A General Approach to Dominance in the Plane

Mark T. de Berg, Svante Carlsson and Mark H. Overmars

RUU-CS-88-35
November 1988

Rijksuniversiteit Utrecht

Vakgroep informatica

Padualaan 14 3584 CH Utrecht
Corr. adres: Postbus 80.089, 3508 TB Utrecht
Telefoon 030-531454
The Netherlands

A General Approach to Dominance in the Plane

Mark de Berg, Svante Carlsson and Mark Overmars

Technical Report RUU-CS-88-35
November 1988

Department of Computer Science
University of Utrecht
P.O. Box 80.089
3508 TB Utrecht
the Netherlands

A General Approach to Dominance in the Plane

Mark T. de Berg* Svante Carlsson† Mark H. Overmars‡

November 30, 1988

Abstract

Given two points p and q and a set of points O in the plane, p is said to dominate q with respect to O if p dominates q and there is no $o \in O$ such that p dominates o and o dominates q. In other words, O is a set of obstacles that might block the "rectangular view" from p to q. Given sets P and O we are interested in determining all pairs $(p, q) \in P \times P$ such that p dominates q with respect to O. This generalizes notions of direct dominance and rectangular visibility that have been studied before. An algorithm is presented that solves the problem in optimal time $O(n \log n + k)$, where n is the size of $P \cup O$ and k is the number of answers. A second problem asks to store the sets P and O such that queries of the form "given a query point q, compute all points p in P, such that q dominates p with respect to O" can be answered efficiently. A static structure is devised with a query time of $O(\log n + k)$ using $O(n \log^2 n)$ storage. Using a different approach, we devise a fully dynamic structure in which queries cost $O(\log^2 n + k)$ time. This structure uses $O(n \log n)$ storage and updating P or O can be done in $O(\log^2 n)$ and $O(\log^3 n)$ respectively. Finally the notion of dominance with respect to obstacles is extended to obstacle sets that may contain arbitrary objects.

1 Introduction

In d–dimensional space the notion of dominance is defined as follows: a point $p = (p_1, p_2, \ldots, p_d)$ is said to *dominate* a point $q = (q_1, q_2, \ldots, q_d)$, denoted as $p \succ q$, iff $p_i \geq q_i$ for all $1 \leq i \leq d$ and $p \neq q$. Furthermore, given a set P of points, the notion of direct dominance (as introduced in [4]) is defined: a point p *directly dominates* a

*Dept. of Computer Science, University of Utrecht, Utrecht, The Netherlands. This author was partially supported by the Dutch Organisation for Scientific Research (N.W.O.).

†Dept. of Computer Science, Lund University, Lund, Sweden. This author was supported by NFR grant 8992-100 (Sweden).

‡Dept. of Computer Science, University of Utrecht, Utrecht, The Netherlands

point q, denoted as $p \not\succ q$, iff $p \succ q$ and there is no point r in P, such that $p \succ r$ and $r \succ q$.

In this paper we generalize this notion in the following way: we are not only given a set of points P, but also a set O of so-called obstacle points.

Definition 1 *Let O be a set of points in d-dimensional space, and let $p = (p_1, \ldots, p_d)$ and $q = (q_1, \ldots, q_d)$ be two points in E^d. Now p is said to* dominate q with respect to O, *denoted as $p \succ_O q$, iff $p \succ q$ and there is no point $o \in O$ such that $p \succ o$ and $o \succ q$.*

In other words, the rectangle with p and q as opposite vertices does not contain any obstacle point in O. (Note that it can contain other points from P.) Overmars and Wood[12] (see also [8]) consider this as a special kind of visibility, called *rectangular visibility*, but they only considered the case $P = O$.

A pair $(p, q) \in P \times P$ such that $p \succ_O q$ is called a *dominance pair in P with respect to O*, or a *dominance pair* for short. Note that for two points $p, q \in P$ $p \succ_\emptyset q$ is equivalent to $p \succ q$ and $p \succ_P q$ is equivalent to $p \not\succ q$; thus our new notion captures the notions of (direct) dominance.

In the sequel of this paper we will restrict our attention to the case where $d = 2$. From now on n will denote $|P| + |O|$ and k the number of answers. The paper is organized as follows.

In section 2 the All Pairs Problem is considered: Given a set P of points and a set O of obstacle points, determine all dominance pairs in P with respect to O. This is thus a generalization of the All Pairs Problem considered in [8, 12] and the direct dominance problem considered by Güting et al. in [4]. An algorithm is presented that solves this problem in optimal time $O(n \log n + k)$. This immediately implies the known bounds for direct dominance and rectangular visibility in [4, 8, 12]. The method is an extension of [4] and uses a combination of divide-and-conquer and plane-sweep.

In section 3 we look at the Query Problem: We want to structure the points in P and the obstacle points in O such that, given a query point q, all points in P that are dominated by q with respect to O can be determined efficiently. This problem has also been studied in [8, 12] for the case where $P = O$. First we will follow a similar approach as in the All Pairs Problem and devise a static structure with a query time of $O(\log n + k)$. This structure uses $O(n \log^2 n)$ storage. Taking a different approach leads to a fully dynamic structure, using $O(n \log n)$ storage, with a query time of $O(\log^2 n + k)$. Updating P and O can now be done in $O(\log^2 n)$ and $O(\log^3 n)$ respectively. Again the results of [8, 12] follow from our more general results.

