PLANAR STAGE GRAPHS: CHARACTERIZATIONS AND APPLICATIONS

Frank Bauernöppel[||*‡†]
(bauernoe@informatik.hu-berlin.de)

Evangelos Kranakis[*†]
(kranakis@scs.carleton.ca)

Danny Krizanc[*†]
(krizanc@scs.carleton.ca)

Anil Maheshwari [‡†*‡†]
(manil@tifrvax.tifr.res.in)

Jörg-Rüdiger Sack[*†]
(sack@scs.carleton.ca)

Jorge Urrutia[§†]
(jorge@csi.uottawa.ca)

Abstract

We consider combinatorial and algorithmic aspects of the well-known paradigm "killing two birds with one stone". We define a stage graph as follows: vertices are the points from a planar point set, and $\{u, v\}$ is an edge if and only if the (infinite, straight) line segment joining u to v intersects a given line segment, called a stage. We show that a graph is a stage graph if and only if it is a permutation graph. The characterization results in a compact linear space representation of stage graphs. This has been exploited for designing improved algorithms for matching in permutation graphs, two processor task scheduling for dependency graphs known to be permutation graphs, and dominance related problems for planar point sets.

1980 Mathematics Subject Classification: 68R10, 68U05
CR Categories: F.2.2
Key Words and Phrases: Algorithms and Data Structures, Dominance, Matching, Processor Scheduling, Permutational Graphs, Coding of Orders, 2-dimensional Partial Orders, Stage Graphs.
Carleton University, School of Computer Science: TR-95-06
Note: This technical report is a revised and expanded version of TR-239

Results in this paper will appear in the proceedings of ICALP 95.

[*]Carleton University, School of Computer Science, Ottawa, ON, Canada

[†]Research supported in part by NSERC (Natural Sciences and Engineering Research Council of Canada) grant.

[‡]Tata Institute of Fundamental Research, Bombay 400 005, India

[§]University of Ottawa, Department of Computer Science, Ottawa, ON, Canada

[||]Institut für Informatik, Humboldt-Universität zu Berlin, 10099 Berlin, Germany

[**]Research supported in part under an R&D agreement between Carleton University and ALMERCO Inc.

[††]Work by the author was carried during a stay at Carleton University.

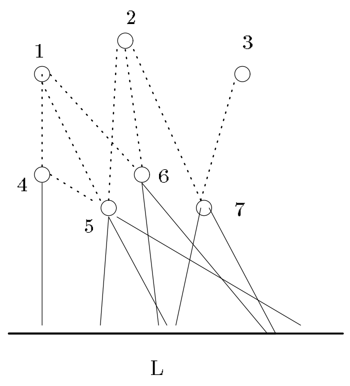

Figure 1: Stage representation of the graph with vertices $1, 2, 3, 4, 5, 6, 7$ and edges $\{1,4\}, \{1,5\}, \{1,6\}, \{2,5\}, \{2,6\}, \{2,7\}, \{3,7\}, \{4,5\}$.

.

1 Introduction

Suppose that an archer is hunting birds flying over hunting grounds described as a bounded region possibly with holes formed by obstacles such as mountains, lakes, dense forests, etc. In an attempt to minimize the number of arrows used, the archer tries to identify pairs of birds that can be pierced by a single arrow; this is possible, if the positions of two birds line up with some point on the hunting grounds. This corresponds to the well known paradigm of "killing two birds with one stone."

The planar archer problem can be modeled as follows: assume that $X = \{p_1, \ldots, p_n\}$ is a collection of points in the plane (in general position) such that the y-coordinate of each element of X is strictly greater than zero and let L be a line segment contained in the x-axis called stage. Given X and L construct a graph $G(X, L)$ with vertex set X such that two vertices p_i, p_j of $G(X, L)$ are adjacent if the line through p_i and p_j intersects L. $G(X, L)$ will be called the stage graph of X and L (for an illustraction see Figure 1).

Applications of stage graphs may arise in several problems such as the positioning of floodlights to illuminate fixed objects in space and the positioning of directional satellite antennae to pick up signals from ground stations, not to mention the traditional problem of "killing two birds with one stone." An important relationship to two-processor task scheduling and to dominance related problems will be discussed and exploited.

1.1 Results of the Paper

The organization of the paper is as follows.

In Section 2 we present our characterization theorem for stage graphs. We prove that, the family of stage graphs is exactly the family of permutation graphs. This yields an efficient algorithm for recognizing such graphs. The characterization implies a compact linear space representation (encoding) for stage graphs. Also viewing permutation graphs as stage graphs allows a geometric interpretation of permutation graphs. We exploit this for the design of several algorithms including an efficient solution to the archer's problem and to dominance related problems.

In Section 3 we study the archer's problem. The problem of minimizing the number of arrows the archer needs naturally corresponds to that of finding a maximum matching in stage graphs. Therefore it is possible to solve the problem e.g., by using the Micali and Vazirani matching algorithm [15]. This results in an $O(\sqrt{n}m)$ algorithm where n and m are the number of vertices and edges of the graph respectively. A more efficient algorithm is obtained when stating the problem as a two-processor task scheduling problem. Efficient algorithms for finding tightest two-processor schedules are known [3, 7, 8, 9]. We follow the approach of [3] that leads to an $O(n+m)$ time algorithm for the scheduling problem [19]. Through vector dominance and using computational geometry techniques we establish that the problem has an $O(n\log^3 n)$ solution. We therefore not only solve the archer's problem efficiently, but also provide a novel and improved algorithm for matching in permutation graphs. Furthermore, if the dependency graph of a scheduling problem is known to be a permutation graph, then we now have an improved two-processor scheduling algorithm (if the number of edges is $\Omega(n\log^3 n)$).

