Algorithmica (1993) 9: 615–628

Algorithmica

Finding Least-Weight Subsequences with Fewer Processors

Tak Wah Lam[1] and Kwong-fai Chan[1]

Abstract. By restricting weight functions to satisfy the quadrangle inequality or the inverse quadrangle inequality, significant progress has been made in developing efficient sequential algorithms for the least-weight subsequence problem [10], [9], [12], [16]. However, not much is known on the improvement of the naive parallel algorithm for the problem, which is fast but demands too many processors (i.e., it takes $O(\log^2 n)$ time on a CREW PRAM with $n^3/\log n$ processors). In this paper we show that if the weight function satisfies the inverse quadrangle inequality, the problem can be solved on a CREW PRAM in $O(\log^2 n \log\log n)$ time with $n/\log\log n$ processors, or in $O(\log^2 n)$ time with $n \log n$ processors. Notice that the processor-time complexity of our algorithm is much closer to the almost linear-time complexity of the best-known sequential algorithm [12].

Key Words. Parallel algorithms, Dynamic programming, Monotone matrix.

1. Introduction.

The least-weight subsequence problem was first defined by Hirschberg and Larmore [10] as follows: Given an integer n and a real-valued weight function $w(i, j)$ which can be computed in constant time for all $0 \le i < j \le n$, find a sequence of integers $\{\alpha_1, \alpha_2, \ldots, \alpha_m\}$ such that $0 = \alpha_1 < \alpha_2 < \cdots < \alpha_m = n$ and $\sum_{1 \le i < m} w(\alpha_i, \alpha_{i+1})$ is minimized. Such a sequence is called a least-weight subsequence for $[0, n]$.

The weight function is said to be concave if it satisfies the quadrangle inequality [17], i.e., for all $i_0 \le i_1 < j_0 \le j_1$, $w(i_0, j_0) + w(i_1, j_1) \le w(i_0, j_1) + w(i_1, j_0)$. The weight function is said to be convex if it satisfies the inverse quadrangle inequality, i.e., for all $i_0 \le i_1 < j_0 \le j_1$, $w(i_0, j_0) + w(i_1, j_1) \ge w(i_0, j_1) + w(i_1, j_0)$.

By imposing the concave property on the weight function, Hirschberg and Larmore [10] improved the sequential-time complexity of finding the least-weight subsequence from n^2 to $n \log n$. Later, Galil and Giancarlo [9] generalized this result to both the concave case and the convex case. Though the quadrangle inequality and its inverse look very similar, an algorithm working for the concave case might not work in the convex case, and vice versa. In fact, a linear-time algorithm for the concave least-weight subsequence problem was found by Wilber [16] and an improved linear-time algorithm was found by Eppstein [6], yet the best-known algorithm for the convex case, which was due to Klawe and Kleitman [12], still requires $O(n\alpha(n))$ time, where $\alpha(n)$ is the inverse Ackermann's function.

Let us switch to parallel algorithms. Using standard techniques for contracting trees [14], [15], we can find the least-weight subsequence in $O(\log^2 n)$ time on a CREW (concurrent read, exclusive write) PRAM with $n^3/\log n$ processors. Details are shown in Section 2. Recently Atallah, *et al.* [4] have shown that two concave

---

[1] Department of Computer Science, University of Hong Kong, Pokfulam, Hong Kong.

Received November 6, 1990; revised May 10, 1991. Communicated by Takao Nishizeki.

matrices can be multiplied in $O(\log n \log\log n)$ time on a CREW PRAM with $n^2/\log n$ processors. As the least-weight subsequence problem can be reduced to a series of $\log n$ matrix multiplications, we immediately obtain a parallel algorithm using $O(\log^2 n \log\log n)$ time and $n^2/\log n$ processors for the concave case. This algorithm can be adapted to the convex case, too.

Other than these results, little was previously known on the parallel complexity of the problem when the weight function satisfies either the concave property or the convex property. Due to the apparent sequential nature of the problem, a brute-force approach may seem to be the only way to go, that is, a lot of processors are required to perform redundant computation, and the total work becomes very unfavorable. As far as we know, parallel solutions to other dynamic programming problems, such as finding the optimal binary search tree, also have this kind of defect. In this paper we show a new parallel algorithm for the convex least-weight subsequence problem, which can be implemented in $O(\log^2 n \log\log n)$ time with $n/\log\log n$ processors, or in $O(\log^2 n)$ time with $n \log n$ processors.

The parallel computation model used in this paper is the CREW PRAM. A PRAM basically consists of a large number of independent processors which communicate through a global memory [8]. According to the restrictions on the access of the global memory, PRAMs can be further classified. In the CREW PRAM, more than one processor can simultaneously read the same global memory cell, but simultaneous write into the same memory cell is not allowed. More details of the model can be found in the survey by Karp and Ramachandran [11].

In the following, unless it is stated explicitly, we assume that the weight function is convex.

The remainder of this paper is organized as follows. Section 2 describes a naive parallel algorithm for this problem. Section 3 defines the maximal least-weight subsequence, which is shown to have some nice properties to enable us to identify it in parallel easily. Section 4 depicts a straightforward algorithm implementing the idea stated in Section 3. Finally, Section 5 further improves the processor complexity of the algorithm by making use of a known result in monotone matrix-searching [2], [3].

Before our discussion of the algorithms, let us look at the formal definition of least-weight subsequences. We overload the set notation $\{\alpha_1, \alpha_2, \ldots, \alpha_m\}$ to denote a sequence of integers $\alpha_1 < \alpha_2 < \cdots < \alpha_m$.

DEFINITION. For any $0 \le i < j \le n$, a subsequence in the interval $[i, j]$ is any sequence of integers $S = \{\alpha_1, \alpha_2, \ldots, \alpha_m\}$ with $\alpha_1 = i$ and $\alpha_m = j$.

- Let $w(S)$ denote the total weight of the subsequence S, i.e., $\sum_{1 \le l < m} w(\alpha_l, \alpha_{l+1})$.
- S is a least-weight subsequence (LWS) for $[i, j]$ if $w(S) = \min\{w(S') | S'$ is a subsequence in $[i, j]\}$.
- Let $e(i, j)$ denote the total weight of any LWS for $[i, j]$. Define $e(i, i) = 0$ for all $0 \le i \le n$.