In section 4 we generalize the notion of dominance with respect to obstacles to the situation in which the obstacle set O contains obstacles of arbitrary size and shape. It will be shown that O can be reduced to a set O' of points only of size $O(n)$. Then we can apply the results of the previous sections. For a large class of obstacles this will give good results to both problems.

Finally, in section 5 we briefly summarize our results and indicate some directions for further research.

2 The All Pairs Problem

Given two sets $P = \{p_1, \ldots, p_{n_p}\}$ and $O = \{o_1, \ldots, o_{n_o}\}$ of points in the plane, we are interested in computing all dominance pairs in P with respect to O.

To this end we will develop a divide-and-conquer method. Let V be the set of different x-coordinates of the points in $P \cup O$. If $|V| = 1$, the problem becomes 1-dimensional and is easy to solve: we sort the points in P and O according to y-coordinate and report all pairs of points in P that have no point of O between them in this sorted list. In this way all dominance pairs are reported in time $O(n \log n + k)$.

When $|V| > 1$, let $x_{mid} \notin V$ be such that the number of values in V that are $< x_{mid}$ and the number of values $> x_{mid}$ differ by at most 1. Let l be the vertical line with x-coordinate x_{mid}. l divides the plane in two halves and, hence, splits P and O in two halves. It is obvious that whether or not two points in one half of the plane form a dominance pair cannot be influenced by an obstacle point from the other half. So we can recursively treat the two halves in the same way. After this it remains to compute the dominance pairs between the points in different halves. This merge step will be described below. We thus arrive at the following algorithm:

1. Let V be the set of different x-coordinates in $P \cup O$ and let n' be the size of V.

2. If $n' = 1$ solve the problem as a 1-dimensional problem as described above.

3. Otherwise, let $x_{mid} \notin V$ be chosen such that the number of elements in V that are $< x_{mid}$ and the number of elements in $V > x_{mid}$ are both $\leq \lceil \frac{1}{2} n' \rceil$. Split P into $P_1 = \{p \in P | p_x < x_{mid}\}$ and $P_2 = \{p \in P | p_x > x_{mid}\}$. Split O into $O_1 = \{o \in O | o_x < x_{mid}\}$ and $O_2 = \{o \in O | o_x > x_{mid}\}$.

4. Report all dominance pairs in P_1 with respect to O_1 and in P_2 with respect to O_2 recursively in the same way.

5. Report all dominance pairs $(p, q) \in P_2 \times P_1$ with respect to O.

Step 1 of the algorithm can be performed in time $O(n)$ if we have P, O and V sorted by x-coordinate. After presorting on x-coordinate, which requires time $O(n \log n)$ these sets can be maintained sorted during the recursive calls.

When the recursion stops (in step 2) we have to sort the, say, n_i points that are in the sets by y-coordinate. This has to be done n' times, but, since we have $\sum_{i=1}^{n'} n_i = n$, the total time required for step 2 will be bounded by $O(n \log n)$ plus $O(1)$ time for every answer found.

Now let $T(n', n)$ be the time needed for the algorithm, then we have:

$$\begin{aligned} T(n', n) &= O(n \log n + k) + T'(n', n) \\ T'(n', n) &= T'(\lceil \tfrac{1}{2} n' \rceil, n - l) + T'(\lfloor \tfrac{1}{2} n' \rfloor, l) + O(n) + f(n) \end{aligned} \tag{1}$$

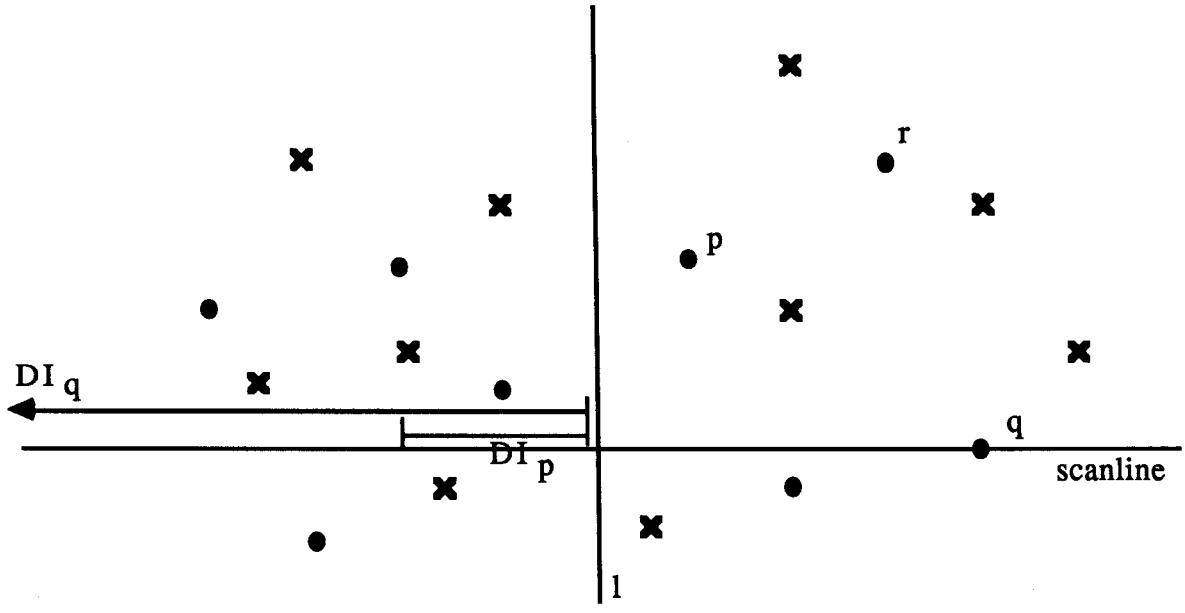

Figure 1: Some dominance intervals. $DI_r = \emptyset$. Points in P are indicated with a dot, points in O with a cross.