In Section 4 we present conceptually simple, new, and improved algorithms for vector dominance and rectangle query problems. Let $P = \{p_1, p_2, ..., p_n\}$ be a planar point set of n distinct points $p_i = (x_i, y_i)$, $i = 1, ..., n$. A point p_i is said to *dominate* a point p_j, if $x_i \geq x_j$ and $y_i \geq y_j$ and $i \neq j$. We present new simple and optimal sequential and EREW PRAM algorithms for reporting all dominance pairs. The algorithms also present an improvement in that the previous algorithm [10] for that problem required CREW processors. A problem related to dominances is the *rectangle query problem* for planar point sets P. A query consists of a pair of points (p_i, p_j), where $p_i, p_j \in P$, and we need to answer whether the rectangle formed by the query points is empty or not. We design an $O(n\log n)$ space data structure which answers rectangle queries in $O(1)$ time. The data structure can be constructed in sequential $O(n\log n)$ time and in $O(\log n)$ time using n EREW PRAM processors. Our parallel rectangle query algorithm improves on previous ($O(n^2)$ space, $O(1)$ query) or ($O(n\log n)$ space, $O(logn)$ query) results.

Finally, in Section 5 we conclude with a few open problems.

2 Characteriation of Single Stage Graphs

Let L be a line segment L, the "stage", contained in the x-axis of the plane and a set of points $X = \{p_1, \ldots, p_n\}$ in general position with positive y-coordinates. We give a characterization for the graph $G(X, L)$ with vertex set X in which two vertices are adjacent if the line connecting them intersects L.

Let $P(X, <)$ be a poset. Then a realizer of P of size $k+1$ is a collection of linear orders $\{L_0(X, <_0), L_1(X, <_1), \ldots, L_k(X, <_k)\}$ such that

$$L_0(X, <_0) \cap L_1(X, <_1) \cap \cdots \cap L_k(X, <_k) = P(X, <),$$

where the intersection is defined by $x < y \Leftrightarrow x <_i y$, for all i. It can easily be proved that every poset can be obtained as the intersection of a number of linear orders. Dushnik and Miller [6] define the dimension of P, denoted $\dim P$, to be the smallest possible size of a realizer of P. Partial orders of dimension 2 are known to be permutation graphs. In the following theorem we will establish that stage graphs are permutation graphs. This yields an $O(\min\{n^2, n + m\log n\})$ algorithm to recognize stage graphs with m edges and n vertices.

Theorem 2.1 *A graph G is a stage graph if and only if G is a permutation graph.*

Proof Consider a set $X = \{p_1, \ldots, p_n\}$ of n points on the plane with y-coordinates greater than zero and a line segment L contained in the x-axis, with end points p and q. Let $G(X, L)$ be the stage graph of X and L. We start first by proving that $G(X, L)$ is a comparability graph, i.e. we show that it is possible to orient the edges of $G(X, L)$ such that if $p_i \to p_j$ and $p_j \to p_k$ then $p_i \to p_k$. To this end, let us assume that two vertices p_i and p_j of $G(X, L)$ are adjacent, i.e. the line through p_i and p_j intersects L. We orient the edge $\{p_i, p_j\}$ of $G(X, L)$, $p_i \to p_j$ if the y-coordinate of p_i is smaller than that of p_j, otherwise we orient $p_j \to p_i$. We now prove that the orientation thus obtained in $G(X, L)$ is transitive. Observe that $p_j \to p_i$ if and only if the triangle $\Delta(p_i, p, q)$ defined by p_i and the end points p and q of L is contained in the triangle $\Delta(p_j, p, q)$ defined by p_j and p and q. Thus if $p_i \to p_j$ and $p_j \to p_k$ then $\Delta(p_k, p, q) \supset \Delta(p_j, p, q) \supset \Delta(p_i, p, q)$ and thus $p_i \to p_k$. This orientation of $G(X, L)$ defines a partial order $P(X, <)$ on X in which $p_i < p_j$ if $p_i \to p_j$.

We now show that $P(X, <)$ has dimension 2. To prove this we will produce two linear extensions $L_1(X, <_1)$ and $L_2(X, <_2)$ of $P(X, <)$ such that $L_1(X, <_1) \cap L_2(X, <_2) = P(X, <)$. To produce $L_1(X, <_1)$ sort the points of X in the counterclockwise direction with respect to p, i.e. $p_i <_1 p_j$ if the slope of the line joining p_i to p is smaller than the slope of the line joining p_j to p. In $L_2(X, <_2)$ we now define $p_i <_2 p_j$ if the slope of the line joining p_i to q is greater than the slope of the line joining p_j to q (see Figure 2, where $L_1(X, <_1) = \{p_1 <_1 p_4 <_1 p_3 <_1 p_5 <_1 p_2\}$ and $L_2(X, <_2) = \{p_1 <_2 p_3 <_2 p_4 <_2 p_2 <_2 p_5\}$). It now follows that $P(X, <) = L_1(X, <_1) \cap L_2(X, <_2)$. Partial orders of dimension 2 are precisely permutation graphs.

