JOURNAL OF ALGORITHMS 9, 418–425 (1988)

The Concave Least-Weight Subsequence Problem Revisited

ROBERT WILBER

AT&T Bell Laboratories, Murrary Hill, New Jersey 07974

Received August 27, 1987; accepted November 3, 1987

We are given an integer n and a real-valued function $w(i, j)$ defined for integers $0 \le i < j \le n$ and with the property that $w(i_0, j_0) + w(i_1, j_1) \le w(i_0, j_1) + w(i_1, j_0)$ for $0 \le i_0 < i_1 < j_0 < j_1 \le n$. The *concave least-weight subsequence problem* is to find an integer $k \ge 1$ and a sequence of integers $0 = l_0 < l_1 < \cdots < l_{k-1} < l_k = n$ such that $\Sigma_{i=0}^{k-1} w(l_i, l_{i+1})$ is minimized. One application of this problem is determining optimal line breaks in a text formatting system. D. S. Hirschberg and L. L. Larmore (*SIAM J. Comput.* **16** (1987), 628–638) showed that the concave least-weight subsequence problem can be solved in $O(n \log n)$ time and that if a certain extra condition is imposed it can be solved in $O(n)$ time. Here we show that the concave least weight subsequence problem can always be solved in $O(n)$ time, without any extra conditions. © 1988 Academic Press, Inc.

1. INTRODUCTION

Hirschberg and Larmore [2] define the *least-weight subsequence* problem (henceforth called the LWS problem) as follows. Given an integer n, and a real-valued weight function $w(i, j)$ defined for integers $0 \le i < j \le n$, find an integer $k \ge 1$ and a sequence of integers $0 = l_0 < l_1 < \cdots < l_{k-1} < l_k = n$ such that $\Sigma_{i=0}^{k-1} w(l_i, l_{i+1})$ is minimized. The weight function is *concave* if

$$w(i_0, j_0) + w(i_1, j_1) \le w(i_0, j_1) + w(i_1, j_0), \qquad \text{for } 0 \le i_0 < i_1 < j_0 < j_1 \le n. \quad (1)$$

If the weight function is concave then we have an instance of the *concave LWS problem.*

An important instance of the concave LWS problem is the problem of optimally breaking up the text of a paragraph into lines in a text formatting system [2, 3]. The text to be formatted is given as a sequence of syllables

$s_1, \ldots, s_n$, with the space or punctuation following a word being considered to be part of the last syllable of the word. We wish to break up the syllables into one or more lines so that each line other than the last one is filled with "one line's worth" of text—neither too little nor too much. In this case $w(i, j)$ is the penalty associated with a line that starts at syllable s_{i+1} and ends at syllable s_j. Typically $w(i, j)$ would be defined as something like $(\text{len}(i, j) - \text{parwidth})^2$, where $\text{len}(i, j)$ is the "natural length" of a line starting at syllable s_{i+1} and ending at syllable s_j and *parwidth* is the width of the paragraph being formatted. (There are additional complications, such as handling the last line and adding in penalties for hyphenation.) The weight function can be computed in constant time because $\text{len}(i, j)$ can be computed as $\text{length}(j) - \text{length}(i)$, where $\text{length}(i)$ is the natural length of a line with syllables s_1 through s_i. The values for *length* can be computed at the start in $O(n)$ time. Reasonable weight functions for this problem are concave.

Other applications described by Hirschberg and Larmore where the concave LWS problem arises are an airplane refueling problem and minimizing the height of B-trees [2].

The LWS problem can be solved by dynamic programming in $O(n^2)$ time (with or without the concavity constraint). Hirschberg and Larmore [2] give an $O(n \log n)$ time algorithm for the concave LWS problem. They also show that the concave LWS problem can be solved in $O(n)$ time if the weight function meets an additional *difference-of-weight constraint*. The weight function satisfies the difference-of-weight constraint if $w(i, k) = \text{diff}(i, j, \text{scale}(k)) + w(j, k)$, for $0 \leq i < j < k \leq n$, where:

(i) $\text{Scale}(i)$ is a real-valued strictly monotone increasing function defined for integers $i \in [0, n]$.

(ii) $\text{Diff}(i, j, x)$ is a real-valued function defined for integers $0 \leq i < j < n$ and real $x > \text{scale}(j)$, and is monotone increasing in x for fixed i and j.