where k is the number of dominance pairs, $0 \leq l \leq n$ and $f(n)$ is the time needed to perform the merge step (step 5). Assuming that $f(n)$ is non-decreasing and at least linear this leads to

$$T(n', n) = O(n \log n + k + \log n'(n + f(n)) = O(f(n) \log n + k) \qquad (2)$$

because $n' = O(n)$.

It remains to be shown how the merge step (step 5) can be performed efficiently. First note that we only have to look at pairs $(p, q) \in P_2 \times P_1$ since all points in P_2 have larger x-coordinate than the points in P_1. So $p \succ q$ iff $p_y \geq q_y$.

The idea is as follows: We move a scanline downward over the plane, halting at every point in $P \cup O$. When we encounter a point $q \in P_1$, we will report all pairs $(p, q) \in P_2 \times P_1$ with $p \succ_O q$. We know that p must lie above (or on) the scanline. To find these points, for every point $p \in P_2$ above the scanline we keep track of a so-called *dominance interval* DI_p at the current position y^* of the scan line. This dominance interval consists of all x-values $< x_{mid}$ such that $x \in DI_p \Leftrightarrow p \succ_O (x, y^*)$. In other words, when the x-coordinate of a point $q \in P_1$ on the scanline lies in DI_p then $p \succ_O q$. Note that DI_p is indeed always an interval, which is of the form $]o_x : x_{mid}[$ where o_x is either $-\infty$ or the x-coordinate of some obstacle point (or DI_p is empty). See figure 1 for some examples of dominance intervals.

Suppose for the moment that no two points have the same y-coordinate. (If points do have the same y-coordinate we handle them from right to left. If a point $p \in P_2$ coincides with a point $o \in O$ then o is treated first. If a point $q \in P_1$ coincides with a point $o \in O$ then q is treated first. The reader can easily verify that this will be the correct order.) If we encounter a point $p \in P_2$ we must initialize

$DI_p :=]-\infty : x_{mid}[$. If we encounter a point $q \in P_1$, for all $p \in P_2$ with $q_x \in DI_p$ we report the pair (p, q) as a dominance pair. Obstacle points must be treated in the following way: An obstacle point $o = (o_x, o_y)$ dominates all the points to the left and below of it, so when we encounter o we have to change the dominance intervals for all points p that dominate o in the following way: if $o \in O_1$ then for all p with $o_x \in DI_p$ we must set $DI_p :=]o_x : x_{mid}[$ and if $o \in O_2$ then for all p with $o_x \leq p_x$ we must set $DI_p := \emptyset$.

Observe that after we have handled an obstacle point some of the dominance intervals become identical. To avoid changing all these intervals again at some later obstacle point we from now on treat them simultaneously. To this end we store identical DI's only once and associate a bag with it that contains all the points p for which $DI_p = DI$. This bag must allow for the following operations in constant time: inserting an element, deleting an element when we have a pointer to it and joining two bags. Moreover, all the elements should be enumerated in time the number of elements. This can be implemented e.g. as a doubly linked list.

To be able to handle obstacle points $o \in O_2$ efficiently, we must be able to determine all points in P_2 above the scanline that lie to the right of o and remove them. To this end we keep track, during the recursive calls, of so-called *highest dominated obstacles* which are defined as follows:

Definition 2 *Let O be a set of obstacle points. The* highest dominated obstacle in *O of some point p, denoted as $HDO_O(p)$, is the obstacle o with the property that it has the largest y-coordinate of all points in O that are dominated by p. If there is no such point then $HDO_O(p) = (p_x, -\infty)$.*

Now suppose that during the sweep we have for every point $o \in O_2$ a list of pointers to the points $p \in P_2$ such that $o = HDO_{O_2}(p)$. Thus we have a list of pointers to exactly those points whose domination intervals have to be set to $\emptyset$ (i.e. that have to be removed from the bag they are in) when we encounter o and o can be handled efficiently.

We now present the reporting of the dominance pairs in step 5 of the algorithm in more detail:

> Move a scanline downward over the set of points, halting at every point $(x, y) \in P \cup O$. (To do this we need a list of points $\in P \cup O$, sorted according to y-coordinate. This sorted list can be obtained from the sorted lists of $P_1 \cup O_1$ and $P_2 \cup O_2$ by a simple merge. This means that we only have to sort explicitly when the recursion halts. This sorting was already performed to compute the answers in step 2.) While we move the scanline we maintain the following data structure:
>
> A sorted list L of the different left endpoints of the dominance intervals of points in P_2 above the scanline. Every left endpoint has a bag associated with it, that contains all the points in P_2 that have that left

endpoint as the left endpoint of their dominance interval.