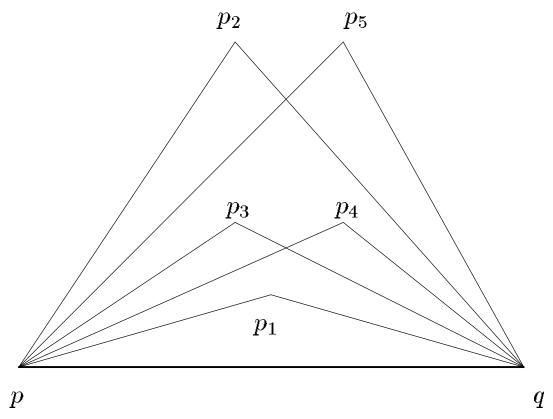

Figure 2: The orders $L_1(X, <_1)$ and $L_2(X, <_2)$

.

Conversely, let $P(X, <)$ be an ordered set of dimension 2 and $L_1(X, <_1)$, $L_2(X, <_2)$ be two total orders on X such that $P(X, <) = L_1(X, <_1) \cap L_2(X, <_2)$. Choose two points p, q on the x-axis as depicted in Figure 3. Let p_i be an element of X. Let $r(i)$ and $s(i)$ be the ranks of p_i in $L_1(X, <_1)$ and $L_2(X, <_2)$, respectively. Consider a set $\{\lambda_1, \ldots, \lambda_n\}$ of n lines through p sorted in increasing order according to their slopes and a set $\{\beta_1, \ldots, \beta_n\}$ of n lines through q sorted in decreasing order according to their slopes such that each λ_i intersects each β_j at a point with positive y-coordinate, $1 \leq i, j \leq n$. Let us label with p_i the point at which $\lambda_{r(i)}$ and $\beta_{s(i)}$ intersect and identify the points of X with $p_1, \ldots, p_n$ (see Figure 2, where $L_1(X, < 1) = \{p_2 <_1 p_4 <_1 p_3 <_1 p_1 <_1 p_5\}$ and $L_2(X, <) = \{p_3 <_2 p_1 <_2 p_5 <_2 p_2 <_2 p_4\}$). It is now easy to see that the set X of points on the plane labeled $p_1, \ldots, p_n$ and the line segment L are such that $G(X, L)$ is the stage graph of $P(X, <)$. ■

Corollary 2.1 *Stage graphs can be recognized in $O(\min\{n^2, n + m \log n\})$ time.*

PROOF Recognition of orders of dimension two (permutation graphs) can be done in $O(\min\{n^2, n + m \log n\})$ time [14, 17, 20]. ■

3 Matching/Scheduling Algorithms

In this section we provide an efficient solution to the archer's problem. As mentioned earlier, the problem of minimizing the number of arrows the archer needs naturally corresponds to that of finding a maximum matching in stage graphs. Using the Micali and Vazirani matching algorithm [15] an $O(\sqrt{n}m)$ solution is obtained where n and m are the number of vertices and edges of the

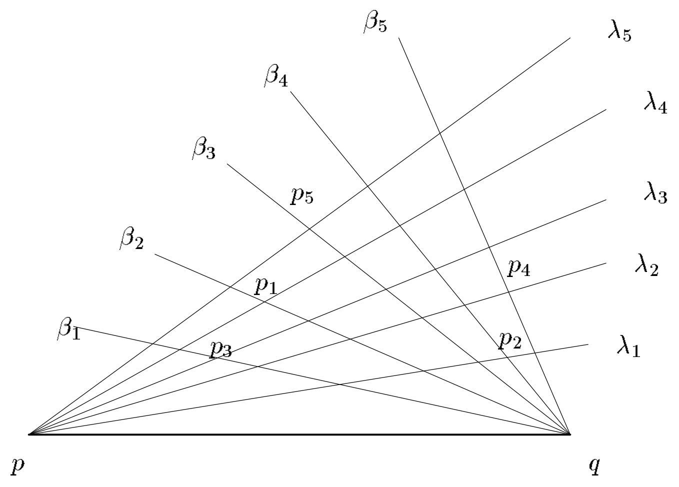

Figure 3: The orders L_1, L_2

graph, respectively. A more efficient solution is obtained by exploiting a relationship between matching and processor scheduling discussed in Section 3.1. Our matching algorithm as presented in Section 3.2 is based on the characterization theorem for stage graphs established above. Our result implies novel and improved solutions to matching in permutation graphs and two-processor task scheduling for permutation graph dependencies.

3.1 Relationship Between Matching and Processor Scheduling

As pointed out, e.g., in [16], there is an important relation between maximum matchings in co–comparability graphs and the following scheduling problem: Let $G = (V, E)$ be a directed acyclic graph; let G have n vertices and m edges. Vertex $v \in V$ is a *successor* of a vertex $u \in V$ if there is a directed path from u to v in G. A *two-processor scheduling* for G is an assignment of time units $1, 2, 3, \ldots$ to the vertices $v \in V$ such that

1. each vertex $v \in V$ is assigned exactly one time unit,
2. at most two elements are assigned the same time unit, and
3. if v is a successor of u in G, then u is assigned a lower time unit than v.

The edges of G represent dependencies among the set of n vertices (tasks) to be executed. The largest time unit assigned to a vertex is called the length of the schedule.

It has been shown that the following holds [7]: Given a directed acyclic graph G with n vertices then there is a two-processor scheduling for G of length

l iff there is a matching of size $n - l$ in $G^{*\prime}$, the undirected complement of the transitive closure G^* of G. Two vertices u and v are connected by an undirected edge in G' iff neither uv nor vu is an edge in graph G. Moreover, a matching in G' can be obtained from a schedule by simply matching all pairs of vertices that are assigned the same time unit. Note that G' is a co–comparability graph.