2. The Conventional Approach. Figure 1 shows a simple parallel algorithm solving the LWS problem without any restriction on the weight function. It requires $n^3/\log n$ processors and $O(\log^2 n)$ time. The algorithm consists of two

```
For all 0 ≤ i ≤ j ≤ n do in parallel
  { if i = j then E[i, j] = 0 else E[i, j] = w(i, j); }

Stage 1—computing e(i, j)'s

Repeat ⌈log n⌉ times
  for all i, j such that 0 ≤ i, j ≤ n and j − i > 1 do in parallel
  { temp ← min_{i<k<j}(E[i, k] + E[k, j]);
    if temp < E[i, j] then
    { E[i, j] ← temp;
      K[i, j] ← k, where temp = E[i, k] + E[k, j];
      /* choose an arbitrary one if more than one such k exists */ } }

Stage 2—backward tracing

B[0, n] ← 1;
Repeat ⌈log n⌉ times
  for all i, j such that 0 ≤ i < j ≤ n do in parallel
  if B[i, j] = 1 then
  { S[K[i, j]] ← 1;
    B[i, K[i, j]] ← 1; B[K[i, j], j] ← 1;
    B[i, j] ← 0;   /* to avoid concurrent write */ }
```

Fig. 1. Algorithm $\mathscr{A}$—a naive parallel algorithm for finding the LWS.

stages: stage 1 computes the value of $e(i, j)$ for all i, j; stage 2 then constructs an LWS for $[0, n]$. The result is eventually stored in an array S such that $S[i] = 1$ if and only if i is an element of the LWS. Since any LWS for $[0, n]$ ends with 0 and n, we initialize $S[0]$, $S[n]$ to 1 and $S[i]$ to 0 for all other i's.

In the algorithm an $(n + 1) \times (n + 1)$ matrix E is used to store all $e(i, j)$'s. Note that, for any $0 \le i < j \le n$, if $e(i, j)$ is less than $w(i, j)$, then $e(i, j)$ can be expressed as $e(i, k) + e(k, j)$ for some k strictly between i and j. Thus, we initialize each $E[i, j]$ to $w(i, j)$. In each iteration the current value of each $E[i, j]$ is compared with that of $(E[i, k] + E[k, j])$ for all k strictly between i and j. $E[i, j]$ is always updated with the smallest value. If there is an LWS for $[i, j]$ consisting of at most $2^l + 1$ elements, then $E[i, j]$ will be equal to $e(i, j)$ after the lth iteration. In other words, every $E[i, j]$ will have received the value of $e(i, j)$ after the $\lceil \log n \rceil$th iteration.

An array K is used to keep the history of how the smallest value of each $E[i, j]$ is obtained. Initially, each $K[i, j]$ has the value i. Whenever $E[i, j]$ is updated to the value $(E[i, k] + E[k, j])$ for some k, the value k will be stored into $K[i, j]$. After stage 1, the value in each $E[i, j]$ should become $e(i, j)$. We then retrieve the LWS for $[0, n]$ by tracing the array K in a backward manner.

3. The Maximal Least-Weight Subsequence. It is obvious that there may be more than one distinct LWS in any particular interval $[i, j]$. Even if we can find an LWS for the first half of the interval (i.e., $[i, (i + j)/2]$) and one for the other half (i.e. $[(i + j)/2, j]$), it is not necessary that they can be combined to form an LWS for the whole interval. In other words, to apply the approach of "divide and conquer" to this problem, we need to find all LWSs for each half interval, and then examine which pair can be combined to form a final solution. Of course, it requires a lot

of processors in order to compute all LWSs for the subintervals efficiently. The convex property of the weight function, however, guarantees the existence of a particular LWS which can be found by the "divide and conquer" approach with much fewer processors. The details are as follows:

LEMMA 1. *For any* $0 \leq i < j \leq n$, *let* S_1 *and* S_2 *be any two LWSs for* $[i, j]$, *then the union of* S_1 *and* S_2 *is also an LWS for* $[i, j]$.

PROOF. See Appendix 1. □

By applying Lemma 1 inductively, we can show that the union of all LWSs for $[i, j]$ is also an LWS, which is called the maximal LWS for $[i, j]$ and is denoted by $MS_{i,j}$. The essence of this paper is that the maximal LWS can be found efficiently with a linear number of processors. Now let us look at some properties of the maximal LWS.

FACT 2. *For any* $0 \leq i < j \leq n$, *let* $MS_{i,j} = \{\alpha_1, \alpha_2, \ldots, \alpha_m\}$. *Then, for any* $1 \leq r < s \leq m$, *the maximal LWS for* $[\alpha_r, \alpha_s]$ *is exactly* $\{\alpha_r, \alpha_{r+1}, \ldots, \alpha_s\}$.

Moreover, even for any pair k, l with $i \leq k < l \leq j$, not necessarily chosen from $\{\alpha_1, \ldots, \alpha_m\}$, $MS_{k,l}$ still contains all elements of $MS_{i,j}$ within the interval $[k, l]$. Figure 2 depicts this relationship. To be more specific, let us look at the following lemma.

LEMMA 3. *Given any* $0 \leq i < j \leq n$, *let* x *be an element of the maximal LWS for* $[i, j]$. *Then, for all* k, l *with* $i \leq k < l \leq j$ *and* $k \leq x \leq l$, x *is also an element of the maximal LWS for* $[k, l]$.

PROOF. See Appendix 2. □

Perhaps the most interesting property is about the ease of constructing the maximal LWS for a longer interval from two maximal LWSs of consecutive shorter intervals.

THEOREM 4. *For any* i, k, j *with* $0 \leq i < k < j \leq n$, *let* $MS_{i,k} = \{\alpha_1, \alpha_2, \ldots, \alpha_{m_1}\}$ *and* $MS_{k,j} = \{\beta_1, \beta_2, \ldots, \beta_{m_2}\}$. *Then there exist* $1 \leq h < m_1$ *and* $1 \leq l \leq m_2$ *such that* $MS_{i,j} = \{\alpha_1, \alpha_2, \ldots, \alpha_h, \beta_l, \beta_{i+1}, \ldots, \beta_{m_2}\}$ (see Figure 3).

PROOF. Let $MS_{i,j} = \{\gamma_1, \gamma_2, \ldots, \gamma_{m_3}\}$. Let γ_r be the largest element in $MS_{i,j}$ strictly less than k. By Lemma 3, γ_1, γ_2, ..., and γ_r must be elements of $MS_{i,k}$. Given that $MS_{i,k} = \{\alpha_1, \alpha_2, \ldots, \alpha_{m_1}\}$, we let α_h denote the element in $MS_{i,k}$ equal to γ_r.

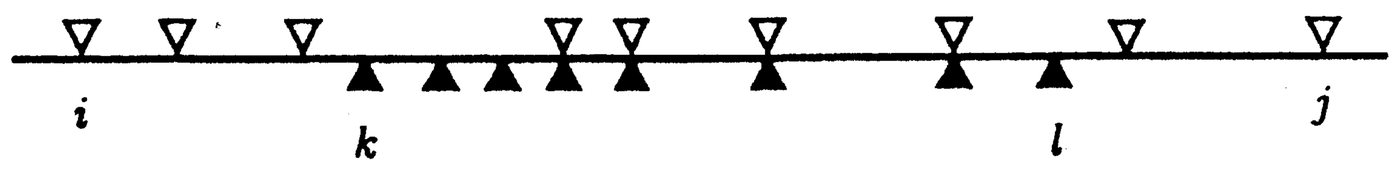

Fig. 2. The relationship between $MS_{i,j}$ and $MS_{k,l}$. ▽ is the element of the maximal LWS for $[i, j]$ and ▲ is the element of the maximal LWS for $[k, l]$.

