1992

Computing the All-Pairs Longest Chains in the Plane

Mikhail J. Atallah
Purdue University, mja@cs.purdue.edu

Danny Z. Chen

Report Number:

92-046

COMPUTING THE ALL-PAIRS LONGEST CHAINS IN THE PLANE

Mikhail J. Atallah
Danny Z. Chen

CSD-TR-92-046
August 1992
(Revised November 1994)

International Journal of Computational Geometry & Applications
© World Scientific Publishing Company

Computing the All-Pairs Longest Chains in the Plane

Mikhail J. Atallah
Department of Computer Sciences, Purdue University,
West Lafayette, Indiana 47907, USA

and

Danny Z. Chen
Department of Computer Science and Engineering, University of Notre Dame,
Notre Dame, Indiana 46556, USA

Received (received date)
Revised (revised date)
Communicated by Editor's name

ABSTRACT

Many problems on sequences and on special kinds of graphs involve the computation of longest chains passing points in the plane. Given a set S of n points in the plane, we consider the problem of computing the matrix of longest chain lengths between all pairs of points in S, and the matrix of "parent" pointers that describes the n longest chain trees. We present a simple sequential algorithm for computing these matrices. Our algorithm runs in $O(n^2)$ time, and hence is optimal. We also present a rather involved parallel algorithm that computes these matrices in $O((\log n)^2)$ time using $O(n^2/\log n)$ processors in the CREW PRAM model. These matrices enable us to report, in $O(1)$ time, the length of a longest chain between any two points in S by using one processor, and the actual chain by using k processors, where k is the number of points of S on that chain. The space complexity of the algorithms is $O(n^2)$.

Keywords: Computational geometry, CREW PRAM, increasing chains, longest paths, Monge matrices, parallel algorithms, sequential algorithms

1. Introduction

Problems that involve longest increasing subsequences of a given sequence of numbers have attracted a lot of attention in the past. Probably the most studied version is that of the *longest increasing subsequence* (LIS), for which many $O(n \log n)$ time sequential algorithms are known (examples are the results,[1,2,3] and many others). There is also a well-known connection between increasing subsequences and problems on certain specialized classes of graphs such as permutation graphs, circle and circular-arc graphs, and interval graphs (for example, see references[4,5,6,7,8,9,10]). This is not surprising, since all these problems involve objects that can be defined by using two parameters.

This paper considers the all-pairs version of the problem, whose formulation we state precisely next. Because our solution techniques are drawn from computational geometry, we have chosen to formulate the problem as one on points in the plane: In terms of sequences, the y-coordinate of a planar point corresponds to the value of an entry in the sequence, and the position at which that entry occurs in the sequence is determined by the x-coordinate; in terms of interval graphs, the x-coordinate corresponds to the beginning position of an interval and the y-coordinate corresponds to its ending position; and so on. Thus our results have applications to all of these problems. For example, in a sequence, we are in effect computing a description of all longest increasing subsequences between all the pairs of positions in the sequence; in a connected interval graph, we are computing a description of all longest paths that are proper and extend from left to right (recall that a set of intervals is proper if every interval in the set contains no other interval except itself); also in interval graphs, by using our algorithms on anti-chains rather than chains of points, we can compute a description of all tallest towers (where a tower is a sequence of intervals such that if interval I follows interval J, then J contains I).

A point p in the plane is said to ***dominate*** another point q iff $x(p) \geq x(q)$ and $y(p) \geq y(q)$, where $x(p')$ and $y(p')$ respectively denote the x- and y-coordinates of a point p'. Let S be a set of n points in the plane, and let $\sigma = (p_1, p_2, \ldots, p_k)$ be a sequence of points such that each p_i is in S. The sequence σ is ***increasing*** iff p_i dominates p_{i-1} for all i, $1 < i \leq k$; such a sequence is called a ***chain***, and we say that it ***begins*** at p_1, that it ***ends*** at p_k, and that its ***length*** is k (if the points are weighted then the length of σ is the sum of the weights of its points). The chain σ is ***longest*** if no other chain starting at p_1 and ending at p_k has greater length than σ.

The problem we consider is that of computing an $n \times n$ matrix D of the lengths of longest chains between all the pairs of points in S; that is, $D(p,q)$ is equal to the length of a longest p-to-q chain. By convention, for $p \neq q$, if q does not dominate p then $D(p,q) = -\infty$. We also compute an $n \times n$ matrix P (shorthand for "parent") such that $P(p,q)$ is the successor of p in a p-to-q longest chain.

Sequentially, we give a simple $O(n^2)$ time algorithm for the unweighted case of the problem (i.e., the weight of every point in S is 1). Clearly, knowing P allows one processor to trace a longest p-to-q chain in time proportional to its length.

In parallel, we solve the weighted version of the problem in $O((\log n)^2)$ time using $O(n^2/\log n)$ processors in the CREW PRAM model. We also show that a longest p-to-q chain can be obtained in constant time by using k CREW PRAM processors, where k is the number of points of S on that chain. The parallel algorithm bears very little resemblance to the sequential one, which seems hard to "parallelize". It also solves a more general (weighted) version of the problem. Recall that the CREW PRAM is the synchronous shared-memory model in which multiple processors can simultaneously read from the same memory location but at most one processor is allowed to write to a memory location at each time unit.

An $O(n^2 \log n)$ time sequential algorithm for this problem is quite trivial to

Figure 1: (a) Monotonicity holds for some chains, and (b) it fails to hold for others.

obtain, and to the best of our knowledge this was the best previously known bound for this all-pairs version of the problem. There is a published $O(n^2)$ time sequential algorithm[11] for a special case of this problem: That for chains which start in S and end on a set of points that lie on a vertical line V, where V is to the right of S. In parallel, complexity bounds similar to ours were only known for the special case of the layers of maxima problem, which can be viewed as the version of our problem where the chains of interest begin in S but must end at the point $(+\infty, +\infty)$.[12] It is actually quite easy to use the methods[12,13] to solve the version of the problem where the chains of interest begin in S but must end on a set of points on a vertical line V that is to the right of S.