Efficient algorithms for finding a tightest two-processor schedule are known [8, 9]. We follow the approach of [3] that leads to a linear time algorithm for the scheduling problem [19] when the graph G is transitively closed. This approach uses a vertex numbering assigning numbers $1, 2, \ldots$ to the n vertices of G in increasing order. By $L(u)$ we denote the label of vertex u and by $N(u)$ we denote the list $(L(v_1), L(v_2), \ldots, L(v_k))$ of the labels of the successors v_i of u in G sorted in decreasing order.

Suppose the numbers $1, 2, \ldots, k-1$ have already been assigned. A vertex u is labeled with value $L(u) = k$, if

1. all successors of u in G are already labeled, and
2. for each other vertex u' fulfilling 1, the sorted list $N(u')$ is lexicographically not smaller than $N(u)$. (Ties are broken arbitrarily.)

Once the vertex labeling is complete, the matching is found in a greedy manner by a list schedule according to decreasing vertex labels: Starting with the highest label n, a vertex is matched with the highest label possible.

The $|V| + |E|$ time algorithm of [19] is optimal when the restriction graph G is given explicitly. This yields a $O(n^2)$ time maximum matching algorithm for the corresponding co–comparability graph G'.

Since stage graphs can be represented in $O(n)$ space, we are interested in a faster algorithm for this class of graphs.

3.2 Efficient Matching and Processor Scheduling

We use the stage characterization theorem to obtain more efficient matching and scheduling algorithms.

Observe that the complement of a permutation graph is also a permutation graph. Hence the problem reduces to that of finding an optimal two processor schedule of a permutation graph. Using the above derived geometric interpretation of permutation graphs we show that simple geometric arguments and data structures suffice to design a matching algorithm whose run-time is sublinear in the number of edges of the graph.

To achieve this, we partition the vertices into *levels*. The level of a vertex $v \in V$ is the length of the longest path from v to a vertex of outdegree zero. Implicitly, a vertex v points to a vertex u, if both x and y-coordinate of u are bigger than that of v. This partitioning into levels corresponds to vertex domination in the geometric representation of a permutation graph. The partition into levels can therefore be done in $O(n \log n)$ time (see e.g.,[18]).

It is easy to see that the following holds: If vertex $u \in V$ is at a higher level than vertex $v \in V$, then $L(u) > L(v)$. This can be shown by an inductive argument. Let l_u,l_v be the level of u and v respectively. Then, v has at least one successor at level $l_v - 1 \geq l_u$ whereas u has no successor at this level. It remains to determine the order in which the vertices within a level are labeled. Any algorithm which explicitely determines the sorted list $N(v)$ requires $\Omega(n^2)$ time (based on the total size). Instead we determine the order of the vertices by computing the sorted lists $N(v)$ only partially using a geometric argument.

Denote by $DomReg(p)$ the upper right quadrant of an axis-aligned coordinate system whose origin is at point p. In this section, a point p dominates a point q if q lies in $DomReg(p)$. Let R be any region of the plane, then $Max(R)$ denotes the maximum label of all labeled points which lie in R, it is set to zero if R contains no labeled point.

Now let u and v be two points on a common level and assume w.l.o.g. that u lies above v, i.e. u's $y-$coordinate is larger than v's. The intersection of $DomReg(u)$ and $DomReg(v)$ is a quadrant called $SharedQ(u, v)$. Then $DomReg(u)$ can be partitioned into $SharedQ(u, v)$ and the remaining half-open rectangle, called $Rec(u)$; similarly, for v. Now, we observe that $L(u) > L(v)$ if and only if $Max(Rec(u))$ is greater than $Max(Rec(v))$.

This reduces the problem of performing a comparison operation of the form "$L(u) > L(v)$" (as needed for sorting each layer) to a comparison between two integers (labels) obtained via Maximum-Labeled-Element-Queries in half-open rectangles. There are different approaches to solving such queries: one is to state the problem as a (dynamic) 3-d range searching problem where the third coordinate is the label, the other, taken here, is to use the traditional range-range priority search trees (see e.g., [18]).

The primary tree stores the points sorted by x-coordinate in its leaves. Located at each internal node of the primary tree is a y-sorted secondary tree containing all points in the subtree; in addition, the secondary tree contains, at each internal node, the maximum label of all elements stored in its subtree.

To perform a Maximum-Labeled-Element-Query, we must find the maximum labeled point in the region bounded by $[x_a, x_b]$ and $[y_a, y_b]$ (where one of the coordinates is infinity). We locate the $O(\log n)$ roots of subtrees spanning the x-range $[x_a, x_b]$ and for each of these we use the $O(\log n)$ secondary trees spanning the y-range to find the maximum labels of all points in the entire half-open rectangle. It is easily observed that the time for a Maximum-Labeled-Element-Query is $O(\log^2 n)$. In total, $O(\log^2 n)$ roots of subtrees are to be located and the Maximum-Labeled-Element is the maximum of the $O(\log^2 n)$ candidate values so obtained in the secondary trees.

Initially we build the primary and secondary trees for all points, setting all labels to zero. The labels for layer i are calculated using (only) the labels of layers 1 to i-1; the locations of all points remain unchanged. Updating the label of a point requires updating the corresponding leaf and the nodes (storing the secondary trees) on the path to the root. The total time per label-update is therefore $O(\log^2 n)$. Next we summarize the algorithm.