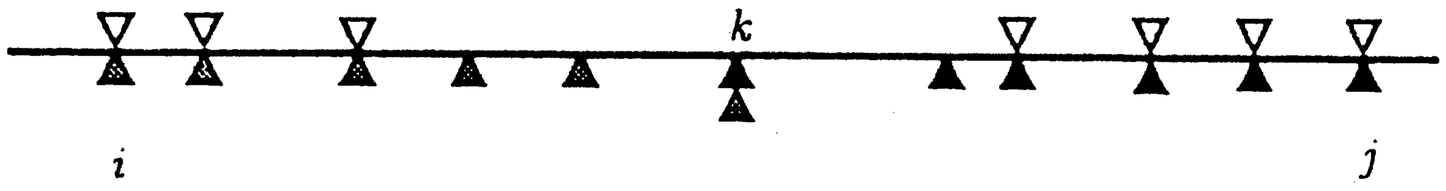

Fig. 3. $MS_{i,j}$ is composed of $MS_{i,k}$ and $MS_{k,j}$. ∇ is the element of the maximal LWS for $[i,j]$, /▲ is the element of the maximal LWS for $[i,k]$, and ▲ is the element of the maximal LWS for $[k,j]$.

As γ_{r+1} is the smallest element in $MS_{i,j}$ greater than or equal to k, by Lemma 3 again, we conclude that $\gamma_{r+1}, \ldots, \gamma_{m_3}$ is in $MS_{k,j}$. Let β_l be the element in $MS_{k,j}$ such that $\beta_l = \gamma_{r+1}$.

We can deduce from Fact 2 that $\{\alpha_1, \ldots, \alpha_h\} = MS_{i,\alpha_h} = \{\gamma_1, \ldots, \gamma_r\}$, and $\{\beta_l, \ldots, \beta_{m_2}\} = MS_{\beta_l,j} = \{\gamma_{r+1}, \ldots, \gamma_{m_3}\}$. Therefore, $\{\gamma_1, \gamma_2, \ldots, \gamma_{m_3}\}$ is equal to $\{\alpha_1, \alpha_2, \ldots, \alpha_h, \beta_l, \beta_{l+1}, \ldots, \beta_{m_2}\}$. □

Basically, Theorem 4 states that, for any $i < k < j$, $MS_{i,j}$ is composed of the first h elements of $MS_{i,k}$ and the last $(m_2 - l + 1)$ elements of $MS_{k,j}$. Now the question is how to locate α_h and β_l. $MS_{i,j}$ is the maximal LWS for $[i,j]$, so intuitively, we should find the largest h and the smallest l, which make the total weight of the sequence $\{\alpha_1, \ldots, \alpha_h, \beta_l, \ldots, \beta_{m_2}\}$ be the minimum over all possible choices of h and l. Corollary 5 states formally the conditions for choosing h and l.

COROLLARY 5. *For any i, k, j with $0 \le i < k < j \le n$, let $MS_{i,k} = \{\alpha_1, \alpha_2, \ldots, \alpha_{m_1}\}$ and $MS_{k,j} = \{\beta_1, \beta_2, \ldots, \beta_{m_2}\}$. Let h, l be integers, with $1 \le h < m_1$ and $1 \le l \le m_2$, satisfying the following two conditions:*

1.

$$w(\{\alpha_1, \alpha_2, \ldots, \alpha_h, \beta_l, \beta_{l+1}, \ldots, \beta_{m_2}\}) = \min_{1 \le h' < m_1,\ 1 \le l' \le m_2} w(\{\alpha_1, \alpha_2, \ldots, \alpha_{h'}, \beta_{l'}, \beta_{l'+1}, \ldots, \beta_{m_2}\}),^2$$

2. *For all h', l' with $h \le h' < m_1$ and $1 \le l' \le l$, if $h' \ne h$ or $l' \ne l$, then*

$$w(\{\alpha_1, \alpha_2, \ldots, \alpha_{h'}, \beta_{l'}, \beta_{l'+1}, \ldots, \beta_{m_2}\}) > w(\{\alpha_1, \alpha_2, \ldots, \alpha_h, \beta_l, \beta_{l+1}, \ldots, \beta_{m_2}\}).$$

Then $MS_{i,j} = \{\alpha_1, \alpha_2, \ldots, \alpha_h, \beta_l, \beta_{l+1}, \ldots, \beta_{m_2}\}$.

PROOF. Let $S = \{\alpha_1, \alpha_2, \ldots, \alpha_h, \beta_l, \beta_{l+1}, \ldots, \beta_{m_2}\}$, where h, l are integers satisfying the conditions stated above. From Theorem 4, we know that there exist $1 \le h_0 < m_1$ and $1 \le l_0 \le m_2$ such that $MS_{i,j} = \{\alpha_1, \alpha_2, \ldots, \alpha_{h_0}, \beta_{l_0}, \beta_{l_0+1}, \ldots, \beta_{m_2}\}$. According to the definition of maximal LWS, it is clear that

$$w(MS_{i,j}) = \min_{1 \le h' < m_1,\ 1 \le l' \le m_2} w(\{\alpha_1, \alpha_2, \ldots, \alpha_{h'}, \beta_{l'}, \beta_{l'+1}, \ldots, \beta_{m_2}\}).$$

---

[2] Since $MS_{\alpha_1,\alpha_{h'}} = \{\alpha_1, \ldots, \alpha_{h'}\}$ and $MS_{\beta_{l'},\beta_{m_2}} = \{\beta_{l'}, \ldots, \beta_{m_2}\}$, the value of $w(\{\alpha_1, \ldots, \alpha_{h'}, \beta_{l'}, \ldots, \beta_{m_2}\})$ can be computed by taking the sum of $e(\alpha_1, \alpha_{h'})$, $w(\alpha_{h'}, \beta_{l'})$, and $e(\beta_{l'}, \beta_{m_2})$.

Referring to the first condition for choosing h and l, we see that $w(S) = w(MS_{i,j})$. Thus, S is an LWS for $[i, j]$, and S must be a subset of $MS_{i,j}$. In other words, $h_0 \geq h$ and $l_0 \leq l$. Due to the second condition for choosing h and l, if $h_0 \neq h$ or $l_0 \neq l$, we get $w(MS_{i,j}) > w(S)$, which is a contradiction. Therefore, $h_0 = h$ and $l_0 = l$, or, equivalently, $MS_{i,j} = \{\alpha_1, \alpha_2, \ldots, \alpha_h, \beta_l, \beta_{l+1}, \ldots, \beta_{m_2}\}$. □

4. A Simple Algorithm. Corollary 5 enables us to compute the maximal LWS for $[0, n]$ in a recursive manner, that is, first compute the maximal LWSs for two shorter intervals $[0, n/2]$ and $[n/2, n]$, then combine their elements selectively. Algorithm $\mathscr{B}$, as shown in Figure 4, is a straightforward implementation of this idea. It takes $O(\log^2 n)$ time on a CREW PRAM with $n^2/\log n$ processors. Although the improvement of the processor complexity is not significant, it provides a framework for building a better algorithm with linear processor complexity.