Furthermore we maintain pointers from the points in P_1 to the corresponding points in (the bags in) L and we have a pointer from every point in P_1 (P_2) to its highest dominated obstacle in O_1 (O_2) and vice versa. When we halt at a point (x, y) we have the following cases:

$(x, y) \in P_1$: Walk with x along L as long as the left endpoint of the current bag is $< x$ and report $(p, (x, y))$ as a dominance pair for each point p in these bags.

$(x, y) \in O_1$: Walk with x along L, joining all the bags with left endpoint $\leq x$ into a new bag with x as left endpoint.

$(x, y) \in P_2$: Add (x, y) to the bag in L with $-\infty$ as left endpoint (or, if necessary, create a new bag).

$(x, y) \in O_2$: Remove (using the cross pointers) all points that have this obstacle as their highest dominated obstacle from the bags in L. (If a bag becomes empty, then remove the bag and its corresponding DI from L.)

It remains to show how the highest dominated obstacles are maintained during the recursive calls, i.e. we must show how $HDO_O(p)$ can be computed for $p \in P$ from $HDO_{O_1}(p)$ for $p \in P_1$ and $HDO_{O_2}(p)$ for $p \in P_2$. For points in P_1 we simply have $HDO_O(p) = HDO_{O_1}(p)$. To find $HDO_O(p)$ for points in P_2 we perform a second (downward) sweep, this time halting at every point in $P_2 \cup O$. (This could, with a little care, also be done during the first sweep.) Now the highest dominated obstacle for a point $p \in P_2$ changes iff an obstacle in O_1 is encountered between p and $HDO_{O_2}(p)$. So we maintain a list of all points in P_2 above the scanline whose highest dominated obstacle in O_2 has not been encountered yet. When an obstacle in O_1 is encountered then this will be the new HDO for all points still in this list and the list is emptied. (Thus the list in fact contains those points in P_2 above the scanline whose new highest dominated obstacle has not been determined yet.) When an obstacle $o \in O_2$ is encountered all points p for which $o = HDO_{O_2}(p)$ and that are still in the list are removed from the list: their highest dominated obstacles does not change. This can easily be done using the pointers already available.

The above leads to:

Lemma 1 *The merge step can be performed in time $O(n + k)$.*

Proof: The correctness of the algorithm follows from the above discussion.

To analyse the time complexity we have to look at the four different cases in the first sweep:

$(x, y) \in P_1$: We spend $O(1 + \#\text{answers})$ time.

$(x, y) \in O_1$: We spend $O(1 + (\#(\text{bags with left endpoint} \leq x)\text{-}1))$ time. We charge the costs for joining the bags as follows to points in P_2: Let q_b be the first point that is put in bag b. Then we charge the costs of joining bags $b_1, \ldots, b_s$ into bag b_s to the points $q_{b_1}, \ldots, q_{b_{s-1}}$. It is clear that a point can be charged costs only once this way. Thus every point $\in P_2$ gets an extra $O(1)$ at most. The remaining $O(1)$ time of step (ii) is simply charged to (x, y).

$(x, y) \in P_2$: We spend $O(1)$ time to insert the point in the bag.

$(x, y) \in O_2$: We spend $O(\#\text{deletions})$ time for deleting the points from the bags. We charge these costs to the deleted points. Since a point can be deleted only once, every point $\in P_2$ is charged with this extra $O(1)$ at most once.

In the second sweep clearly only a constant amount of time is spent at at every encountered point. The total time bound follows immediately. (Note that no sorting is required because the points were already presorted. We only have to merge the lists which requires time $O(n)$.) □

So we have $f(n) = O(n)$ in equation 2. This leads to the following result:

Theorem 1 *All dominance pairs in a set of points P with respect to a set of points O can be computed in time $O(n \log n + k)$, where $n = |P| + |O|$ and k is the number of answers.*

3 The Query Problem

The Query Problem is stated as follows: structure a set P of points and a set O of obstacle points in such a way that, given a query point q, all points in P that are dominated by q with respect to O can be determinated efficiently. (We call such a query a *domination query.*) First we will let ourselves be inspired by the solution to the All Pairs Problem presented in the previous section. This will lead to a static structure with a query time of $O(\log n + k)$. Then a different approach is taken to obtain a fully dynamic structure with a query time of $O(\log^2 n + k)$.

3.1 A static structure

Together with the notion of highest dominated obstacle, defined in the previous section, *lowest dominating obstacles* will play a crucial role in the development of a static data structure. They are defined as follows:

Definition 3 *Let O be a set of obstacle points. The* lowest dominating obstacle in *O of some point p, denoted as $LDO_O(p)$, is the obstacle o with the property that it has the smallest y-coordinate of all points in O that dominate p. If there is no such point then $LDO_O(p) = (p_x, \infty)$.*

Since only points to the left of the query point q are of importance (the points to the right cannot be dominated and the obstacles to the right cannot influence any dominations), our main structure will be a binary tree T on the x–coordinates of the points in $P \cup O$. If P_δ (O_δ) denotes the subset of P (O) stored in the subtree of T rooted at δ then the points that are to the left of q are exactly the points in $\bigcup_\delta P_\delta \cup O_\delta$ for nodes δ that are left son of a node on the path to q_x but not on that path themselves.