We now briefly discuss how our approach differs from the one for the special version of the problem mentioned above, in which all the chains start in S and end on a set of points on a vertical line V that is to the right of S. That special version of the problem is substantially easier, both sequentially and in parallel, because for a fixed point $p \in S$, the collection of longest chains that begin at p and end on V have the following monotonicity property: Two such longest chains that end at (respectively) q' and q'', $y(q') < y(q'')$, can always be chosen such that at no point is the chain to q' higher (geometrically) than the chain to q'' (intuitively, if it is higher then there is a crossing between the two chains and we can "uncross" them). Figure 1(a) illustrates this. Such a monotonicity property is lacking in the general version of the problem considered here: If q' and q'' do not lie on the same vertical line (see Figure 1(b)), then monotonicity need not hold, in the sense that either one of the two p-to-q'' chains shown could be a longest chain to q'', so that such a chain to q'' might go either "above" or "below" a longest p-to-q' chain.

We are unable to obtain an $O(n^2)$ time sequential solution to the weighted version of this problem, and we leave this as an open problem. Our parallel bounds, on the other hand, hold for the weighted version of the problem.

Figure 2: The points of $MD(p)$ are circled.

The rest of this paper is organized as follows. Section 2 deals with the sequential algorithm, which is fairly simple. Section 3 discusses the parallel algorithm. We have chosen to give the basic terminology and definitions separately for each of the parallel and sequential algorithms, since they have little in common (this way the reader interested in one of the two will not be forced to read material unrelated to her interest). Section 4 concludes by mentioning a related new result and posing some open problems.

2. The Sequential Algorithm

This section presents the $O(n^2)$ time sequential algorithm.

2.1. Preliminaries

The input consists of the set S of n points in the plane. For a point $p \in S$, we use $DOM(p)$ to denote the subset of points in $S - \{p\}$ that are dominated by p. A point p of S is a ***maximum*** (or ***maximal element***) of S iff no other point of S dominates p. We use $MAX(S)$ to denote the set of maxima of S, listed by increasing x-coordinates (and hence by decreasing y-coordinates). For a point $p \in S$, we abbreviate as $MD(p)$ the set of maxima of $DOM(p)$ (i.e., $MAX(DOM(p))$). Figure 2 illustrates the definition of $MD(p)$.

For a point $p \in S$, imagine partitioning $DOM(p) \cup \{p\}$ into k subsets, where $k = \max\{D(q,p) \mid q \in DOM(p) \cup \{p\}\}$, such that the points in each subset all have their longest chains to p of the same length. The subset of $DOM(p) \cup \{p\}$ whose points have a distance j to p is called the j-th ***domination layer*** of p, denoted by $Layer_j(p)$. For example, $Layer_1(p) = \{p\}$, $Layer_2(p) = MD(p)$, and so on. In general, for each j, $Layer_j(p) = MAX((DOM(p) \cup \{p\}) - \cup_{i=1}^{j-1} Layer_i(p))$. We assume that each layer of p is sorted by increasing x-coordinates (hence by decreasing y-coordinates).

It should be clear that, if we were able to compute the domination layers of each $p \in S$, then we would effectively have computed the desired D matrix. Our sequential algorithm will therefore mainly concern itself with the computation of these domination layers and of the P matrix. (The parallel algorithm deals with the weighted version and will use a different approach — in fact most of the definitions given above will not be used in the parallel algorithm.)

2.2. The Algorithm

Below is a high-level description of the sequential algorithm. We are assuming that no two points in S have the same x- (resp., y-) coordinate, i.e., that if $p, q \in S$ and $p \neq q$ then $x(p) \neq x(q)$ and $y(p) \neq y(q)$ (the algorithm can easily be modified for the general case). By convention, walking ***forward*** (resp., ***backward***) along an $MD(p)$ means moving along it by increasing (resp., decreasing) x-coordinates.

Step 1. We first compute $MD(p)$ for every point $p \in S$. These $MD(p)$'s can all be easily computed in $O(n^2)$ time as follows. We sort the points in S by their x-coordinates, and then for each $p \in S$ we do the following. From the sorted list, we obtain $DOM(p)$, in $O(n)$ time. Then we obtain the maximal elements of $DOM(p)$, also in $O(n)$ time (this is possible since $DOM(p)$ is available sorted). These maximal elements of $DOM(p)$ are, by definition, $MD(p)$.

Step 2. We compute, for each pair of points p, $q \in S$, the position of $y(p)$ in the list $Y(MD(q))$, which is the list obtained from $MD(q)$ by replacing every point by its y-coordinate. This is easy to do in $O(n)$ time for a particular $Y(MD(q))$ and all $p \in S$, by merging $Y(MD(q))$ with the sorted list of the y-coordinates of the points in S. Doing this once for each $q \in S$ takes a total of $O(n^2)$ time.