(iii) For fixed i and j the equation $\text{diff}(i, j, x) = 0$ can be solved for x in constant time.

The first two conditions can always be met by, for example, defining $w(i, i) = W$, where W is a very large value, $\text{scale}(i) = i$, $\text{diff}(i, j, x) = w(i, x) - w(j, x)$ when x is an integer, and linearly interpolating between $\text{diff}(i, j, \lfloor x \rfloor)$ and $\text{diff}(i, j, \lceil x \rceil)$ when x is not an integer. However, with this definition for *diff* there might not be any more efficient way of finding roots than using an $O(\log n)$ time binary search. Hirschberg and Larmore show that for the line breaking problem and the other problems they describe a reasonable choice can be made for the weight function so that it satisfies the difference-of-weight constraint, and thus their linear time algorithm can be used.

For the sake of both generality and aesthetics it is desirable to avoid the need for the difference-of-weight constraint, and use only the concavity property. Here we show that the concave LWS problem can always be solved in $O(n)$ time.

2. The Conventional Algorithm

We first review the standard $O(n^2)$ algorithm for the LWS problem. Let $f(0) = 0$ and for $1 \leq j \leq n$ let $f(j) =$ the weight of the lowest weight subsequence between 0 and j. For $0 \leq i < j \leq n$ define $g(i, j)$ as the weight of the lowest weight subsequence between 0 and j whose next to the last index is i. (That is, the lowest weight subsequence of the form $0 = l_0 < l_1 < \cdots < l_{k-1} = i < l_k = j$.) Then we have

$$f(j) = \min_{0 \leq i < j} g(i, j), \qquad \text{for } 1 \leq j \leq n, \tag{2a}$$

and

$$g(i, j) = f(i) + w(i, j), \qquad \text{for } 0 \leq i < j \leq n. \tag{2b}$$

We may represent g by an upper triangular matrix indexed by row from 0 to $n - 1$ and by column from 1 to n. Equations (2a) and (2b) tell us that we can compute g one column at a time—once the values of g are known for column j we can compute $f(j)$ and then we can compute the values of g for column $j + 1$. In practice when the value of $f(j)$ is determined we also store the value of i for which $g(i, j)$ is minimized; with this information we can recover the optimal sequence in $O(n)$ time after f has been computed. This programming detail will be ignored. The standard algorithm requires computing all $\binom{n+1}{2}$ entries of g so it takes $O(n^2)$ time.

3. The New Algorithm

The linear time algorithm for the concave LWS problem also computes all values of f but does this while computing only $O(n)$ values of g.

Let M be an $n \times m$ real-valued matrix, and let $i(j)$ be the smallest row index such that $M(i(j), j)$ equals the minimum value in the jth column of M. Matrix M is *monotone* if for all $1 \leq j_0 < j_1 \leq m$ we have $i(j_0) \leq i(j_1)$.

M is *totally monotone* if every submatrix of M is monotone. It is easy to see that this is equivalent to the condition that every 2×2 submatrix of M is monotone. Aggarwal *et al.* [1] describe an algorithm that, given an $n \times m$ totally monotone matrix, computes $i(j)$ for each $j \in [1, m]$ in only $O(m + n)$ time.[1] We will use this algorithm as a subroutine (and will call it the SMAWK algorithm). In applications the $n \times m$ matrix is not represented explicitly; instead there is a subroutine that given i and j computes the value of $M(i, j)$. The proof of the linear time bound assumes that this subroutine works in $O(1)$ time.

If we add $f(i_0) + f(i_1)$ to both sides of (1) and apply (2b) we get

$$g(i_0, j_0) + g(i_1, j_1) \le g(i_0, j_1) + g(i_1, j_0), \qquad \text{for } 0 \le i_0 < i_1 < j_0 < j_1 \le n. \tag{3}$$

Let W be some large value ($1 + n \cdot \max_{i,j} w(i, j)$ will do). We may extend the definition of g by setting

$$g(i, j) = W, \qquad \text{for } 1 \le j \le i \le n - 1. \tag{4}$$

Now g can be regarded as an $n \times n$ matrix and (3) and (4) imply that it is totally monotone. The goal is to determine the row index of the minimum value in each column of g, so we would like to simply apply the SMAWK algorithm. But we cannot, because for $i < j$ the value of $g(i, j)$ depends upon $f(i)$ which depends upon all values of $g(l, i)$ for $0 \le l < i$. So we cannot compute the value of an arbitrary cell of g in $O(1)$ time.