The following lemma, the validity of which the reader can easily verify, identifies which of the points in such a P_δ are answers:

Lemma 2 *Let P, O_1 and O_2 be sets and q a point such that all points in $P \cup O_1$ are to the left of those in $O_2 \cup \{q\}$. Then, for $p \in P$, we have that $q \succ_{O_1 \cup O_2} p$ iff:*
(i) $q_y \in [p_y : (LDO_{O_1}(p))_y[$ and
(ii) $p_y > (HDO_{O_2}(q))_y$

With this lemma in mind we augment every node δ in the main tree T with the following information:

- A segment tree (see, e.g.,[1, 7]) ST_δ containing the segments $[p_y : (LDO_{O_\delta}(p))_y[$ for the points p in P_δ. The segments at a node β in ST_δ are ordered on their left endpoints. (This ordering is necessary to be able to report the segments satisfying condition (ii) of the lemma in time $O(1+\sharp\text{segments})$ at a node whose segments satisfy condition (i).)

- A binary tree OT_δ on (the y–coordinates of) the points in O_δ. (This tree is used to be able to keep track of the highest dominated obstacle for q encountered so far, which is needed to check condition (ii).)

Now the points in P dominated with respect to O by q can be found with the following algorithm (in this algorithm HDO_{curr} will hold the y–coordinate of the highest dominated obstacle encountered so far):

1. Initialize $HDO_{curr} := -\infty$. Search with q_x in T. Let γ be the leaf where the search path ends.

2. If the point r that γ contains lies to the left of q then if $r \in P$ then report r and if $r \in O$ then set $HDO_{curr} := r_y$.

3. Walk back along the search path. At every node δ do the following:
   If the search path at δ went to the left then do nothing. If the path went to the right then search in $ST_{lson(\delta)}$ for the segments that contain q_y and whose left endpoint is greater than HDO_{curr}. Report the points corresponding to those segments. Furthermore search in that case in $OT_{lson(\delta)}$ for the highest obstacle o below q and set $HDO_{curr} := \max(HDO_{curr}, o_y)$.

This leads to:

Lemma 3 *A structure exists that uses $O(n \log^2 n)$ storage and in which domination queries can be answered in time $O(\log^2 n + k)$, where k is the number of answers found. This structure has a preprocessing time of $O(n \log^2 n)$.*

Proof: The bounds on the storage and the preprocessing time follow readily from the $O(n \log n)$ bounds on both storage and preprocessing time of a segment tree on n segments. The query time follows from the fact that we spend $O(\log n + k)$ at every node on the search path to q_x. This is true since we have the segments at a node in the segment tree ordered according to their left endpoint.

The correctness of the algorithm can be seen as follows. When we arrive at some node δ in T we have already encountered all the nodes lower on the search path and thus all the points which lie in between (with respect to their x–coordinates) the points in $P_\delta \cup O_\delta$ and point q. This means that HDO_{curr} is the y–coordinate of the highest dominated obstacle for q of all obstacle points to the right of the points in $P_{lson(\delta)}$. Hence by Lemma 2 the correct answers are reported. □

We will now reduce the query time to $O(\log n + k)$. To achieve this some extra information has to be added to the segment trees ST_δ such that, if the leaf γ where the search with q_y ends in some ST_δ is known, the answers in this ST_δ can be enumerated in time $O(1 + \sharp\text{answers})$. Recall that a segment provided an answer if it contained p_y and had a left endpoint that was greater than HDO_{curr}. To be able to enumerate the answers at one particular node in an ST_δ in $O(1 + \sharp\text{answers})$ time the segments were stored at the nodes ordered according to their left endpoint. Now let for a node β in an ST_δ $maxleft(\beta)$ denote the maximum of the left endpoints of the segments stored at β (if there are no segments stored at β then $maxleft(\beta) = -\infty$). Now consider two nodes β_1 and β_2 on the search path to q_y and suppose $maxleft(\beta_1) > maxleft(\beta_2)$. Then if there are no answers reported at node β_1 then node β_2 cannot contain any answers either. So what we do is the following. At every leaf in an ST_δ we have a list of pointers to the nodes β on the path to this leaf that is sorted on the $maxleft(\beta)$ values. (Note that a segment tree already costs $O(n \log n)$ storage, so this extra information does not increase the bound on the storage.) This way the answers in an ST_δ can be enumerated in time $O(1 + \sharp\text{answers})$ by walking along this list and visiting the nodes β as long as answers are found. Now if we apply fractional cascading ([3]) to the main tree T with its associated structures ST_δ and OT_δ, then the leaves in the ST_δ's as well as the leaves in the OT_δ's where the search path to q_y ends can be found in constant time, thus giving a query time as desired.

The preprocessing time for this structure is still $O(n \log^2 n)$: with some care the extra lists stored at the leaves of a segment trees can be build in $O(n \log n)$ time per segment tree and applying fractional cascading costs only linear time. Hence we have:

Theorem 2 *A static structure exists that solves the Query Problem with a query time of $O(\log n + k)$. It uses $O(n \log^2 n)$ storage and has a preprocessing time of $O(n \log^2 n)$.*

Remark: It is possible to make this structure dynamic with respect to P with an update time of $O(\log^2 n)$ by using a different structure (instead of the segment trees) as associated structure. The conditions of Lemma 2 can namely also be stated as $(p_y, (LDO_{O_1}(p))_y) \in](HDO_{O_2}(q))_y : q_y] \times [q_y : \infty]$. Hence we could use a structure for range queries as associated structure and, because the second range is half-infinite, a priority search tree ([6]) suffices. This not only reduces the update time of the associated structure to $O(\log n)$, but it also uses only linear space. This would result in a structure with a query time of $O(\log^2 n)$ and an update time (for P only) of $O(\log^2 n)$, using $O(n \log n)$ storage.