1. Build a range-range priority search tree on all points, setting the labels of all points equal to zero.
2. Partition the point set into layers, $1, 2, \ldots, k$

```
3. Label the $n_1$ points on the first layer $1, 2, \ldots, n_1$ (arbitrarily)
4. For layer $i = 2, 3, \ldots, k$ do
   begin
   sort the points on layer $i$ (using the above described comparison operator)
   assign consecutive labels to the points
   update the labels in the search tree structure
   end
5. Perform a greedy matching on the labeled graph.
```

If an optimal sorting algorithm is used, the total number of queries can be bounded by $\sum_{i=1}^{k} n_i \log n_i$, where n_i denotes the cardinality of layer i. The matching can be done by a sequence of $O(n)$ Maximum-Labeled-Element-Queries using the quadrant $DomReg(p)$ for finding point q to be matched with p. (This requires $O(n)$ deletions as well.) From the above it follows that the total time complexity of the matching algorithm is $O(n \log^3 n)$.

Theorem 3.1 *Matching in permutational graphs can be performed in time $O(n \log^3 n)$, where n is the number of vertices of G.*

Corollary 3.1 *The problem of minimizing the total number of arrows to kill all n birds can be solved in $O(n \log^3 n)$ time.*

Since complement graphs of permutation graphs are permutation graphs we get the following result.

Theorem 3.2 *Two-processor task scheduling for dependency graphs known to be permutation graphs can be solved in $O(n \log^3 n)$ time where n is the number of vertices (not dependencies).*

4 Dominance Problems

4.1 Motivation and Related Results

Dominance problems arise naturally in a variety of applications and they are directly related to well studied geometric and non-geometric problems. These problems include: range searching, finding maximal elements and minimal layers, computing a largest area empty rectangle in a point set, determining the longest common sequence between two strings, and interval/rectangle intersection problems, etc. Dominance computations were also required for our matching algorithm. Again we use the characterization theorem for stage graphs.

Let $P = \{p_1, p_2, ..., p_n\}$ be a planar point set of n distinct points $p_i = (x_i, y_i)$, $i = 1, ..., n$. A point p_i is said to *dominate* a point p_j, if $x_i \geq x_j$ and $y_i \geq y_j$ and $i \neq j$. Preparata and Shamos [18] presented optimal sequential algorithms for counting and reporting the dominances for each point of the set P and running in $O(n \log n)$ and $O(n \log n + k)$ time, respectively, where k is the total number of of dominance pairs. In the reporting mode of the problem, all dominance pairs are to be enumerated. Goodrich [10] solved this problem in $O(\log n)$ time using $O(n + k/\log n)$ CREW PRAM processors, where k is the total number of dominance pairs. The two-set dominance counting problem was solved by Atallah et al. [1] in optimal $O(\log n)$ time using $O(n)$ processors, where n is the total number of points in the given sets. In this problem, given two point sets A and B, all pairs (a, b) are to be counted where $a \in A$ dominates $b \in B$. In the reporting mode of the problem, all dominance pairs are to be enumerated. Goodrich [10] solved this problem in $O(\log n)$ time using $O(n/\log n + k)$ CREW PRAM processors, where k is the total number of dominance pairs. In [5] direct dominance problems have been studied for the CREW-PRAM.

A problem related to dominances is the *rectangle query problem* for planar point sets P. A query consists of a pair of points (p_i, p_j), where $p_i, p_j \in P$, and we need to answer whether the rectangle formed by the query points is empty or not. Such rectangle queries find application e.g., in data bases. Given $O(n^2)$ space, queries can easily be answered in $O(1)$ time. The space can be reduced to $O(n \log n)$ using data structures that support range searching [18], unfortunately the query time increases to $O(\log n)$.

4.2 Our Results

We present a simple optimal parallel algorithm for reporting all dominances in a planar n-point set; it runs in $O(\log n)$ time using $O(n + k/\log n)$ EREW PRAM processors. For the rectangle query problem, we provide an $O(n \log n)$ size data-structure, where the queries can still be answered in $O(1)$ time. Furthermore, the data-structure is very-simple and can be computed in sequential $O(n \log n)$ time and in parallel $O(\log n)$ time using $O(n)$ EREW PRAM processors, respectively. For details on the parallel model of computation, see e.g., [12, 11].

Our methods for solving both problems is different from the existing methods; we reduce the problems to one dimensional problems. This is achieved by ordering the points with respect to x-coordinate and then redefining these problems with respect to the corresponding permutation on y-axis.

4.3 Algorithms for Reporting Dominances

In this subsection, we provide algorithms for reporting dominances of a planar point set P. Without loss of generality assume that the points of the set $P = \{p_1, p_2, ..., p_n\}$, where $p_i = (x_i, y_i)$, $i = 1, ..., n$, are sorted with respect to increasing x-coordinate. Therefore, we relabel each point p_i by its index i and from now on, we refer to a point p_i by its index i. Let Y be the array consisting of labels of points in P, sorted with respect to increasing y-coordinate; i.e. Y is a permutation of $\{1, ..., n\}$. Let i appear at the position pos_i, where $1 \leq pos_i \leq n$, in Y. From the above definitions it follows that a point i dominates a point $j \in P$ if and only if $i > j$ and $pos_i > pos_j$. Hence the points dominated by i are the elements of the subarray $Y[1, ..., pos_i]$ which are less than i. So the dominance problem reduces to that of reporting all elements of subarray $Y[1, ..., i-1]$ which are less than $Y[i]$, for all $i \in \{2, ..., n\}$.