For simplicity, we assume that n is a power of 2. The backbone of Algorithm $\mathscr{B}$ is a recursive procedure called *maximal_lws* which on parameters (i, t) computes the maximal LWS for $[i, i + n/2^t]$. The result is stored in the array $next[i..i + n/2^t]$ in such a way that if x belongs to the maximal LWS for $[i, i + n/2^t]$, then $next[x]$ points to the next element after x in the maximal LWS for $[i, i + n/2^t]$, otherwise $next[x]$ is set to ∞.

The main body of procedure *maximal_lws* consists of seven steps. Step 1 divides the interval $[i, i + n/2^t]$ into two equal-size subintervals $[i, i + n/2^{t+1}]$ and $[i + n/2^{t+1}, i + n/2^t]$, and computes the maximal LWS for each subinterval recurs-

To find the maximal LWS for $[0, n]$, we call the procedure *maximal_lws*$(0, 0)$.

Constant $t_0 = \lfloor \log(n/\log n) \rfloor$.

Procedure *maximal_lws*(i, t)
If $t = t_0$ then
  Find the maximal LWS for $[i, i + n/2^t]$ sequentially and then update $next[i]$, $next[i + 1], \ldots, next[i + n/2^t - 1]$.
else (* $t < t_0$ *)

1. $k \leftarrow i + n/2^{t+1}$; $j \leftarrow i + n/2^t$;
   call *maximal_lws*$(i, t + 1)$, *maximal_lws*$(k, t + 1)$ in parallel.
2. For all $i \leq \alpha' < k$, if $next[\alpha'] \neq \infty$, then compute $e(i, \alpha')$ which is the sum of all $w(x, next[x])$, where $i \leq x < \alpha'$ and $next[x] \leq \alpha'$.
3. For all $k \leq \beta' \leq j$, if $next[\beta'] \neq \infty$, then compute $e(k, \beta')$ which is the sum of all $w(x, next[x])$, where $k \leq x < \beta'$ and $next[x] \leq \beta'$.
4. For all $k \leq \beta' \leq j$, if $next[\beta'] \neq \infty$, then $e(\beta', j) \leftarrow e(k, j) - e(k, \beta')$.
5. For all $i \leq \alpha' < k$, if $next[\alpha'] \neq \infty$, compute $mate(\alpha')$, that is the smallest β' in the range $[k, j]$, which minimizes the value $e(i, \alpha') + w(\alpha', \beta') + e(\beta', j)$.
6. Compute α_h, that is the largest α' with $i \leq \alpha' < k$ and $next[\alpha'] \neq \infty$, which minimizes the value $e(i, \alpha') + w(\alpha', mate(\alpha')) + e(mate(\alpha'), j)$.
7. β_l, $next[\alpha_h] \leftarrow mate(\alpha_h)$;
   for all $\alpha_h < x < \beta_l$, $next[x] \leftarrow \infty$.

Fig. 4. Algorithm $\mathscr{B}$—a CREW parallel algorithm solving the convex LWS problem in $O(\log^2 n)$ time with $n^2/\log n$ processors.

ively in parallel. Then steps 2–6 compute the maximal LWS for $[i, i + n/2^t]$ according to the conditions stated in Corollary 5. Step 7 updates the array *next*.

We now analyze the processor and time bounds of Algorithm $\mathscr{B}$. The depth of recursion is clearly less than $\log n$. At the first level of recursion, there is only one instance of *maximal_lws* to be executed; in general, at the $(t + 1)$th level, there are 2^t instances of *maximal_lws* to be executed in parallel, each of which works on an interval of size $n/2^t$.

As shown in Figure 4, t_0 denotes the integer $\lfloor \log(n/\log n) \rfloor$. We now consider the situation at the $(t + 1)$th level of recursion for any $t < t_0$. Each of the 2^t parallel instances of *maximal_lws* is going to execute steps 1–7 in Figure 4. Let $m = n/2^t$, i.e., the size of the interval worked by each instance. Both steps 2 and 3 are some sort of parallel prefix computation [13] and can be implemented in $O(\log m)$ time with $m/\log m$ processors [1], [11]. Other steps are simpler. Steps 4, 6, and 7 can be done in $O(\log m)$ time with $m/\log m$ processors; step 5 requires $m^2/\log m$ processors though the time is still $O(\log m)$. Using Brent's scheduling principle [5], we can reduce the number of processors for all the steps except step 5 to $m/\log n$ by assigning one processor to simulate $O(\log n/\log m)$ processors. Similarly, the processor requirement of step 5 can be reduced to at most $m^2/\log n$. The time required for each step, however, becomes $O(\log n)$. Sum over all 2^t instances of *maximal_lws*, the total number of processors used is bounded above by $2^t m^2/\log n$, which is actually $(n^2/2^t)/\log n$. In summary, Algorithm $\mathscr{B}$ at any level of recursion, except the last one, requires $O(\log n)$ time and at most $n^2/\log n$ processors.

When the recursion reaches the last level, there are 2^{t_0} instances of *maximal_lws* to be executed in parallel, each for an interval of size $n/2^{t_0}$. For each instance of *maximal_lws*, we simply assign one processor to find the corresponding maximal LWS sequentially in a brute-force way [10], [16]. This takes $O((n/2^{t_0})^2)$ time. As $t_0 = \lfloor \log(n/\log n) \rfloor$, so $n/2^{t_0} < 2 \log n$ and $2^{t_0} \leq n/(\log n)$. Thus, the time needed is $O(\log^2 n)$, and the processor requirement over all 2^{t_0} instances is at most $n/\log n$.

On the whole, Algorithm $\mathscr{B}$ requires $O(\log^2 n)$ time and $n^2/\log n$ processors. Moreover, if we exclude step 5, $n/\log n$ processors are indeed sufficient for the rest of the algorithm.

To prove the correctness of Algorithm $\mathscr{B}$, we show by backward induction on t that the procedure *maximal_lws*$(i, i + n/2^t)$ stores the maximal LWS for $[i, i + n/2^t]$ in the array *next* correctly. The basis where $t = t_0$ is trivial.

Next, we turn to the induction step. By the induction hypothesis, the maximal LWSs for $[i, i + n/2^{t+1}]$ and $[i + n/2^{t+1}, i + n/2^t]$ are stored correctly in the array *next* after step 1. Thus, the values of all $e(x, y)$'s computed from step 2 to step 4 match with their definitions exactly.

In steps 5 and 6, α_h and β_l (i.e., mate(α_h)) are chosen to minimize the sum $e(i, \alpha') + w(\alpha', \beta') + e(\beta', i + 2^t)$ over all α' and β' which are elements in $MS_{i, i+n/2^{t+1}}$ and $MS_{i+n/2^{t+1}, i+n/2^t}$, respectively, so they satisfy the first condition in Corollary 5. Moreover, by the minimization of $mate(\alpha_h)$ in step 5 together with the maximization of α_h in step 6, the second condition of Corollary 5 is also satisfied. Thus, by Corollary 5, we conclude that α_h and β_l are adjacent elements of the maximal LWS for $[i, i + n/2^t]$. The subarray $next[i..i + n/2^t]$, after being updated at step 7, exactly stores the maximal LWS for $[i, i + n/2^t]$. This completes the induction proof.