3.2 A dynamic structure

In the previous section a static structure was presented that solved the Query Problem. Because of its use of lowest dominating obstacles it was inherently static with respect to the set of obstacles. In this section a fully dynamic structure is devised. The key notion in the development of the structure is that of *maximal elements* (a point p is *maximal with respect to* O iff there is no obstacle point in O that dominates p.) Munro et al. namely prove the following ([8]):

Lemma 4 *Let P and O be two sets of points in the plane. Let $q = (q_x, q_y)$ and let $P' = \{p \in P | p_x \leq q_x, p_y \leq q_y$ and $p \neq q\}$ and $O' = \{o \in P | o_x \leq q_x, o_y \leq q_y$ and $o \neq q\}$. Then the points in P that are dominated by q with respect to O are precisely the points in P' that are maximal with respect to O'.*

Remark: In fact they proved the lemma only for the case $P = O$. It can easily be seen, however, that the lemma also holds in our more general case.

3.2.1 Maximal elements queries

We will first devise a structure that stores the points in $P \cup O$ in such a way that, given two vertical lines $l : x = l_x$ and $r : x = r_x$, the points in $P_{l,r} = \{p \in P | l_x < p_x \leq r_x\}$ that are maximal with respect to $O_{l,r} = \{o \in O | l_x < o_x \leq r_x\}$ can be determined. (We will refer to this as a *maximal elements query*, as opposed to a domination query, when confusion might arise.) Then, with the above lemma in mind, we use this structure to solve the domination query problem.

The points in $P \cup O$ are stored in the leaves of a binary tree T, sorted on (increasing) x–coordinate. For a node δ in T, let T_δ denote the subtree of T rooted at δ and let P_δ and O_δ denote those points in P and O respectively that are stored in T_δ. At a node δ in T we store the following information. First of all we have two values $XMAX_\delta$ and $YMAX_\delta$, which are the maximum of the x– respectively y–coordinates of the obstacle points in O_δ. If O_δ is empty then these values are set to $-\infty$. (The $XMAX_\delta$-values are not used yet. They will, however, be needed when this structure is used to devise a structure for domination queries.) Secondly, of all the points in P_δ that are maximal with respect to O_δ, we store at δ the point with

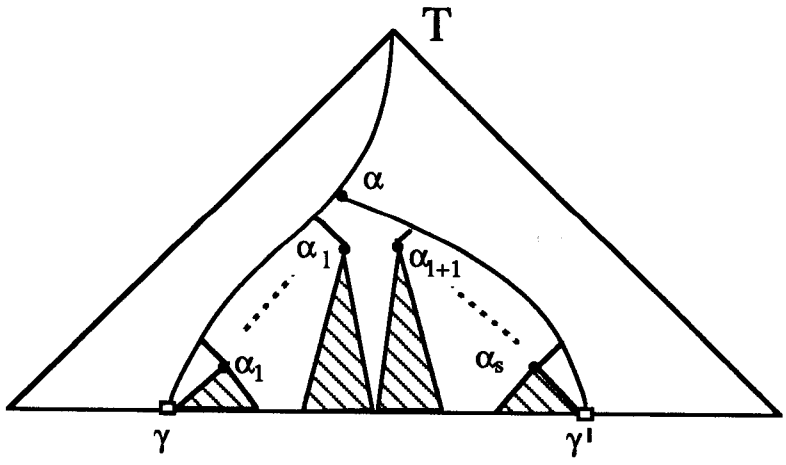

Figure 2: The leaves γ and γ' and the nodes α and $\alpha_1, \alpha_2, \ldots, \alpha_s$.

greatest y–coordinate that is not already stored somewhere higher in the tree. This point stored at node δ will be called p_δ. Note that p_δ can be empty for some nodes δ and that points can only be stored on their search paths. Thus we have some sort of priority queue on the y–coordinates of the points, implemented as a heap, with the restriction that a point can only be stored at nodes where it is maximal. (Thus the structure is related to the priority search tree of McCreight[6].)

Now given two vertical lines $l : x = l_x$ and $r : x = r_x$, the points in $P_{l,r}$ that are maximal with respect to $O_{l,r}$ can be determined as follows: Consider the leaves γ and γ' where the search paths to l_x and r_x end. Clearly the points with x–coordinate between l_x and r_x must be stored in the leaves between (possibly including) γ and γ'. Let α be the node where the search paths to l_x and r_x split and let $\alpha_1, \alpha_2, \ldots, \alpha_s$ be the nodes that are either right son of a node on the path from α to γ but not on that path itself, or left son of a node on the path from α to γ' but not on that path itself (see Figure 2). Now the fact that a point can only be stored somewhere on its search path means that the points in $P_{l,r}$ must be stored either on one of the paths to γ and γ' or in one of the subtrees T_{α_i}. The idea is to examine the points from right to left, thereby keeping track of the highest obstacle point encountered so far (since we go from right to left this obstacle point will namely dominate all points with smaller y–coordinates encountered later) and reporting all the points that we come across that have greater y–coordinate than this obstacle point.