We provide first a sequential algorithm for the above problem and then show that it can be easily parallelized. The sequential algorithm is based on the merge sort algorithm; it runs in $O(n \log n + k)$ time using linear space, where k is the total number of dominance relations in the given point set P.

The sequential algorithm has $O(\log n)$ merge stages. In order to simplify notation, we present the last merge stage. Assume that we know all dominances for each point within subarrays $Y[1, ..., n/2]$ and $Y[n/2+1, ..., n]$. We wish to compute dominances for each point in $Y[1, ..., n]$. Observe that we need only compute the points dominated by $Y[n/2+1, ..., n]$ in $Y[1, ..., n/2]$, since no point in $Y[1, ..., n/2]$ dominates any point in $Y[n/2+1, ..., n]$. The dominances are computed as follows. First note that the arrays $Y[1, ..., n/2]$ and $Y[n/2+1, ..., n]$ have already been sorted in increasing order during the recursion. Now rank each element of $Y[n/2+1, ..., n]$ in $Y[1, ..., n/2]$. Suppose an element $Y[i]$, where $n/2+1 \leq i \leq n$, is ranked at the position j $(1 \leq j \leq n/2)$ in $Y[1, ..., n/2]$, the points dominated by $Y[i]$ in $Y[1, ..., n/2]$ are $Y[1]$, $Y[2]$, ..., $Y[j]$. After cross ranking, we can report dominances in time proportional to the number of dominance pairs. We summarize the result in the following theorem.

Theorem 4.1 *All dominances of an n-point planar set can be computed in $O(n \log n + k)$ time using $O(n)$ space, where k is the total number of dominance pairs.*

PROOF The correctness of the algorithm is straightforward. Now we analyze its complexity. The merge-sort algorithm takes $O(n \log n)$ time using $O(n)$ space. During each stage in merging, the ranking of the sub-arrays can be achieved in linear time with respect to their sizes. Since in each stage we report a set of new dominance pairs, overall time complexity of the algorithm follows. In order to perform the $i+1$st stage of merge-sort, we need only the computation of the ith stage; thus the algorithm requires only linear space. ∎

Now we parallelize the above algorithm by using the results of [4, 13]. The parallel-merge sort algorithm of [4] cross-ranks elements of each subarray during each stage of merging. As observed above, after cross ranking, the problem reduces to that of reporting subarrays $Y[1, ..., j]$ for an appropriate j, where $1 \leq j \leq n/2$, for each $Y[i]$, where $n/2+1 \leq i \leq n$. Subarrays can be optimally reported on EREW PRAM by the algorithm of [13]. We summarize the result in the following theorem.

Theorem 4.2 *All dominances of an n-point planar set can be computed in $O(\log n)$ time using $O(n + k/\log n)$ processors on the EREW PRAM, where k is the total number of dominances.*

PROOF The correctness of the algorithm is straightforward. We analyze the complexity of the algorithm. Parallel merge sort requires $O(\log n)$ time using $O(n)$ processors on the EREW PRAM [4]. Further, it also cross ranks subarrays in each step. Using this information, the value of k can be computed in $O(\log n)$ time using $O(n)$ processors. Allocate $O(n + k/\log n)$ processors to report all dominances. We also need to store the sorted sub-arrays at each intermediate stage in merge-sort. Using the algorithm of [13], the required subarrays can be reported in $O(\log n)$ time using using $O(n + k/\log n)$ processors [13]. Hence, it follows, that all dominances can be reported in $O(\log n)$ time using $O(n + k/\log n)$ EREW PRAM processors. ■

4.4 Algorithms for the Rectangle Query Problem

In this subsection we address the rectangle query problem. Given an n-point planar set P, the queries are of the form (p_i, p_j), where $p_i, p_j \in P$, and we need to output, whether or not the rectangle formed by p_i and p_j contains a point of P in its interior. We provide sequential and parallel algorithms to compute an $O(n \log n)$ size data-structure, such that the queries can be answered in $O(1)$ time.

As in the previous subsection, we assume that the points of the set $P = \{p_1, p_2, ..., p_n\}$, where $p_i = (x_i, y_i)$, $i = 1, ..., n$, are sorted with respect to increasing x-coordinate. Therefore, we relabel each point p_i by its index i. We refer to a point p_i by its index i. Notice that our queries are of type (i, j), where $1 \leq i, j \leq n$. Furthermore, we can assume that $i < j$, otherwise we can interchange i and j.

We compute two data-structures, the first one answers the queries where $y_i \leq y_j$, and the other one answers the queries where $y_i > y_j$. Since the procedure for computing both data-structures and answering the corresponding queries is analogous, we only discuss the computation of data-structure which handles the queries where $y_i \leq y_j$.

Let Y be the array corresponding to the labels of points in P sorted with respect to *increasing* y-coordinate. Let (i, j) be a query pair, where $i < j$ and $y_i \leq y_j$. Let i appear at the position pos_i in Y, where $1 \leq pos_i \leq n$. The following lemma enables us to reduce our problem of detecting whether a rectangle is empty or not to a one dimensional problem on Y.