The trick is to start in the upper left corner of g and work rightwards and downwards, at each iteration learning enough new values for f to be able to compute enough new values of g to continue with the next iteration. Actually, during one step of each iteration the algorithm operates on the basis of wishful thinking—it "pretends" to know values of f that it really does not have. At the end of that step the assumed values of f are checked for validity. If the assumed values are correct we win by learning some new values of f. If one of the assumed values is wrong we win anyway by eliminating from further consideration some of the rows of g.

In discussing the algorithm we revert to regarding g as an upper triangular matrix; the values of g defined by (4) are used when needed by the SMAWK algorithm and are otherwise ignored. We use $f(j)$ and $g(i, j)$ to refer to the correct values of f and g, as defined by (2a) and (2b). The currently computed value for $f(j)$ is denoted by $F[j]$, and will sometimes

[1] Aggarwal *et al.* defined monotonicity in terms of the position of the maximum in each row rather than the minimum in each column; we simply exchange the roles of rows and columns and reverse the direction of some of the comparisons in their algorithm.

```
$F[0] \leftarrow c \leftarrow r \leftarrow 0.$
while $(c < n)$
begin
    Step 1: $p \leftarrow \min(2c - r + 1, n)$
    Step 2: Apply the SMAWK algorithm to find the minimum in each column of submatrix
            $G[r, c; c+1, p]$. For $j \in [c+1, p]$ let $F[j]$ = the minimum value found in $G[r, c; j]$.
    Step 3: Apply the SMAWK algorithm to find the minimum in each column of submatrix
            $G[c+1, p-1; c+2, p]$. For $j \in [c+2, p]$ let $H[j]$ = the minimum value found in
            $G[c+1, p-1; j]$.
    Step 4: If there is an integer $j \in [c+2, p]$ such that $H[j] < F[j]$ then set $j_0$ to the
            smallest such integer. Otherwise $j_0 \leftarrow p + 1$.
    Step 5: if $(j_0 = p + 1)$
                then $c \leftarrow p$.
                else $F[j_0] \leftarrow H[j_0]$; $r \leftarrow c + 1$; $c \leftarrow j_0$.
end
```

FIG. 1. The linear time algorithm for the concave LWS problem.

be incorrect. The currently computed value of $g(i, j)$ is denoted by $G[i, j]$, and for $i < j$ is always computed as $F[i] + w(i, j)$ (there is no need to explicitly store the G matrix). So $G[i, j] = g(i, j)$ iff $F[i] = f(i)$. We use $G[i_1, i_2; j_1, j_2]$ to denote the submatrix of G consisting of the intersection of rows i_1 through i_2 and columns j_1 through j_2. $G[i_1, i_2; j]$ denotes the intersection of rows i_1 through i_2 with column j.

The algorithm is shown in Fig. 1. (Remember that rows are indexed from 0 and columns are indexed from 1.) Each time we are at the beginning of the loop the following invariants hold:

(1) $r \geq 0$ and $c \geq r$.

(2) For each $j \in [0, c]$, $F[j] = f(j)$.

(3) All minima in columns $c + 1$ through n of g are in rows $\geq r$.

These invariants are clearly satisfied at the start when $r = c = 0$.

Let S denote the submatrix $G[r, c; c + 1, p]$ (used in Step 2), and let T denote the upper triangular submatrix $G[c + 1, p - 1; c + 2, p]$ (used in Step 3). Figure 2 shows matrix G and submatrices S and T during a typical iteration of the algorithm.

Invariant (2) implies that $G[i, j] = g(i, j)$ for all j and all $i \in [0, c]$ so the entries of submatrix S are the same as the corresponding entries of g.

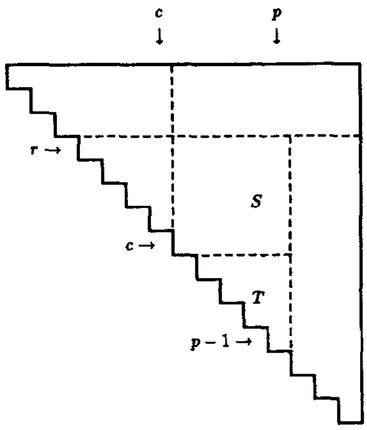

FIG. 2. Matrix G during a typical iteration of the algorithm, with $r = 3$ and $c = 7$.