We will now describe this algorithm in more detail:

1. Search with l_x and r_x in T until the search paths split. At every node δ on the path to (and including) this splitting node α test if p_δ exists and if so, report this point if $l_x < (p_\delta)_x \leq r_x$. (From now on we will omit the test whether or not some p_δ exists in the description of the algorithm for reasons of brevity.)

2. Search from α with r_x. Let γ' be the leaf where the search ends. If $\gamma'_x \leq r_x$ then report $p_{\gamma'}$ and set $YMAX := YMAX_{\gamma'}$ otherwise set $YMAX := -\infty$.

3. Walk back from γ' to α. At every node δ do:
   (i) If $(p_\delta)_x \leq r_x$ then report p_δ.
   (ii) If we came from the right then REPORT($lson(\delta)$,$YMAX$) and set $YMAX :=$ max($YMAX, YMAX_{lson(\delta)}$).

4. Search from α with l_x. At every node δ do:
   (i) If $(p_\delta)_x > l_x$ then report p_δ if $(p_\delta)_y > YMAX$.
   (ii) If we go to the left then REPORT($rson(\delta)$,$YMAX$) and set $YMAX :=$ max($YMAX, YMAX_{rson(\delta)}$).

5. At the leaf γ where the search ends check to see if p_γ has x–coordinate greater than l_x and y–coordinate greater than $YMAX$ and report p_γ if so.

The procedure REPORT is used to report all the answers in a subtree T_{α_i} (see Figure 2). It looks as follows:

```
procedure REPORT(δ,YMAX)
begin
  if p_δ exists
  then if (p_δ)_y > YMAX
       then report p_δ as an answer
            REPORT(rson(δ),YMAX)
            REPORT(lson(δ),max(YMAX, YMAX_rson(δ)))
end
```

Lemma 5 *The algorithm described above reports a point if and only if it is a point in $P_{l,r}$ that is maximal with respect to $O_{l,r}$.*

Proof: $\Longrightarrow$: If p is reported in step 1 of the algorithm then p is explicitly tested to be in $P_{l,r}$. Furthermore a point can be stored only at nodes where it is maximal; since for a node δ on the path to α we have $O_{l,r} \subseteq O_\delta$ this implies that a point stored at such a node δ is necessarily maximal with respect to $O_{l,r}$.

If point $p_{\gamma'}$ is reported in step 2 then $(p_{\gamma'})_x \leq r_x$ is tested. $(p_{\gamma'})_x > l_x$ follows from the fact that the search paths have already split and maximality follows from the fact that $p_{\gamma'}$ is necessarily the rightmost point in $P_{l,r} \cup O_{l,r}$ and thus cannot be dominated by a point in $O_{l,r}$.

As for the points p_δ reported in step 3(i) we note that $p_\delta \in P_{l,r}$ again follows from the test $(p_\delta)_x \leq r_x$ and the fact that the search paths have already split. Maximality follows from p_δ being maximal at δ and the fact that O_δ must contain all obstacle points in $O_{l,r}$ that lie to the right of p_δ.

To prove maximality for the points reported in step 3(ii) we note that at a node δ we have $YMAX = \max_{o \in O_{l,r}}\{o_y | o_x > \delta_x\}$, since we keep track of the maximum of the y–coordinates of the points encountered so far and we encounter them in a right to left order. Therefore if we call REPORT($lson(\delta), YMAX$) from a node δ on the path from γ' to α, $YMAX$ is the maximum of the y–coordinates of the obstacle points in $O_{l,r}$ that are stored in $T_{rson(\delta)}$. From the second test in REPORT and the fact that points are only stored at nodes at which they are maximal, maximality of

the reported points follows. It is easy to see that the $YMAX$–values are correctly maintained during the recursive calls so that only maximal points are reported. The points reported in this step are in $P_{l,r}$ since they lie in between the search paths (see Figure 2).

For the points reported in step 4 and 5 analogous considerations hold, thus proving the first part of the lemma.

$\Longleftarrow$: Let $p \in P_{l,r}$ be maximal with respect to $O_{l,r}$. We will prove that p is reported. Since the leaves γ and γ' are explicitly tested we can assume that p lies somewhere in between, say $p \in P_{\alpha_j}$ ($\alpha_1, \ldots, \alpha_s$ being defined as in Figure 2). Note that p is maximal with respect to O_{α_j} since $O_{\alpha_j} \subseteq O_{l,r}$. Now $p \in P_{\alpha_j}$ does not necessarily mean that p is stored somewhere in T_{α_j}; p can be stored in T_{α_j}, but p can also be stored higher in the tree. In the second case p must be stored on the search path to α_j which is the same as (a part of) the search path to l_x or r_x. Hence p must be checked in step 1, 3(i) or 4(i) and be reported there. This leaves the first case, namely when p is stored somewhere in T_{α_j}. Note that in some stage of the algorithm REPORT$(\alpha_j, YMAX)$ must be called and that if the point that is tested in REPORT is maximal with respect to $O_{l,r}$, it is reported and REPORT goes into recursion. This means that if all the nodes from α_j to the node δ^* where p is stored are maximal with respect to $O_{l,r}$, then p must be reached and thus reported. Now let δ be such a node ($\delta \neq \alpha_j$) and assume that δ contains a point p_δ that is maximal with respect to $O_{l,r}$. We will show that in that case $father(\delta)$ must also contain a point that is maximal. Together with δ^* containing point p (which was maximal) this implies that all the nodes from δ^* to α_j must contain maximal points and thus that p is reported. Now $p_{father(\delta)}$ (which exists since otherwise p_δ would be stored there, because p_δ is reported and thus necessarily maximal at $father(\delta)$) must be maximal for the following reason: Since $(p_{father(\delta)})_y > (p_\delta)_y$ (because they are both maximal at $father(\delta)$ but $p_{father(\delta)}$ is stored there) the situation when an obstacle point o dominates $p_{father(\delta)}$ but not p_δ must be as follows: $p_{father(\delta)}$ must lie to the left of p_δ and o must lie between them. This, however, would mean that $o \in O_\delta$, thus contradicting the fact that $p_{father(\delta)}$ is maximal at δ. This proves the second part of the lemma. □