Step 3. For each point $p \in S$, we obtain the domination layers of p and the column that corresponds to p in matrix P. We do this in $O(n)$ time for each p, as follows. Clearly, we already have $Layer_1(p)$ $(= \{p\})$ and $Layer_2(p)$ $(= MD(p))$. We obtain $Layer_{j+1}(p)$ from $Layer_j(p)$ in $O(|Layer_j(p)| + |Layer_{j+1}(p)|)$ time, as follows. Let $Layer_j(p) = (a_1, a_2, \ldots, a_k)$, where $x(a_1) < x(a_2) < \cdots < x(a_k)$. We shall walk along the $Layer_j(p)$ list, creating the $Layer_{j+1}(p)$ list as we go along, in left to right order. When we reach a_i while scanning $Layer_j(p)$, we compute the portion of $Layer_{j+1}(p)$ that is in $MD(a_i)$ but not in $DOM(a_{i+1})$; we call this portion $\mathit{Interval}_{j+1}(p, a_i)$ (it forms a contiguous interval of $MD(a_i)$). Figure 3 illustrates the definition of $\mathit{Interval}_{j+1}(p, a_i)$. Note how, in Figure 3, point w is in $Layer_{j+1}(p) \cap MD(a_2)$ but not in $\mathit{Interval}_{j+1}(p, a_2)$. We shall compute $\mathit{Interval}_{j+1}(p, a_1)$, $\mathit{Interval}_{j+1}(p, a_2)$, $\ldots$, $\mathit{Interval}_{j+1}(p, a_k)$, in that order. While doing this, we maintain a variable called ***cutoff*** whose significance is that, when we finish processing a_i, ***cutoff*** contains the rightmost point in $\cup_{\ell=1}^{i} \mathit{Interval}_{j+1}(p, a_\ell)$; intuitively, ***cutoff*** is the "dominant" point among those in $\cup_{\ell=1}^{i} \mathit{Interval}_{j+1}(p, a_\ell)$ as far as the (yet to be computed) lists $\mathit{Interval}_{j+1}(p, a_{i+1}), \ldots, \mathit{Interval}_{j+1}(p, a_k)$ are concerned. In Figure 3, after $\mathit{Interval}_{j+1}(p, a_1)$ is computed, ***cutoff*** is point t, and after $\mathit{Interval}_{j+1}(p, a_2)$ is computed ***cutoff*** is point q'.

To determine $\mathit{Interval}_{j+1}(p, a_1)$, we simply start at the beginning of $MD(a_1)$ and walk forward along $MD(a_1)$ until we first reach a point $q \in MD(a_1)$ for which $y(q) <$

Figure 3: The points of $\mathit{Interval}_{j+1}(p, a_2)$ are shown circled.

$y(a_2)$ (we do not count q as being part of our "walk" along $MD(a_1)$). The (possibly empty) portion of $MD(a_1)$ so traced is obviously equal to $\mathit{Interval}_{j+1}(p, a_1)$. If $\mathit{Interval}_{j+1}(p, a_1)$ is not empty, then we set *cutoff* equal to the predecessor of q in $MD(a_1)$; otherwise *cutoff* remains undefined. For the example shown in Figure 3, $q = u$ and $\mathit{cutoff} = t$. We then proceed to processing a_2.

If *cutoff* is undefined (i.e., if $\mathit{Interval}_{j+1}(p, a_1)$ turned out to be empty), then we process a_2 exactly as explained above for a_1. Otherwise we process a_2 as follows. Recall that we already know, from Step 2, the outcome of a hypothetical binary search for $y(a_3)$ in $Y(MD(a_2))$: Let q' be the predecessor of $y(a_3)$ in $Y(MD(a_2))$, that is, the lowest point of $MD(a_2)$ whose y-coordinate is larger than $y(a_3)$. If no such point q' exists on $MD(a_2)$ then surely $\mathit{Interval}_{j+1}(p, a_2)$ is empty and we move on to processing a_3 (leaving *cutoff* unchanged). So suppose that q' exists. If $x(q') < x(\mathit{cutoff})$ then $\mathit{Interval}_{j+1}(p, a_2)$ is empty and we move on to processing a_3 (leaving *cutoff* unchanged). If $x(q') > x(\mathit{cutoff})$ then we start at q' and walk backward along $MD(a_2)$ until we reach a point whose x-coordinate is less than $x(\mathit{cutoff})$; the portion of $MD(a_2)$ so traced is $\mathit{Interval}_{j+1}(p, a_2)$. In Figure 3, the portion so traced consists of points q', v, and u, in that order (point s is not traced because $x(s) < x(t)$). In that case, we also update *cutoff* by setting it equal to q' before we proceed to processing a_3.

We process a_3, a_4, $\ldots$, a_k, in that order, exactly as explained above except that, when a_{i+1} is processed, a_i, a_{i+1}, and a_{i+2} play the role of a_1, a_2, and a_3, respectively.

Once we have obtained $\mathit{Layer}_{j+1}(p)$ from $\mathit{Layer}_j(p)$, we must compute $P(w, p)$ for every $w \in \mathit{Layer}_{j+1}(p)$ (clearly, such a $P(w, p)$ is in $\mathit{Layer}_j(p)$). This is easily done for all $w \in \mathit{Layer}_{j+1}(p)$ in $O(|\mathit{Layer}_j(p)| + |\mathit{Layer}_{j+1}(p)|)$ time, by merging the two lists $\mathit{Layer}_{j+1}(p)$ and $\mathit{Layer}_j(p)$.

This completes the description of the sequential algorithm. We now turn our attention to the parallel algorithm.

3. The Parallel Algorithm

This section presents the $O((\log n)^2)$ time, $O(n^2/\log n)$ processor algorithm for the weighted version of the problem.

3.1. Preliminaries

Let S be a set of n *weighted* points in the plane (i.e., each point $p \in S$ is associated with a nonnegative weight). The length of an increasing chain C through some points in S is the total sum of the weights of the points in $C \cap S$.

In what follows, we shall focus on showing that the claimed time complexity for computing matrices D and P in the *weighted* case of the problem can be achieved with $O(n^2 \log n)$ work. (Recall that the ***work*** complexity of a parallel algorithm is equal to the total number of operations performed by that algorithm.) This will imply the $O(n^2/\log n)$ processor bound, by Brent's theorem[14]:

Theorem 1 (Brent) *Any synchronous parallel algorithm taking time* T *that consists of a total of* W *operations can be simulated by* P *processors in time* $O((\mathsf{W}/\mathsf{P})+\mathsf{T})$.

Remark: There are actually two qualifications to the above Brent's theorem before one can apply it to a PRAM: (i) At the beginning of the i-th parallel step, we must be able to compute the amount of work W_i performed by that step, in time $O(\mathsf{W}_i/\mathsf{P})$ and with P processors, and (ii) we must know how to assign each processor to its task. Both qualifications (i) and (ii) to the theorem will be easily satisfied in our algorithms, therefore the main difficulty will be how to achieve W operations in time T.