Therefore S is totally monotone and for $j \in [c + 1, p]$, Step 2 sets $F[j]$ to the minimum value of subcolumn $g(r, c; j)$. Also, since submatrix S contains all cells in column $c + 1$ of g that are in rows $\geq r$ we have $F[c + 1] = f(c + 1)$ at the end of Step 2. On the other hand, we do not necessarily have $F[j] = f(j)$ for any $j \in [c + 2, p]$, since g has cells in those columns that are in rows $\geq r$ and not in submatrix S.

In Step 3 we proceed as if $F[j] = f(j)$ for all $j \in [c + 1, p - 1]$. Since this may be false, some of the values in T may be bogus. However, T is always totally monotone, for if we add $F[i_0] + F[i_1]$ to both sides of (1) we get $G[i_0, j_0] + G[i_1, j_1] \leq G[i_0, j_1] + G[i_1, j_0]$—it does not matter whether or not $F[i_0] = f(i_0)$ and $F[i_1] = f(i_1)$. Thus the SMAWK algorithm works correctly and $H[j]$ is set to the minimum value of subcolumn $G[c + 1, p - 1; j]$ (which is not necessarily the same as the minimum value of subcolumn $g(c + 1, p - 1; j)$). (If $r = c$ then T has 0 rows and 0 columns. In that case Step 3 does nothing and Step 4 sets j_0 to $p + 1$.)

In Step 4 we verify that $F[j] = f(j)$ for $j \in [c + 2, p]$, or else find the smallest j where this condition fails. The first column of T has just one cell, $G[c + 1, c + 2]$. Since $F[c + 1] = f(c + 1)$ we have $G[c + 1, j] = g(c + 1, j)$ for all j and, in particular, for $j = c + 2$. So $H[c + 2]$ is the minimum value in subcolumn $g(c + 1, p - 1; c + 2)$. Thus if $H[c + 2] \geq F[c + 2]$ then $F[c + 2] = f(c + 2)$. If that is the case then $G[c + 2, j] = g(c + 2, j)$ for all j so both cells in the second column of T are correct, and $H[c + 3]$ is the minimum value in subcolumn $g(c + 1, p - 1; c + 3)$. Continuing in the same way we see that if $H[c + 3] \geq F[c + 3]$ then $F[c + 3] = f(c + 3)$ and all cells in the third column of T are correct, and so on. So if for all $j \in [c + 2, p]$ we have $H[j] \geq F[j]$ then for all $j \in [0, p]$ we have $F[j] = f(j)$. In that case the **then** clause of the **if** statement in Step 5 is executed. Since c is set to p invariant (2) is satisfied at the start of the next iteration. Since r is not changed in this case invariant (3) and the first part of invariant (1) are still satisfied. Also since $p > r$ we have $c > r$ and the second part of invariant (1) is satisfied.

The other case is that we find $j_0 \in [c+2, p]$ such that $H[j_0] < F[j_0]$ and $H[j] \geq F[j]$ for $j \in [c+2, j_0 - 1]$. Then by the reasoning above at the end of Step 4 we have $F[j] = f(j)$ for all $j < j_0$ and $H[j_0] =$ the minimum value in subcolumn $g(c+1, p-1; j_0)$. In Step 5 the **else** clause is executed and after $F[j_0]$ is set to $H[j_0]$ we have $F[j] = f(j)$ for all $j \leq j_0$, so invariant (2) still holds after c is set to j_0. The minimum value in column j_0 of g was found in a row $\geq c + 1$, so since g is totally monotone we know that all minima in columns $> j_0$ are in rows $\geq c + 1$. Thus invariant (3) holds after r is set to $c + 1$ and c is set to j_0. The value of r is increased so $r > 0$ and the first part of invariant (1) is satisfied. Since $j_0 \geq c + 2$ (using the old value of c) we have the new value of $c > r$, so the second part of invariant (1) is satisfied.