5. Reduction to Matrix-Searching. In this section we show how to reduce the number of processors used in Algorithm $\mathscr{B}$ to linear. As mentioned in last section, for any instance of procedure *maximal_lws* working on an interval of size m, all the steps except step 5 can be done in $O(\log m)$ time with $m/\log m$ processors. To improve the straightforward implementation of step 5, which requires $O(m^2/\log m)$ processors, we give a reduction of step 5 to a searching problem on a totally monotone matrix [2], [3] of size at most $m \times m$. Aggarwal and Park [3] have already shown how to solve such a matrix-searching problem on a CREW PRAM in $O(\log m \log\log m)$ time with $m/\log\log m$ processors. Our reduction thus implies that step 5 can also be done in $O(\log m \log\log m)$ time with $m/\log\log m$ processors. Using the same analysis in Section 4, we can easily show that the processor requirement of Algorithm $\mathscr{B}$ can be improved to $n/\log\log n$ while the overall time used becomes $O(\log^2 n \log\log n)$.

There is an alternate CREW algorithm for the matrix-searching problem, which takes $O(\log m)$ time but $m \log m$ processors [4]. Thus, Algorithm $\mathscr{B}$ can also be implemented in $O(\log^2 n)$ time with $n \log n$ processors.

Before we give the reduction, let us review the matrix-searching problem. Let T be an $r \times s$ matrix with real entries. Let $\delta(i)$ be the smallest column index j such that $T(i, j)$ equals the minimum[3] value in the ith row of T. The matrix T is monotone if, for $1 \le i_1 \le i_2 \le r$, $\delta(i_1) \le \delta(i_2)$. T is totally monotone if every 2×2 submatrix of T is monotone. The totally monotone matrix-searching problem is that given an $r \times s$ totally monotone matrix, it has to compute all $\delta(i)$'s.

The subproblem in step 5 of Algorithm $\mathscr{B}$ is to compute $mate(\alpha')$ for each α' with $next[\alpha'] \neq \infty$. Recall that $mate(\alpha')$ is the smallest β', with $next[\beta'] \neq \infty$, which minimizes the sum $e(i, \alpha') + w(\alpha', \beta') + e(\beta', j)$. Consider the subarrays $next[i..k-1]$ and $next[k..j]$. Let $i = \alpha_1 < \alpha_2 < \cdots < \alpha_r < k$ and $k = \beta_1 < \beta_2 < \cdots < \beta_s = j$ be the indices of those entries with value $\neq \infty$ in the two subarrays. Define a matrix T of size $r \times s$ as follows: for all $1 \le x \le r$ and $1 \le y \le s$, $T(x, y) = e(\alpha_1, \alpha_{r+1-x}) + w(\alpha_{r+1-x}, \beta_y) + e(\beta_y, \beta_s)$. Note that, for any $i \le \alpha' < k$ with $next[\alpha'] \neq \infty$, α' is equal to α_{r+1-x} for some $1 \le x \le r$, and $mate(\alpha')$ is exactly $\beta_{\delta(x)}$. Thus, computing $mate(\alpha')$'s is equivalent to finding all $\delta(i)$'s of T.

Next we prove that T is totally monotone:

LEMMA 6. *For any i, k, j with $0 \le i < k < j \le n$, let the maximal LWS for $[i, k]$ be $\{\alpha_1, \alpha_2, \ldots, \alpha_r, k\}$ where $i = \alpha_1 < \alpha_2 < \cdots < \alpha_r < k$, and let the maximal LWS for $[k, j]$ be $\{\beta_1, \beta_2, \ldots, \beta_s\}$ where $k = \beta_1 < \beta_2 < \cdots < \beta_s = j$. Define a matrix T of size $r \times s$ as follows: for any $1 \le x \le r$ and $1 \le y \le s$,*

$$T(x, y) = e(\alpha_1, \alpha_{r+1-x}) + w(\alpha_{r+1-x}, \beta_y) + e(\beta_y, \beta_s).$$

Then T is totally monotone.

---

[3] In the literature the matrix-searching problem is concerned with finding the maximum entry on each row of the matrix; yet existing algorithms can also be adpated to find the minimum instead of the maximum.

PROOF. Consider any 2×2 submatrix of T, say, on rows i_1, i_2 and columns j_1, j_2. Assume $i_1 < i_2$ and $j_1 < j_2$. We will prove that if $T(i_1, j_2) < T(i_1, j_1)$, then $T(i_2, j_2) < T(i_2, j_1)$, or, equivalently, the submatrix is monotone.

Obviously, $\alpha_{r+1-i_2} < \alpha_{r+1-i_1} < \beta_{j_1} < \beta_{j_2}$, so we can make use of the convex property of $w(i, j)$'s to obtain the following inequality:

$$w(\alpha_{r+1-i_1}, \beta_{j_1}) + w(\alpha_{r+1-i_2}, \beta_{j_2}) \leq w(\alpha_{r+1-i_2}, \beta_{j_1}) + w(\alpha_{r+1-i_1}, \beta_{j_2}).$$

If $T(i_1, j_2) < T(i_1, j_1)$, then, by the definition of T, we have

$$\begin{aligned} &e(\alpha_1, \alpha_{r+1-i_1}) + w(\alpha_{r+1-i_1}, \beta_{j_2}) + e(\beta_{j_2}, \beta_s) \\ &\quad < e(\alpha_1, \alpha_{r+1-i_1}) + w(\alpha_{r+1-i_1}, \beta_{j_1}) + e(\beta_{j_1}, \beta_s). \end{aligned}$$

Combining the two inequalities and adding $e(\alpha_1, \alpha_{r+1-i_2})$ to both sides, we obtain

$$\begin{aligned} &e(\alpha_1, \alpha_{r+1-i_2}) + w(\alpha_{r+1-i_2}, \beta_{j_2}) + e(\beta_{j_2}, \beta_s) \\ &\quad < e(\alpha_1, \alpha_{r+1-i_2}) + w(\alpha_{r+1-i_2}, \beta_{j_1}) + e(\beta_{j_1}, \beta_s). \end{aligned}$$

According to the definition of T,

$$T(i_2, j_2) = e(\alpha_1, \alpha_{r+1-i_2}) + w(\alpha_{r+1-i_2}, \beta_{j_2}) + e(\beta_{j_2}, \beta_s)$$

and

$$T(i_2, j_1) = e(\alpha_1, \alpha_{r+1-i_2}) + w(\alpha_{r+1-i_2}, \beta_{j_1}) + e(\beta_{j_1}, \beta_s).$$

Hence, $T(i_2, j_2) < T(i_2, j_1)$. □

We are now ready to lay down the details for implementing step 5. Recall that the underlying interval is of size m. By applying Anderson and Miller's optimal list-ranking algorithm [1] to the subarray $next[i..k-1]$, we can find out the number of entries with value $\neq \infty$ (i.e., r), and then build an array $\tilde{\alpha}[1..r]$ storing the indices of these noninfinity entires (i.e., $\alpha_1, \alpha_2, \ldots, \alpha_r$). Similarly, we compute the value of s and build an array $\tilde{\beta}[1..s]$ storing $\beta_1, \beta_2, \ldots, \beta_s$. All the computation mentioned so far can be done in $O(\log m)$ time by $m/\log m$ processors. Now any element of T can be computed easily from the arrays $\tilde{\alpha}$ and $\tilde{\beta}$.