Here as in the shortest path results,[15,16] an important method we use involves multiplying special kinds of matrices. Although the situation depicted in Figure 1(b) implies that the monotonicity structure that gives rise to such matrices is not always available, the fact that we can deal with the situation in Figure 1(a) will be useful. (This will all be made precise later; for now we are only giving an overview.) All matrix multiplications in our algorithm are in the $(\max, +)$ closed semi-ring, i.e., $(M' * M'')(i,j) = \max_k \{M'(i,k) + M''(k,j)\}$.

Lemma 1 *Let V, V', and V'' be three vertical lines with $x(V) < x(V') < x(V'')$. Let S' (resp., S'') be the set of points in S whose x-coordinates are $\geq x(V)$ (resp., $x(V')$) and $\leq x(V')$ (resp., $x(V'')$). Let V (resp., V', V'') contain a set X (resp., Y, Z) of points such that the weights of the points in X (resp., Y, Z) are all zero, that X (resp., Z) contains the horizontal projections of the points of S' (resp., S'') onto V (resp., V''), and that the y-coordinates of Y are the union of those of X with those of Z (see Figure 4). Let $\Omega = S' \cup S'' \cup X \cup Y \cup Z$. Then for every increasing chain C through Ω from a point $p \in X$ to a point $q \in Z$, $y(p) \leq y(q)$, there is a p-to-q increasing chain C' through Ω such that C' is at least as long as C and that C'' goes through some point $w \in Y$.*

Proof. Obvious. □

Figure 4: Illustrating Lemma 1.

Lemma 2 *Let V (resp., V', V'') be a vertical line that contains a set X (resp., Y, Z) of points that are ordered by increasing y-coordinates along V (resp., V', V'') and whose weights are all zero. Let $\Omega = S \cup X \cup Y \cup Z$. For subsets A, B of Ω, let $M[A, B]$ be the length matrix of the longest chains through Ω from the points in A to the points in B. Assume that $x(V) < x(V') < x(V'')$, that $|X| = |Z|$, and that the y-coordinates of Y are the union of those of X with those of Z (hence $|Y| = 2|X|$). Then given the matrices $M[X, Y]$ and $M[Y, Z]$, the matrix $M[X, Z]$ can be computed in $O(\log |X|)$ time and $O(|X|^2)$ work in the CREW PRAM model.*

Proof. Lemma 1 implies that $M[X, Z] = M[X, Y] * M[Y, Z]$. For every point $u \in X$ and $w \in Z$, let $\theta(u, w)$ be the (geometrically) lowest point v of Y for which $M[X, Z](u, w) = M[X, Y](u, v) + M[Y, Z](v, w)$. If $y(u) > y(w)$ (i.e., $M[X, Z](u, w) = -\infty$), then we define $\theta(u, w)$ to be the point v' of Y such that $y(v') = y(u)$ (such a v' exists in Y by the definition of Y). It is not hard to see that the following property holds: For any points u_1, u_2 in X and w_1, w_2 in Z, if u_1 is below u_2 and w_1 is below w_2, then we have the following properties:

1. $\theta(u_1, w_1)$ is below or equal to $\theta(u_1, w_2)$, and

2. $\theta(u_1, w_1)$ is below or equal to $\theta(u_2, w_1)$.

It was shown directly[13] (and, indirectly[12]) that whenever the above relationships hold, then the θ matrix can be computed in parallel within the claimed complexity bounds: The relationships presented above are identical to property (4) on page 975 of the paper,[13] where, in the last sentence of the paragraph preceding property (4), it was stated that this property is the only structure needed by the algorithm[13] to compute matrix θ (we refer the reader to the algorithm in Section 6 of the paper,[13] in which it is transparent that the properties described above is indeed all that is needed). By using this algorithm in our case, matrix θ can be computed in

$O(\log|X|)$ time and $O(|X|^2)$ work on the CREW PRAM. Once we have matrix θ, it is straightforward to obtain from it the desired $M[X,Z]$ matrix within the same complexity bounds. □

3.2. The Algorithm for Chain Lengths

The algorithm given in this subsection concerns itself with the computation of the chain lengths only, not of the P matrix that describes the n trees of longest chains. Including the computation of P here would have cluttered the exposition. The next subsection will deal with the computation of P. In addition, it is not immediately clear that the availability of P makes possible the reporting of a k-point chain in $O(k)$ work and constant time. This too is postponed until the next subsection.

Let $S = \{p_1, p_2, \ldots, p_n\}$, where $x(p_1) < x(p_2) < \cdots < x(p_n)$. Let $V_0, V_1, \ldots, V_n$ be vertical lines such that $x(V_0) < x(p_1)$, $x(p_n) < x(V_n)$, and $x(p_i) < x(V_i) < x(p_{i+1})$ for all $i \in \{1, 2, \ldots, n-1\}$.

Let T be a complete n-leaf binary tree. For each leaf v of T, if v is the i-th leftmost leaf in T, then associate with v the region I_v of the plane that is between V_{i-1} and V_i. For each internal node v of T, associate with v the region I_v consisting of the union of the regions of its children. That is, if v has children u and w, then $I_v = I_u \cup I_w$.

Let v be any node of T. Suppose that the left (resp., right) boundary of I_v is V_i (resp., V_j), and let $S_v = S \cap I_v$, that is, S_v is the subset of the input points that lie in I_v. Observe that if v is at a height of h in T, then $j - i = 2^h = |S_v|$ (the height of v is the height of the subtree in T rooted at v, with leaves being at a height of zero). Let L_v (resp., R_v) be the set of points on V_i (resp., V_j) that are the horizontal projections of S_v on V_i (resp., V_j). The points of L_v and R_v are, of course, disjoint from the input set S, and we assign to each of them a weight of zero. Observe that

$$\sum_{v \in T} |L_v| = \sum_{v \in T} |R_v| = O(n \log n),$$

because for each level of T, a point $p_i \in S$ appears in exactly one S_v of that level, and hence creates at most two extra points, one in L_v and one in R_v (recall that a level of T is the set of nodes in T that have the same distance to the root of T, so that the root is at level zero, the two children of the root are at level 1, and so on).