Lemma 4.1 *The rectangle formed by (p_i, p_j) is empty if and only if there does not exist any element $Y[k]$, such that $i \leq Y[k] \leq j$, where $pos_i < k < pos_j$.*

PROOF Follows from the definition of the array Y. ■

In the following we first state a sequential algorithm to compute a data-structure, which can answer the existence of $Y[k]$ between (pos_i, pos_j) as stated in the above lemma, and then show how the queries can be answered. Further, we show that the algorithm for computing the data-structure can be easily parallelized.

Before stating our algorithm, we simplify notation by restating the problem. Our aim is to preprocess the array Y (assume $n = 2^l$) such that, given any two indices a and b, where $1 \leq a < b \leq n$, we can determine whether there exist an element in the subarray $\{Y[a+1], ..., Y[b-1]\}$, which is between $Y[a]$ and $Y[b]$ in $O(1)$ sequential time. Intuitively, it seems that we need to precompute this information for some subarrays, and then given a query array, the relevant information should be deduced from a constant number of pre-computed subarrays. We achieve our goal by constructing a complete binary tree T on the elements of Y such that each internal node u of T keeps some information about the array determined by the leaves in the subtree rooted at u. In the following, we precisely state the information maintained at each internal node u of T.

Let $LCA(a,b)$ denote the lowest common ancestor node of the leaves of T holding $Y[a]$ and $Y[b]$. Given two indices a and b, we can determine $LCA(a,b)$, say the node u, of T in $O(1)$ sequential time since T is a complete binary tree. If the leaves of the subtree rooted at u correspond exactly to the subarray $\{Y[a],...,Y[b]\}$, then it is sufficient to store an information at u, about the presence or absence of an element between $Y[a]$ and $Y[b]$ in the subarray $\{Y[a+1],...,Y[b-1]\}$. However, the subarray associated with u, denoted by Y_u, is typically of the form of $\{Y[l],...,Y[a],...,Y[b],...,Y[r]\}$, where $l \leq a < b \leq r$. Hence the information stored at the node u is not sufficient to answer our query, and some additional information is needed, as described next.

Let v and w be the left and right child of u, respectively. Let the subarrays associated with v and w, respectively are $Y_v = \{Y[l],...,Y[a],...,Y[p]\}$ and $Y_w = \{Y[p+1],...,Y[b],...,Y[r]\}$ for some $a \leq p < b$. Notice that the subarrays Y_v and Y_w partition Y_u. Let us define two quantities, called *suffix minima* and *prefix maxima*, respectively over the elements of arrays Y_v and Y_w, which we need for answering our queries.

For any α, where $l \leq \alpha \leq p$, the suffix minima for α in Y_v is defined as follows. Among the elements of the subarray $\{Y[\alpha+1],...,Y[p]\}$ consider only the set of elements which are larger than $Y[\alpha]$, and call this set as $Suff_\alpha$. If $Suff_\alpha \neq \emptyset$, then the suffix minima for α is the element with the minimum value in $Suff_\alpha$, otherwise suffix minima does not exist for α. Similarly we define prefix maxima. For any β, where $p+1 \leq \beta \leq r$, the prefix maxima for β in Y_w is defined as follows. Among the elements of the subarray $\{Y[p+1],...,Y[\beta-1]\}$, consider only the set of elements which are smaller than $Y[\beta]$, and call this set $Pref_\beta$. If $Pref_\beta \neq \emptyset$, then the prefix maxima for β is the element with the maximum value in $Pref_\beta$, otherwise it does not exist for β.

Let us first analyze the complexity of constructing the whole data structure. The algorithm constructs a complete binary tree whose leaves are the elements of Y such that each internal node u has associated with it two arrays, suffix minima and prefix maxima arrays. It can be seen that the data-structure needs $O(n \log n)$ space. Now we show that the data-structure can be computed in $O(n \log n)$ time.

We make two copies of array Y, and on one copy we perform a merge-sort algorithm. The merge-sort algorithm, computes a complete binary tree T', over Y, and at each internal node u of T' it computes a sorted list of elements in the subtree rooted at u. Furthermore, if v and w are the left and right child of u in T', then we also cross rank the elements of v and w. Also store the sorted list, and the cross ranking information, at each internal node of T'. It is easy to see that this can be accomplished in $O(n \log n)$ time and space. Now we work on the other copy of Y to compute the suffix minima and prefix maxima arrays. Consider a node u of T, and let v and w be its left and right child, respectively, as mentioned above. Assume that we know the suffix minima and prefix maxima arrays for v and w and we wish to compute these arrays for the node u. Notice that the suffix minima and prefix maxima for each element in u can be computed by using the cross ranking information among the elements of v and w in the merge-sort tree T'.

It is easy to see that the above data-structure can be computed in $O(n \log n)$ sequential time and in parallel in $O(\log n)$ time using $O(n)$ EREW PRAM processors by using the parallel merge-sort algorithm of [4]. Now we show that the queries can be answered in $O(1)$ time. The following lemma is crucial for the correctness and the complexity.

Lemma 4.2 *Let u be the lowest common ancestor node corresponding to a and b in T, where $a < b$. Let v and w be the left and right child of u, respectively. Let the subarrays associated with v and w be*

$$Y_v = \{Y[l], ..., Y[a], ..., Y[p]\} \text{ and } Y_w = \{Y[p+1], ..., Y[b], ..., Y[r]\},$$

where $a \leq p < b$, respectively. There exists an element between $Y[a]$ and $Y[b]$ in the subarray $\{Y[a+1], ..., Y[b-1]\}$ if and only if either the suffix minima of $Y[a]$ in Y_v is smaller than $Y[b]$, if it exists, or the prefix maxima of $Y[b]$ in Y_w is larger than $Y[a]$, if it exists.