As the size of T can be as large as $O(m^2)$, we do not construct the whole matrix. Instead, we start the matrix-searching algorithm as soon as the arrays $\tilde{\alpha}$ and $\tilde{\beta}$ are available. When the searching algorithm needs to access an entry in T, we evaluate it according to the definition in Lemma 6. Note that it takes constant time for a processor to evaluate any entry of T. Thus, the time and processor required for step 5 are dominated by that of the matrix-searching algorithm.

6. Remarks. In the literature, sequential algorithms for the LWS problem usually find out not only an LWS for the whole interval $[0, n]$, but also an LWS for each subinterval $[0, i]$, where $0 \leq i < n$. It is natural to ask whether the algorithm in

this paper can be extended to find the LWSs for all these subintervals without increasing the time or processor complexity by more than a constant factor. It turns out that this is possible but quite complicated. We need to use a rooted tree instead of a linked list to represent the maximal LWSs within an interval, and need a new version of Theorem 4 to enable us to relate the rooted tree of an interval to that of its subintervals.

As we have mentioned in the introduction, the best-known sequential algorithms for the convex and the concave LWS problem use different approaches. It is not surprising that the algorithm in this paper does not work when the weight function is concave. A new technique may be required to handle the concave case.

Appendix 1

LEMMA 1. *For any i, j with $0 \le i < j \le n$, let S_1 and S_2 be any two LWSs for $[i, j]$, then the union of S_1 and S_2 (denoted by $S_1 \cup S_2$) is also an LWS for $[i, j]$.*

To prove Lemma 1, we need the following weaker proposition.

PROPOSITION 7. *For any i, j with $0 \le i < j \le n$, let S_1 and S_2 be any two LWSs for $[i, j]$ with no common element except the two endpoints, i.e., $S_1 \cap S_2 = \{i, j\}$, then $S_1 \cup S_2$ is also an LWS for $[i, j]$.*

PROOF. We prove the proposition by induction on the size of S_1, which is denoted by $|S_1|$ in the following.

For the basis where $|S_1| = 2$, S_1 must be equal to $\{i, j\}$, so $S_1 \cup S_2$ is equal to S_2 which is an LWS for $[i, j]$.

Next, we turn to the induction step. Suppose the proposition holds for $|S_1| = 2, \ldots, k-1$ for some arbitrary $k > 2$. Consider $|S_1| = k$. Let

$$S_1 = \{\alpha_1, \ldots, \alpha_k\} \quad \text{and} \quad S_2 = \{\beta_1, \ldots, \beta_l\}.$$

For the trivial case where $l = 2$, $S_2 = \{i, j\}$ and $S_1 \cup S_2 = S_1$. Thus, $S_1 \cup S_2$ is an LWS for $[i, j]$. For $l > 2$, we consider the case where $\alpha_2 < \beta_2$ and the case where $\alpha_2 > \beta_2$ separately.

Case 1: $\alpha_2 < \beta_2$. Let $\alpha_r = \max\{\alpha \in S_1 \mid \alpha < \beta_2\}$, then $\alpha_1 = \beta_1 < \alpha_r < \beta_2 < \alpha_{r+1}$. By the convex property of the $w(i, j)$'s, we obtain

$$w(\beta_1, \beta_2) + w(\alpha_r, \alpha_{r+1}) \ge w(\alpha_r, \beta_2) + w(\alpha_1, \alpha_{r+1}).$$

Adding $e(\beta_2, \beta_l) + e(\alpha_1, \alpha_r) + e(\alpha_{r+1}, \alpha_k)$ to both sides of the inequality, we get

$$\begin{aligned}[w(\beta_1, \beta_2) + e(\beta_2, \beta_l)] + [e(\alpha_1, \alpha_r) + w(\alpha_r, \alpha_{r+1}) + e(\alpha_{r+1}, \alpha_k)] \\ \ge [w(\alpha_1, \alpha_{r+1}) + e(\alpha_{r+1}, \alpha_k)] + [e(\alpha_1, \alpha_r) + w(\alpha_r, \beta_2) + e(\beta_2, \beta_l)].\end{aligned}$$

By the definition of S_1 and S_2, the left-hand side of the above inequality is equal to $w(S_2) + w(S_1) = 2 \times e(i, j)$. Furthermore, according to the definition of the

$e(i, j)$'s, $w(\alpha_1, \alpha_{r+1}) + e(\alpha_{r+1}, \alpha_k)$, as well as $e(\alpha_1, \alpha_r) + w(\alpha_r, \beta_2) + e(\beta_2, \beta_l)$, is greater than or equal to $e(i, j)$. We hence conclude that

$$e(i, j) = w(\alpha_1, \alpha_{r+1}) + e(\alpha_{r+1}, \alpha_k) = e(\alpha_1, \alpha_r) + w(\alpha_r, \beta_2) + e(\beta_2, \beta_l),$$

or, equivalently, $\{\alpha_1, \alpha_{r+1}, \alpha_{r+2}, \ldots, \alpha_k\}$ and $\{\alpha_1, \alpha_2, \ldots, \alpha_r, \beta_2, \beta_3, \ldots, \beta_l\}$ are LWSs for $[i, j]$. Since $r \geq 2$, the size of the subsequence $\{\alpha_1, \alpha_{r+1}, \alpha_{r+2}, \ldots, \alpha_k\}$ is at most $k - 1$. Therefore, by the induction hypothesis, the subsequence

$$\{\alpha_1, \alpha_{r+1}, \alpha_{r+2}, \ldots, \alpha_k\} \cup \{\alpha_1, \alpha_2, \ldots, \alpha_r, \beta_2, \beta_3, \ldots, \beta_l\},$$

which is equal to $S_1 \cup S_2$, is an LWS for $[i, j]$.

Case 2: $\alpha_3 > \beta_2$. Let $\beta_s = \max\{\beta \in S_2 | \beta < \alpha_2\}$, then $\alpha_1 = \beta_1 < \beta_s < \alpha_2 < \beta_{s+1}$. By the convex property of the $w(i, j)$'s, we obtain

$$w(\alpha_1, \alpha_2) + w(\beta_s, \beta_{s+1}) \geq w(\beta_s, \alpha_2) + w(\beta_1, \beta_{s+1}). \tag{1}$$

Case 2.1: $\beta_{s+1} = j$. Adding $e(\beta_1, \beta_s) + e(\alpha_2, \alpha_k)$ to both sides of inequality (1), we get

$$\begin{aligned}[w(\alpha_1, \alpha_2) + e(\alpha_2, \alpha_k)] + [e(\beta_1, \beta_s) + w(\beta_s, \beta_{s+1})] \\ \geq [w(\beta_1, \beta_{s+1})] + [e(\beta_1, \beta_s) + w(\beta_s, \alpha_2) + e(\alpha_2, \alpha_k)].\end{aligned}$$

Again, by the definition of S_1 and S_2, the left-hand side of the above inequality is equal to $w(S_1) + w(S_2) = 2 \times e(i, j)$. By the definition of the $e(i, j)$'s, $w(\beta_1, \beta_{s+1})$, as well as $e(\beta_1, \beta_s) + w(\beta_s, \alpha_2) + e(\alpha_2, \alpha_k)$, is greater than or equal to $e(i, j)$. We therefore conclude that

$$e(i, j) = e(\beta_1, \beta_s) + w(\beta_s, \alpha_2) + e(\alpha_2, \alpha_k),$$

and $\{\beta_1, \beta_2, \ldots, \beta_s, \alpha_2, \alpha_3, \ldots, \alpha_k\}$, which is equal to $S_1 \cup S_2$, is an LWS for $[i, j]$.