The algorithm consists of two phases: Phase 1 is relatively straightforward, while Phase 2 is the key that made our solution possible.

3.2.1. Phase 1

This phase consists of computing, by starting at the leaves and going upward in tree T, one level at a time, the $M[L_v, R_v]$ matrices for nodes v of T, which contain the lengths of all the L_v-to-R_v longest chains (chains that begin on L_v and end on R_v, of course possibly going through points in S_v along the way). This information is trivially available if v is a leaf. So suppose that we have already computed this information for level $\ell + 1$, and we want to compute it for level ℓ.

We claim that it suffices to show that the $M[L_v, R_v]$ matrix can be computed in $O(|S_v|^2)$ work and $O(\log n)$ time for each node v at level ℓ. This claim would imply an $O((\log n)^2)$ time, $O(n^2)$ work procedure for Phase 1, as follows. That the time bound follows from the claim is obvious (we would spend logarithmic time per level, and there is a logarithmic number of levels). The work bound would follow from the fact that there are 2^ℓ nodes v at level ℓ, each having $|S_v| = n/2^\ell$, and hence the total work at level ℓ would be

$$O(2^\ell(n/2^\ell)^2) = O(n^2/2^\ell).$$

Summing over all levels ℓ gives $O(n^2)$ total work. We next prove the claim by showing that the $M[L_v, R_v]$ matrix can indeed be computed in $O(|S_v|^2)$ work and logarithmic time.

Let u (resp., w) be the left (resp., right) child of v in T. Let Y denote $R_u \cup L_w$, that is, Y consists of the horizontal projections of the points of S_v on the vertical line V_j that separates the region I_u from the region I_w. Our tool for computing $M[L_v, R_v]$ is Lemma 2. In order for Lemma 2 to be applicable, we must have matrices $M[L_v, Y]$ and $M[Y, R_v]$. What is already available to us (from the computation at the children u and w of v) is matrices $M[L_u, R_u]$ and $M[L_w, R_w]$. We only show how to obtain $M[L_v, Y]$ from $M[L_u, R_u]$ in the claimed parallel bounds ($M[Y, R_v]$ can be obtained from $M[L_w, R_w]$ in a similar way).

Note that L_v (resp., Y) consists of, in addition to the points in L_u (resp., R_u), the points on the left (resp., right) boundary of I_u that are the horizontal projections of the points in S_w. To obtain $M[L_v, Y]$, what we need to compute is $M[L_v, Y](p, q)$ for every pair of points p and q, where $p \in L_v$, $q \in Y$, and either $p \in L_v - L_u$ or $q \in Y - R_u$. If $y(p) > y(q)$, then computing $M[L_v, Y](p, q)$ is trivial (since it is $-\infty$ in this case). Hence we assume, without loss of generality (WLOG), that $y(p) \leq y(q)$. We also assume that no two points in S_v have the same y-coordinate (the general situation can be easily taken care of). We perform the following computation:

(i) Perform a parallel prefix[17,18] along L_v (resp., Y), in decreasing (resp., increasing) y-coordinates, to compute, for every point $z \in L_v - L_u$ (resp., $Y - R_u$), the lowest (resp., highest) point $l(z)$ (resp., $h(z)$) such that $l(z) \in L_u$ (resp., $h(z) \in R_u$) and that $y(l(z)) \geq y(z)$ (resp., $y(h(z)) \leq y(z)$).

(ii) For every pair of points p and q such that $p \in L_v$, $q \in Y$, and either $p \in L_v - L_u$ or $q \in Y - R_u$, do the following.

If $y(p) \leq y(l(p)) \leq y(q)$, then let $M[L_v, Y](p, q) = M[L_u, R_u](l(p), h(q))$. (Observe that in this case, $y(l(p)) \leq y(h(q)) \leq y(q)$, since each point of S_u has a projection point in each of L_u and R_u.)

Otherwise, let $M[L_v, Y](p, q) = 0$. (In this case, there obviously exists no point $z \in S_u$ such that $y(p) \leq y(z) \leq y(q)$.)

It is easy to see that the computation described above can be easily done in $O(|S_v|^2)$ work and $O(\log n)$ time. Hence matrix $M[L_v, Y]$ can be computed in the claimed parallel bounds.

After both $M[L_v, Y]$ and $M[Y, R_v]$ are available, computing $M[L_v, R_v]$ is easy. Now, simply observe that the conditions of Lemma 2 are satisfied, with L_v playing the role of X and R_v playing the role of Z. That is, we can obtain $M[L_v, R_v]$ from $M[L_v, Y]$ and $M[Y, R_v]$ in $O(|S_v|^2)$ work and logarithmic time. This completes the proof.

Remark: The astute reader may have observed that the above procedure can be modified so as to compute the L_v-to-S_v and S_v-to-R_v chain lengths information. This would involve only a logarithmic factor of additional work, and would exploit the kind of monotonicity property depicted in Figure 1(a) by using the lower-dimensional parallel matrix searching algorithm.[19] However, this would still leave us far from having solved our problem: We would still need something like Phase 2 below, since we cannot afford to multiply "non-square" length matrices — as of now, it is not known how to optimally $(\max, +)$-multiply two non-square length matrices (for example, the best parallel algorithm for multiplying a $1 \times k$ length matrix with a $k \times k$ matrix in logarithmic time takes $O(k \log k)$ work[19]). Observe how Phase 2 below will satisfy the size requirements of Lemma 2, as expressed in the requirement that $|X| = |Z| = |Y|/2$.