The **then** clause of Step 5 leaves $c = p \leq n$ and the **else** clause leaves $c = j_0 \leq n$ so when the **while** loop terminates we must have $c = n$. Then invariant (2) implies that $F[j] = f(j)$ for all $j \in [0, n]$ and we are done.

4. The Time Bound

We show that the algorithm terminates within $O(n)$ time. On any given iteration submatrix S has $c - r + 1$ rows and at most $c - r + 1$ columns. Submatrix T has at most $c - r$ rows and $c - r$ columns. The SMAWK algorithm runs in linear time so the time taken by a single iteration can be bounded by $b \cdot (c - r + 1)$, for some constant b. Say that an iteration is ***abnormal*** if $p \neq 2c - r + 1$ and the **then** clause of Step 5 is executed, and is ***normal*** otherwise. In an abnormal iteration we have $p = n$ and at the end c is set to p, so that $c = n$. Thus the algorithm terminates after the first abnormal iteration (if any) and the total time used by abnormal iterations is $O(n)$. To bound the cost of normal iterations let $\phi(i) =$ the value of $r + c$ at the end of the ith iteration ($\phi(0) = 0$). Since at the end of each iteration $r \leq c \leq n$ we always have $\phi(i) \leq 2n$. We claim that if the ith iteration is normal $\phi(i) \geq \phi(i-1) + c - r + 1$, where the values of c and r are from the start of the ith iteration. This claim implies that for all i the total time used by the first i normal iterations is at most $b \cdot \phi(i)$, so the time taken by all normal iterations is at most $2bn$. To prove the claim, consider first the case where the **then** clause of Step 5 is executed. Since the iteration is normal $p = 2c - r + 1$ so c is replaced by $2c - r + 1$, an increase of $c - r + 1$, and r is unchanged. Thus $\phi(i) = \phi(i-1) + c - r + 1$. If instead the **else** clause is executed c is increased by at least 2 (since $j_0 \geq c + 2$) and r is increased by $c - r + 1$, so $\phi(i) \geq \phi(i-1) + c - r + 3$. So the total time taken by all iterations is $O(n)$.

5. Practical Efficiency Considerations

If one has an instance of the concave LWS problem for which the difference-of-weight constraint can be satisfied (and the constant bounding the time to solve $\text{diff}(i, j, x) = 0$ is reasonably small) then Hirschberg and Larmore's [2] $O(n)$ time algorithm is the method of choice. Otherwise the best method depends on the value of n. Hirschberg and Larmore's general $O(n \log n)$ time algorithm is simple and has a small constant factor buried in the "O," whereas the theoretically superior linear time algorithm given here uses the SMAWK algorithm, which has a fairly large constant factor.

For each $j \in [1, n]$ let $\text{prev}(j)$ be the value of $i \in [0, j-1]$ that minimizes $g(i, j)$, and let $w = \max_{1 \le j \le n}(j - \text{prev}(j))$. For the line-breaking problem n is the number of syllables in the paragraph to be formatted and w is roughly the maximum number of consecutive syllables that fit on a single line without crowding. Hirschberg and Larmore's algorithm [2] can easily be modified to run in $O(n \log w)$ time. There is a simple divide-and-conquer procedure for finding the minima of an $l \times m$ totally monotone matrix in time $O(l \log m)$, with a small constant factor inside the "O." For values of w that are not very large the algorithm described here will run faster if the calls to the SMAWK procedure are replaced by calls to the divide and conquer procedure. This version of the algorithm runs in $O(n \log w)$ time, because $c - r \le w$ at each iteration. It is quite different from Hirschberg and Larmore's algorithm, and like their algorithm it has a moderate sized constant factor.

Acknowledgment

Peter Shor suggested looking at Hirschberg and Larmore's paper [2] and seeing if their $O(n \log n)$ time algorithm could be improved upon.

References

1. A. Aggarwal, M. M. Klawe, S. Moran, P. Shor, and R. Wilber, Geometric applications of a matrix searching algorithm, *Algorithmica* **2** (1987), 195–208.
2. D. S. Hirschberg and L. L. Larmore, The least weight subsequence problem, *SIAM J. Comput.* **16** (1987), 628–638.
3. D. E. Knuth and M. F. Plass, Breaking paragraphs into lines, *Software Practice Experience* **11** (1981), 1119–1184.