Case 2.2: $\beta_{s+1} \neq j$. Let $\alpha_r = \max\{\alpha \in S_1 | \alpha < \beta_{s+1}\}$, then $\beta_1 = \alpha_1 < \alpha_r < \beta_{s+1} < \alpha_{r+1}$. By the convex property of the $w(i, j)$'s, we obtain

$$w(\beta_1, \beta_{s+1}) + w(\alpha_r, \alpha_{r+1}) \geq w(\alpha_r, \beta_{s+1}) + w(\alpha_1, \alpha_{r+1}). \tag{2}$$

After combining inequality (1) and inequality (2) together, and adding

$$e(\beta_1, \beta_s) + e(\beta_{s+1}, \beta_l) + e(\alpha_2, \alpha_r) + e(\alpha_{r+1}, \alpha_k)$$

to both sides of the resultant inequality, we have

$$\begin{aligned}&[e(\beta_1, \beta_s) + w(\beta_s, \beta_{s+1}) + e(\beta_{s+1}, \beta_l)] \\ &\quad + [w(\alpha_1, \alpha_2) + e(\alpha_2, \alpha_r) + w(\alpha_r, \alpha_{r+1}) + e(\alpha_{r+1}, \alpha_k)] \\ &\geq [w(\alpha_1, \alpha_{r+1}) + e(\alpha_{r+1}, \alpha_k)] \\ &\quad + [e(\beta_1, \beta_s) + w(\beta_s, \alpha_2) + e(\alpha_2, \alpha_r) + w(\alpha_r, \beta_{s+1}) + e(\beta_{s+1}, \beta_l)].\end{aligned}$$

Using the same argument as before, we see that the left-hand side of the above inequality is equal to $w(S_2) + w(S_1) = 2 \times e(i, j)$, and $w(\alpha_1, \alpha_{r+1}) + e(\alpha_{r+1}, \alpha_k)$, as well as $e(\beta_1, \beta_s) + w(\beta_s, \alpha_2) + e(\alpha_2, \alpha_r) + w(\alpha_r, \beta_{s+1}) + e(\beta_{s+1}, \beta_l)$, is at least $e(i, j)$. Thus,

$$\begin{aligned} e(i, j) &= w(\alpha_1, \alpha_{r+1}) + e(\alpha_{r+1}, \alpha_k) \\ &= e(\beta_1, \beta_s) + w(\beta_s, \alpha_2) + e(\alpha_2, \alpha_r) + w(\alpha_r, \beta_{s+1}) + e(\beta_{s+1}, \beta_l). \end{aligned}$$

In other words, both sequences

$$\{\alpha_1, \alpha_{r+1}, \alpha_{r+2}, \ldots, \alpha_k\} \quad \text{and} \quad \{\beta_1, \beta_2, \ldots, \beta_s, \alpha_2, \alpha_3, \ldots, \alpha_r, \beta_{s+1}, \ldots, \beta_l\}$$

are LWSs for $[i, j]$. Since $r \geq 2$, the size of the subsequence $\{\alpha_1, \alpha_{r+1}, \alpha_{r+2}, \ldots, \alpha_k\}$ is at most $k - 1$. Therefore, by the induction hypothesis,

$$\{\alpha_1, \alpha_{r+1}, \alpha_{r+2}, \ldots, \alpha_k\} \cup \{\beta_1, \ldots, \beta_s, \alpha_2, \ldots, \alpha_r, \beta_{s+1}, \ldots, \beta_l\},$$

which is equal to $S_1 \cup S_2$, is an LWS for $[i, j]$.

This completes our induction step on all possible cases, and hence finishes the proof for Proposition 7. □

PROOF OF LEMMA 1. Let $S_1 = \{\alpha_1, \alpha_2, \ldots, \alpha_r\}$ and $S_2 = \{\beta_1, \beta_2, \ldots, \beta_s\}$. Suppose S_1 and S_2 have $t \geq 2$ common elements denoted by $\{\gamma_1, \gamma_2, \ldots, \gamma_t\}$. For all $1 \leq k \leq t$, let $\gamma_k = \alpha_{a_k} = \beta_{b_k}$, where $1 \leq a_k \leq r$ and $1 \leq b_k \leq s$.

Note that $e(i, j) = e(\gamma_1, \gamma_2) + e(\gamma_2, \gamma_3) + \cdots + e(\gamma_{t-1}, \gamma_t)$, or, equivalently, an LWS for $[i, j]$ can be formed by taking the union of the LWSs for $[\gamma_1, \gamma_2]$, $[\gamma_2, \gamma_3], \ldots,$ and $[\gamma_{t-1}, \gamma_t]$. Consider any $1 \leq k < t$, both subsequences $\{\alpha_{a_k}, \alpha_{a_k+1}, \ldots, \alpha_{a_{k+1}}\}$ and $\{\beta_{b_k}, \beta_{b_k+1}, \ldots, \beta_{b_{k+1}}\}$ are LWSs for $[\gamma_k, \gamma_{k+1}]$ and have no common elements except the endpoints γ_k and γ_{k+1}. By Proposition 7,

$$\{\alpha_{a_k}, \alpha_{a_k+1}, \ldots, \alpha_{a_{k+1}}\} \cup \{\beta_{b_k}, \beta_{b_k+1}, \ldots, \beta_{b_{k+1}}\}$$

is an LWS for $[\gamma_k, \gamma_{k+1}]$. Thus, $S_1 \cup S_2$, which is exactly the union over all $1 \leq k < t$ of the subsequence $\{\alpha_{a_k}, \alpha_{a_k+1}, \ldots, \alpha_{a_{k+1}}\} \cup \{\beta_{b_k}, \beta_{b_k+1}, \ldots, \beta_{b_{k+1}}\}$, is an LWS for $[i, j]$. □

Appendix 2

LEMMA 3. *Given any $0 \leq i < j \leq n$, let x be an element of the maximal LWS for $[i, j]$. Then, for all k, l with $i \leq k < l \leq j$ and $k \leq x \leq l$, x is also an element of the maximal LWS for $[k, l]$.*

To prove Lemma 3, we first prove the following weaker proposition.