3.2.2. Phase 2

Whereas Phase 1 involved a bottom-up computation in tree T, Phase 2 will involve a top-down computation, by starting at the root of T and proceeding one level at a time until we reach the leaves. The purpose of the computation at a typical level ℓ is more ambitious than in Phase 1: We now seek, for every pair of nodes u, w at level ℓ such that u is to the left of w, the computation of the $M[R_u, L_w]$ matrix of chain lengths (u is to the left of w iff it is in the subtree rooted at the left child of the lowest common ancestor of u and w). The key idea is to get help from the parents of u and w, which we call u' and (respectively) w'. If $u' = w'$, then the desired information is trivially available; so suppose that $u' \neq w'$. We distinguish four cases, which are illustrated in Figure 5.

Case 1: u is the right child of u', and w is the left child of w' (see Figure 5(a)). Simply obtain $M[R_u, L_w]$ from $M[R_{u'}, L_{w'}]$ which was computed earlier in Phase 2.

Case 2: u is the right child of u', and w is the right child of w'. Let β be the left child of w' (see Figure 5(b)). From the $M[R_{u'}, L_{w'}]$ matrix which was computed earlier in Phase 2, obtain the $M[R_u, L_\beta]$ matrix. From the $M[L_\beta, R_\beta]$ matrix which was computed in Phase 1, we obtain the $M[L_\beta, L_w]$ matrix (this computation is similar to steps **(i)** and **(ii)** of Phase 1). We use Lemma 2 to obtain $M[R_u, L_w]$ from $M[R_u, L_\beta]$ and $M[L_\beta, L_w]$.

Case 3: u is the left child of u', and w is the left child of w'. Let α be the right child of u' (see Figure 5(c)). From the $M[L_\alpha, R_\alpha]$ matrix which was computed in Phase 1, we obtain the $M[R_u, R_{u'}]$ matrix. Observe that $M[R_{u'}, L_{w'}]$ was already obtained earlier in Phase 2: Get from it the $M[R_{u'}, L_w]$ matrix. We use Lemma 2 to obtain $M[R_u, L_w]$ from matrices $M[R_u, R_{u'}]$ and $M[R_{u'}, L_w]$.

Figure 5: Illustrating the four cases of Phase 2.

Case 4: u is the left child of u', and w is the right child of w'. Let α be the right child of u' and β be the left child of w' (see Figure 5(d)). Since Phase 1 computed $M[L_\alpha, R_\alpha]$, we can obtain from it $M[R_u, R_\alpha]$, then $M[R_u, R_{u'}]$. Similarly, we obtain $M[L_{w'}, L_w]$ from $M[L_\beta, R_\beta]$ which was computed in Phase 1. Now, $M[R_{u'}, L_{w'}]$ is already available because Phase 2 is already done with processing the pair u', w' (recall that Phase 2 processes the levels from the root down). We use Lemma 2 to obtain the matrix $M[R_u, L_{w'}]$ from $M[R_u, R_{u'}]$ and $M[R_{u'}, L_{w'}]$, with R_u playing the role of X, $R_{u'}$ playing the role of Y, and $L_{w'}$ playing the role of Z. Finally, we use Lemma 2 again, this time to obtain the desired matrix $M[R_u, L_w]$ from $M[R_u, L_{w'}]$ and $M[L_{w'}, L_w]$.

The time taken by Phase 2 is clearly $O((\log n)^2)$, since the procedure takes logarithmic time at leach level of T. The work done for a particular pair u, w at level ℓ is $O((n/2^\ell)^2)$, and, since there are $(2^\ell)^2$ such pairs at level ℓ, the total work done at that level is $O(n^2)$. Summing over all the levels gives $O(n^2 \log n)$ work for Phase 2. Hence it is Phase 2 that is the bottleneck in the work complexity. The space used by Phase 2 is still $O(n^2)$ rather than $O(n^2 \log n)$, however, since we do not need to store the matrices for all the levels as Phase 2 proceeds: When we are done with level ℓ, we can discard the matrices for level $\ell - 1$ since level $\ell + 1$ will only need the information produced at level ℓ (recall that in Phase 2, the nodes of T request help only from their parents, not from their grandparents or from nodes higher up in T).

3.3. Computing the Actual Chains

In this subsection, we discuss how to obtain matrix P which contains the n trees of longest chains, and how to preprocess the longest chain trees, so that each tree can support a longest chain query between any point in S and the point of S at the root of that tree.

First we sketch how the algorithm in the previous subsection can be modified so as to compute the P matrix as well. For each $M[R_u, L_w]$ matrix computed by that algorithm, we compute a companion $P[R_u, L_w]$ matrix whose significance is that, for $p \in R_u$ and $q \in L_w$, $P[R_u, L_w](p,q)$ is the first point of S that lies on a longest p-to-q chain (it is undefined if no p-to-q chain contains any point of S). Note that only the points of S can be "parents". It is quite easy to modify the computation of an $M[X,Z]$ length matrix so that it also produces $P[X,Z]$: If $M[X,Z]$ is obtained by using Lemma 2, then $P[X,Z]$ can be obtained from $P[X,Y]$ or $P[Y,Z]$ as a "byproduct" of this computation. For example, if q dominates p and if $M[X,Z](p,q) = M[X,Y](p,t) + M[Y,Z](t,q)$, then we distinguish two cases for obtaining $P[X,Z](p,q)$: If $P[X,Y](p,t)$ is undefined, then $P[X,Z](p,q) = P[Y,Z](t,q)$ (which could also be undefined); otherwise $P[X,Z](p,q) = P[X,Y](p,t)$. When the modified algorithm finishes computing $P[R_u, L_w]$ for all the leaves u, w (at the end of Phase 2), it is easy to obtain matrix P: If $S_u = \{p_i\}$, $R_u = \{p'_i\}$, $S_w = \{p_j\}$, $L_w = \{p'_j\}$, then we set $P(p_i, p_j)$ equal to $P[R_u, L_w](p'_i, p'_j)$ if the latter is defined; otherwise, we set $P(p_i, p_j)$ equal to p_j if p_j dominates p_i, and set $P(p_i, p_j)$ to be undefined if p_j does not dominate p_i. From now on, we assume that matrix P is available. Note that this matrix is a description of n trees of longest chains, each rooted at a point of S.

