1)$, m will be the best choice for *bestleft* $[j]$ for all $j_i \leqq j \leqq n$. Thus, *Fire* will be iterated, removing all elements of D from *rear* down to and including i. By the forward condition, when *Bridge* (*rear*2, *rear*, m) is false, BC2 (m) is satisfied.

Finally, we show that, since D satisfies BC $(m-1)$ before the mth iteration, $front = bestleft\ (m)$. Suppose not. Then $bestleft\ (m) = i < m$, for some $i \in D$, by BC1 $(m-1)$. By BC2 $(m-1)$, there exists some $j \geqq m$ such that $g(front, j) \leqq g(i, j)$. By the concavity condition on *weight*, $g(front, m) \leqq g(i, m)$, since $m \leqq j$, a contradiction.

Acknowledgments. The authors wish to thank Alan Friese and an anonymous referee for pointing out errors in an early draft of this paper.

REFERENCES

[1] G. DIEHR AND B. FAALAND, *Optimal pagination of B-trees with variable-length items*, Comm. ACM, 27 (1984), pp. 241-247.

[2] D. S. HIRSCHBERG AND L. L. LARMORE, *New applications of failure functions*, J. Assoc. Comput. Mach., to appear.

[3] D. E. KNUTH AND M. F. PLASS, *Breaking paragraphs into lines*, Software—Practice & Experience (1981), pp. 1119-1184.

[4] L. L. LARMORE AND D. S. HIRSCHBERG, *Efficient optimal pagination of scrolls*, Comm. ACM, 28 (1985), pp. 854-856.

[5] ———, *Breaking a paragraph into lines in linear time*, Proc. 22nd Annual Allerton Conference on Comm., Control, and Computing, Monticello, IL, October 1984.

[6] F. FRANCES YAO, *Efficient dynamic programming using quadrangle inequalities*, Proc. 12th Annual ACM Symposium on the Theory of Computing, April 1980, pp. 429-435.