PROPOSITION 8. *Given any $0 \leq i < j \leq n$, let x be an element of the maximal LWS for $[i, j]$. Then, for all k, l with $i \leq k < x < l \leq j$, $e(k, x) + e(x, l) \leq w(k, l)$.*

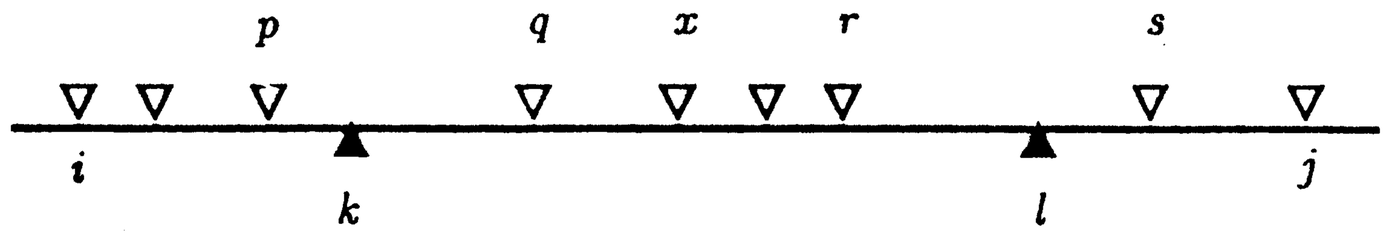

Fig. 5. The relative positions of the variables p, q, r, and s. ∇ is the element of the maximal LWS for $[i, j]$.

PROOF. Recall that $MS_{i,j}$ denotes the maximal LWS for $[i, j]$. Let

$$p = \max\{\alpha \in MS_{i,j} | \alpha \leq k\},\ q = \min\{\alpha \in MS_{i,j} | \alpha > k\},\ r = \max\{\alpha \in MS_{i,j} | \alpha < l\},$$

and

$$s = \min\{\alpha \in MS_{i,j} | \alpha \geq l\}.$$

Figure 5 shows the relative positions of the variables p, q, r, and s. By the convex property of the $w(i, j)$'s, we have the following inequalities:

1. $w(k, q) + w(p, l) \leq w(p, q) + w(k, l)$,
2. $w(r, l) + w(k, s) \leq w(k, l) + w(r, s)$,
3. $w(k, l) + w(p, s) \leq w(p, l) + w(k, s)$.

Furthermore, by the definition of $MS_{i,j}$,

4. $w(p, q) + e(q, x) + e(x, r) + w(r, s) \leq w(p, s)$.

Combining inequalities 1–4, it is easy to see that $w(k, q) + e(q, x) + e(x, r) + w(r, l) \leq w(k, l)$. Since $e(k, x) \leq w(k, q) + e(q, x)$ and $e(x, l) \leq e(x, r) + w(r, l)$, we can finally conclude that $e(k, x) + e(x, l) \leq w(k, l)$. □

PROOF OF LEMMA 3. Suppose the contrary that, for some $i \leq k < l \leq j$, there is an x in $MS_{i,j}$ with $k \leq x \leq l$, but not in $MS_{k,l}$. Obviously, x must be strictly between k and l. Assume $MS_{k,l} = \{\beta_1, \ldots, \beta_m\}$. Let β_r be the largest element in $MS_{k,l}$ smaller than x. Since $\beta_m = l$, r must be smaller than m, and hence β_{r+1} is strictly greater than x. By Proposition 8, $e(\beta_r, x) + e(x, \beta_{r+1}) \leq w(\beta_r, \beta_{r+1})$.

On the other hand, $\{\beta_1, \ldots, \beta_m\}$ is the maximal LWS for $[k, l]$, so the 2-element sequence $\{\beta_r, \beta_{r+1}\}$ is the maximal LWS and hence the only LWS for $[\beta_r, \beta_{r+1}]$. This would mean that $w(\beta_r, \beta_{r+1}) < e(\beta_r, y) + e(y, \beta_{r+1})$ for all $\beta_r < y < \beta_{r+1}$. Thus, contradiction occurs and we prove Lemma 3. □

References

[1] R. J. Anderson and G. L. Miller, Deterministic Parallel List Ranking, *Proceedings of AWOC '88*, Lecture Notes in Computer Science, Vol. 319, Springer-Verlag, Berlin, 1988, pp. 81–90.

[2] A. Aggarwal, M. Klawe, S. Moran, P. Shor, and R. Wilber, Geometric Applications of a Matrix-Searching Algorithm, *Algorithmica*, **2**(2) (1987), 195–208.

[3] A. Aggarwal and J. Park, Notes on Searching in Multidimensional Arrays, *Proceedings of the 29th Annual IEEE Symposium on Foundations of Computer Science*, 1988, pp. 497–512.

[4] M. J. Atallah, S. R. Kosaraju, L. L. Larmore, G. L. Miller, and and S.-H. Teng, Constructing Tress in Parallel, *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, 1989, pp. 421–431.

[5] R. P. Brent, The Parallel Evaluatiοn of General Arithmetic Expressions, *Journal of the Association for Computing Machinery*, **21** (1974), 201–208.

[6] D. Eppstein, Sequence Comparison with Mixed Convex and Concave Costs, *Journal of Algorithms*, **11**(1) (1990), 85–101.

[7] D. Eppstein, Z. Galil, R. Giancarlo, and G. Italiano, Sparse Dynamic Programming, *Proceedings of the First ACM–SIAM Symposium on Discrete Algorithms*, 1990, pp. 513–522.

[8] S. Fortune and J. Wyllie, Parallelism in Random Access Machines, *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, 1978, pp. 114–118.

[9] Z. Galil and R. Giancarlo, Speeding up Dynamic Programming with Applications to Molecular Biology, *Theoretical Computer Science*, **64** (1989), 107–118.

[10] D. S. Hirschberg and L. L. Larmore, The Least Weight Subsequence Problem, *SIAM Journal on Computing*, **16** (1987), 628–638.

[11] R. Karp and V. Ramachandran, A Survey of Parallel Algorithms for Shared Memory Machines, in *Handbook of Theoretical Computer Science*, Vol. A (J. van Leeuwen, ed.), North-Holland, Amsterdam, 1990, pp. 869–941.

[12] M. M. Klawe and D. J. Kleitman, An Almost Linear Algorithm for Generalized Matrix Searching, *SIAM Journal of Discrete Mathematics*, **3**(1), (1990), 81–97.

[13] R. E. Ladner and M. J. Fischer, Parallel Prefix Computation, *Journal of the Association for Computing Machinery*, **27**(4) (1980), 831–838.

[14] G. L. Miller and J. H. Reif, Parallel Tree Contraction and Its Applications, *Proceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science*, 1985, pp. 496–503.

[15] W. Rytter, Notes on Efficient Parallel Computations for Some Dynamic Programming, *Theoretical Computer Science*, **59** (1988), 297–307.

[16] R. Wilber, The Concave Least Weight Subsequence Problem Revisited, *Journal of Algorithms*, **9**(3) (1988), 418–425.

[17] F. F. Yao, Efficient Dynamic Programming Using Quadrangle Inequalities, *Proceedings of the 12th ACM Symposium on Theory of Computing*, 1980, pp. 429–435.