We preprocess each longest chain tree so that the following type of queries can be quickly answered: Given a node p in the tree and a positive integer i, find the i-th node on the path from p to the root of the tree. Such queries are called *level-ancestor queries* by Berkman and Vishkin,[20] who gave efficient parallel algorithms for preprocessing rooted trees so that the level-ancestor queries can be answered quickly. The work of Berkman and Vishkin[20,21] shows (implicitly) that a level-ancestor query can be handled sequentially in constant time, after a logarithmic time and linear work preprocessing in the CREW PRAM model. The preprocessing of the longest chain trees is done by simply applying the result of Berkman and Vishkin to each of the n trees, in $O(\log n)$ time and $O(n^2)$ work altogether.

For the sake of processor assignment in reporting actual longest chains, we also need to compute the number of points of S on the actual longest chain which is to be reported. Suppose a longest chain between points p and q in S is to be reported. The number of points of S on such a p-to-q chain can be obtained from the depth of p in the longest chain tree rooted at q. It is well-known that the depths of nodes in a rooted tree can be computed within the required complexity bounds by using the Euler Tour technique.[22]

To report an actual longest chain between points p and q in S, we do the following (WLOG, we assume that q dominates p). First, we go to the longest chain tree rooted at (say) q, and find the number of nodes on the path in that tree from node p to the root q. Let that number be k. The p-to-q path in that tree corresponds to a longest chain from p to q, which we would like to report. We do so by performing, in parallel, $k-1$ level-ancestor queries, using node p and integers $1, 2, \ldots, k-1$. Each query is handled by one processor in $O(1)$ time. These queries find each point on the

p-to-q chain. Finally, we report the k points of that chain in parallel, by assigning to k processors the task of reporting those k points (one point per processor).

4. Further Remarks

By using the methods we developed here in combination with other ideas, we can improve the processor complexity of the layers of maxima problem: We can achieve the same $O((\log n)^2)$ time complexity as the results [12] with $O(n^2/(\log n)^3)$ processors, instead of the $O(n^2/\log n)$ processors used in the algorithms.[12] The algorithm achieving these parallel bounds for the layers of maxima problem will be given.[23]

We conclude this paper by mentioning several open problems:

- Give an $O(n^2)$ time sequential algorithm for the weighted case.
- Give an $O(n^2)$ time sequential algorithm for the three dimensional version of the problem (unweighted).
- For the three dimensional version of the problem, give an NC parallel algorithm that uses a quadratic (to within a polylogarithmic factor) number of processors.

Acknowledgements

The authors would like to thank the referees for pointing out an error in the manuscript and for other helpful comments. This research was supported by the Leonardo Fibonacci Institute in Trento, Italy. Additional support was provided by the Air Force Office of Scientific Research under Grant AFOSR-90-0107, by the National Library of Medicine under Grant R01-LM05118, and by the National Science Foundation under Grant CCR-9202807. Part of this research was done while the first author was visiting LIPN, Paris, France. A preliminary version of this work appeared in *Lecture Notes in Computer Science, No. 709: Proc. of the Third Workshop on Algorithms and Data Structures*, Montreal, Canada, 1993, pp. 1–13.

References

1. R. B. K. Dewar, S. M. Merritt and M. Sharir, "Some modified algorithms for Dijkstra's longest upsequence problem", *Acta Informatica* **18** (1) (1982) 1–15.
2. E. W. Dijkstra, "Some beautiful arguments using mathematical induction", *Acta Informatica* **13** (1) (1980) 1–8.
3. M. L. Fredman, "On computing the length of longest increasing subsequences", *Discrete Mathematics* (1975) 29–35.
4. S. Even, A. Pnueli and A. Lempel, "Permutation graphs and transitive graphs", *Journal of the ACM* **19** (3) (1972) 400–410.
5. F. Gavril, "Algorithms for a maximum clique and a maximum independent set of a circle graph", *Networks* (1973) 261–273.

6. F. Gavril, "Algorithms on circular-arc graphs", *Networks* (1974) 357–369.

7. U. I. Gupta, D. T. Lee and Y.-T. Leung, "Efficient algorithms for interval graphs and circular-arc graphs", *Networks* (1982) 459–467.

8. W.-L. Hsu, "Maximum weight clique algorithms for circular-arc graphs and circle graphs", *SIAM J. Comput.* (1985) 224–231.

9. A. Pnueli, A. Lempel and S. Even, "Transitive orientation of graphs and identification of permutation graphs", *Canadian Journal of Math.* **23** (1) (1971) 160–175.

10. D. Rotem and U. Urrutia, "Finding maximum cliques in circle graphs", *Networks* (1981) 269–278.

11. A. Apostolico, M. J. Atallah and S. E. Hambrusch, "New clique and independent set algorithms for circle graphs", *Discrete Appl. Math.* **36** (1992) 1–24.

12. A. Aggarwal and J. Park, "Notes on searching in multidimensional monotone arrays", *Proc. 29th Annual IEEE Symposium on Foundations of Computer Science*, 1988, pp. 497–512.

13. A. Apostolico, M. J. Atallah, L. L. Larmore and H. S. McFaddin, "Efficient parallel algorithms for string editing and related problems", *SIAM J. Comput.* 19 (5) (1990) 968–988.

14. R. P. Brent, "The parallel evaluation of general arithmetic expressions", *Journal of the ACM* **21** (2) (1974) 201–206.

15. M. J. Atallah and D. Z. Chen, "Parallel rectilinear shortest paths with rectangular obstacles", *Computational Geometry: Theory and Applications* **1** (1991) 79–113.

16. M. J. Atallah and D. Z. Chen, "On parallel rectilinear obstacle-avoiding paths", *Computational Geometry: Theory and Applications* **3** (1993) 307–313.