Proof Follows from the definition of suffix minima and prefix maxima. ■

Let us recall our problem. We are given a set P of points, sorted with respect to x-coordinate and labeled accordingly. Our queries are of the form (p_i, p_j), where $p_i, p_j \in P$. We want to report whether the rectangle formed by p_i and p_j is empty or not. We first test whether $i < j$, if not, we interchange i, j. We compute two data-structures, one to handle the queries where $y_i < y_j$ and the other one to handle the queries where $y_i > y_j$. Let us concentrate on the queries of the first type. We defined the array Y, which was the order of the indices of points of P with respect to increasing y-coordinate. We compute a data-structure over Y, i.e., a complete binary tree T, where nodes of T also contain appropriate suffix minima and prefix maxima arrays. Given a rectangle query (p_i, p_j), where $i < j$ and $y_i < y_j$, we find the position $a = pos_i$ and $b = pos_j$ in Y of i and j, respectively. Now determine the lowest common ancestor node of a and b, say u in T. Locate the position of $Y[a]$ and $Y[b]$ among the children of u in T and then using the suffix minima and prefix maxima informations computed in T, answer the query. Since finding the lowest common ancestor in a complete binary tree and locating the appropriate $Y[a]$ and $Y[b]$ requires constant time, the queries can be answered in $O(1)$ time. We summarize the results in the following theorem.

Theorem 4.3 *A data structure of size $O(n \log n)$ can be computed in $O(n \log n)$ sequential time and in $O(\log n)$ parallel time using $O(n)$ EREW PRAM processors, rectangle queries can be answered in $O(1)$ sequential time.* ■

5 Conclusion

We have introduced the archer's problem and shown that its solution leads to the intersting class of stage graphs which we characterized to be permutation graphs. The characterization which leads to the solution for the archer's problem allowed for the development of improved algorithms for matching in permutation graphs, for a class of two-processor scheduling problems, and for several geometric problems.

There are several interesting open problems suggested by our investigations. Can the upper bound on the matching be improved? Can the space required by the dominance algorithm be reduced to linear?

References

[1] M. Atallah, R. Cole and M. Goodrich. "Cascading divide-and-conquer: a technique for designing parallel algorithms." SIAM J. Comp. 18 (1989), 499-532.

[2] B. Bollobás, Extremal Graph Theory, Academic Press, 1978.

[3] Coffman, E. G. Jr.; Graham, R. L.; "Optimal scheduling for two-processor systems" Acta Informatica 1 (1972), 200-213.

[4] R. Cole, "Parallel Merge Sort", SIAM J. Comp. 17:4 (1988), 770-785.

[5] A. Datta, A. Maheshwari, J.-R. Sack, "Optimal parallel algorithms for direct dominance problems", Proc. of the First Annual European Symposium on Algorithms, LNCS 726 (1993), 109-120.

[6] B. Dushnik and E. Miller, "Partially ordered sets", Amer. Math. Monthly 55 (1948), 26-28.

[7] Fujii, M.; Kasami, T.; Ninomiya, K. "Optimal sequencing of two equivalent processors." SIAM J. Appl. Math. 17:4 (1969), 784-789.

[8] Gabow, H. N. "An almost-linear algorithm for two-processor scheduling" J. ACM 29:3 (1982), 766-780.

[9] Gabow, H. N.; Tarjan, R. E. "A linear-time algorithm for a special case of disjoint set union" J. Comp. and System Sci. 30 (1985), 209-221.

[10] M. T. Goodrich, "Intersecting line segments in parallel with an output-sensitive number of processors", SIAM J. Comp., 20 (1991), 737-755.

[11] J. JáJá, An introduction to parallel algorithms, Addison-Weseley Publishing Company, 1992.

[12] R. M. Karp and R. Vijaya Ramachandran, "Parallel algorithms for shared-memory machines", Handbook of Theoretical Computer Science, Edited by J. van Leeuwen, Volume 1, Elsevier Science Publishers B.V., 1990.

[13] A. Lingas and A. Maheshwari, "Simple optimal parallel algorithm for reporting paths in trees", Symposium on Theoretical Aspects of Computer Science, LNCS, 1994.

[14] R.M. McConnell and J.P. Spinrad, "Linear-time modular decomposition and efficient transitive orientation of comparability graphs", ACM-SIAM Symposium on Discrete Algorithms (1994), 536-545.

[15] S. Micali and V. V. Vazirani, "An $O(\sqrt{V}E)$ algorithm for finding maximum matching in general graphs", in: Proc. 21st Ann. IEEE Symp. Foundations of Computer Science, (1980),17-27.

[16] Moitra, A.; Johnson, R., C. "A Parallel algorithm for maximum matching on interval graphs." 18th Intl. Conf. on Parallel Processing III (1989), 114-120.

[17] A. Pnueli, S. Even and A. Lempel, "Transitive orientation of graphs and identification of permutation graphs", Canad. J. Math 23 (1971), 160-175.

[18] F. P. Preparata and M. I. Shamos, Computational Geometry: An Introduction, Springer-Verlag, New York, 1985.

[19] Sehti, R. "Scheduling graphs on two processors" SIAM J. Comp. 5:1 (1976), 73-82.

[20] J. Spinrad, "On comparability and permutation graphs", SIAM J. Comp. 14 (1985), 658-670.