17. C. P. Kruskal, L. Rudolph and M. Snir, "The power of parallel prefix", *IEEE Trans. Comput.* **C-34** (1985) 965–968.

18. R. E. Ladner and M. J. Fischer, "Parallel prefix computation", *Journal of the ACM* **27** (4) (1980) 831–838.

19. M. J. Atallah and S. R. Kosaraju, "An efficient parallel algorithm for the row minima of a totally monotone matrix", *Proc. 2nd ACM-SIAM Symp. on Discrete Algorithms*, San Francisco, 1991, pp. 394–403.

20. O. Berkman and U. Vishkin, "Finding level-ancestors in trees", Tech. Rept. UMIACS-TR-91-9, University of Maryland, 1991.

21. O. Berkman and U. Vishkin, personal communication.

22. R. E. Tarjan and U. Vishkin, "An efficient parallel biconnectivity algorithm", *SIAM J. Comput.* **14** (4) (1985) 862–874.

23. M. J. Atallah and D. Z. Chen, manuscript, 1993.

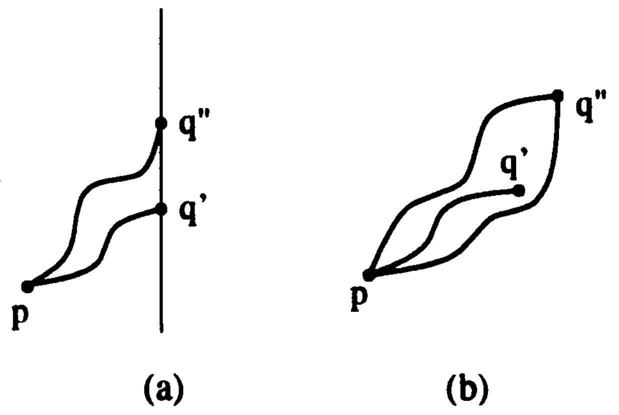

Figure 1

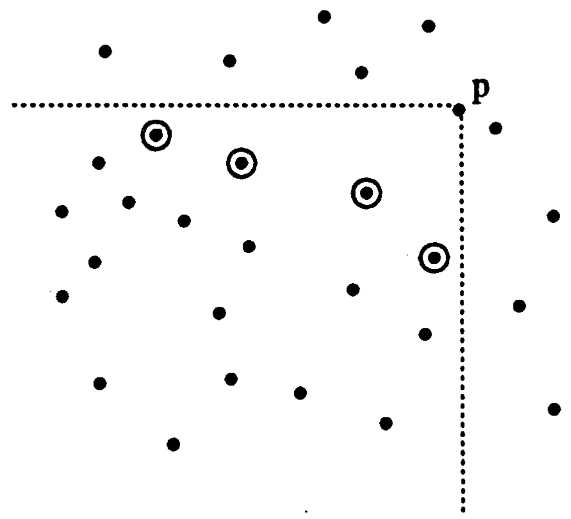

Figure 2

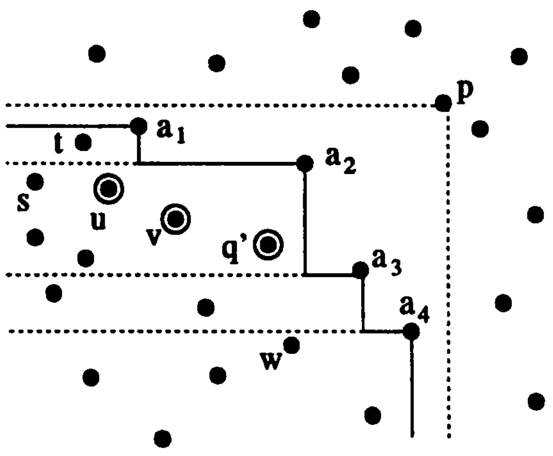

Figure 3

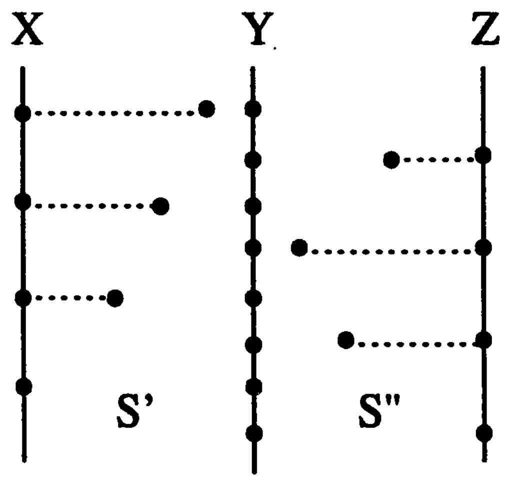

Figure 4

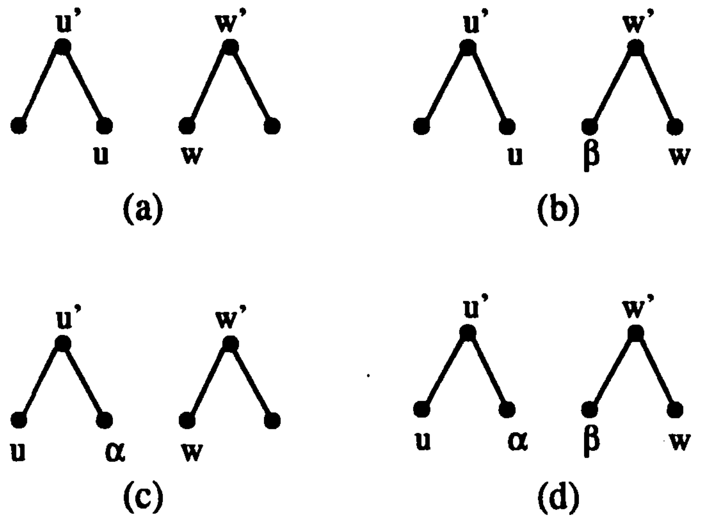

Figure 5