Lemma 6 *The structure described above costs $O(n)$ storage, can be built in time $O(n \log n)$ and a maximal elements query takes time $O(\log n + k)$, where $n = |P| + |O|$ and k is the number of maximal points in $P_{l,r}$ with respect to $O_{l,r}$.*

Proof: The bound on the storage is evident. Building the structure in $O(n \log n)$ can be done as follows: First we build a skeleton tree T on the points in O in time $O(n \log n)$. For every node δ in T we set $p_\delta := \emptyset$, $XMAX := -\infty$ and $YMAX := -\infty$. Then we insert the obstacle points into T. Since there are no points of P present yet, we only have to adjust the $XMAX$– and $YMAX$–values on the search path to an obstacle point if necessary. This clearly can be done in $O(\log n)$ per point. Then the points in P are inserted. As will be shown in Lemma 7, this can also be done in $O(\log n)$ time per point, thus giving a total building time as stated.

Since the correctness of the query algorithm already has been proved, it remains to prove the $O(\log n + k)$ time bound of the algorithm. Because REPORT in fact traverses a (sub)tree and goes into recursion only if an answer is reported, the number of nodes visited in a call to *REPORT* is equal to twice the number of answers found plus one. Hence REPORT takes time $O(1 + \sharp \text{ answers})$ from which the total query time easily follows. □

We will now turn our attention to the dynamic behaviour of the structure. First we consider updates in P. Assuming the point p to be inserted into the stucture is not already present in the structure, the insert algorithm is as follows:

1. Search with p_x in T and create a new leaf for p.
2. Walk back along the nodes δ on the search path as long as p is maximal at δ and $p_y > (p_\delta)_y$. (The maximality of p at a certain node δ can be tested as follows: Since we stop as soon as p is not maximal anymore we know that p is maximal at the son of δ from which we came. If we came from the right this automatically implies that p is maximal at δ. If we came from the left then p is maximal at δ iff $p_y > YMAX_{rson(\delta)}$.) Let δ^* be the last such node.
3. IRESTORE(δ^*, p).

The procedure IRESTORE stores point p in p_{δ^*} and restores tree T_{δ^*}:

```
procedure IRESTORE(δ, p)
begin
  if p_δ exists
  then if (p_δ)_x < δ_x
       then IRESTORE(lson(δ), p_δ)
       else IRESTORE(rson(δ), p_δ)
  p_δ := p
end
```

Deletions can be performed in an analogous way: Delete the leaf representing p from the structure, identify the node δ for which $p_\delta = p$ and delete p from p_δ using the procedure DRESTORE, which is very similar to IRESTORE (DRESTORE(δ) removes p_δ and then checks to see if one of the points stored at the sons of δ has to take the place of p_δ. If so this is done and DRESTORE goes into recursion in the son from which the point was taken.)

Lemma 7 *A point $p \in P$ can be inserted into and deleted from the structure in time $O(\log n)$.*

Proof: We only prove the lemma for insertions. Since deletions are performed in a similar way, the proof is also similar and left to the reader. It is clear that the insertion of p only influences the subtree T_{δ^*}, so it remains to show that T_{δ^*} is correctly rebuilt. Consider a call to IRESTORE(δ, p). We can assume that point p is maximal at δ and has greater y–coordinate than the other points in T_δ that are

maximal at δ. (This follows from the fact that p came from a higher node in the tree.) Thus p clearly has to be stored at δ. If p_δ was empty, then this restores T_δ, otherwise the point p_δ has to be moved down in the tree. So we go into recursion into the subtree of δ in which p_δ belongs. The other subtree is not afffected by the insertion of p at δ. By induction it follows that T_δ must be rebuilt correctly. (Note that at the latest at the leaf where IRESTORE ends, there must be a δ on the path that IRESTORE takes, such that p_δ is empty.) Step 1 and 2 of the algorithm cost $O(\log n)$ time and step 3 costs $O(height(T_{\delta^*})) = O(\log n)$. Hence the algorithm works in time $O(\log n)$.

We are left with the problem of keeping the structure balanced. This can be done using rotations (see, e.g., [5]). In a rotation, however, some nodes δ get 'new' subtrees and, hence, should perhaps also get new p_δ's. What we have to do to get the correct p_δ's at these nodes is remove these p_δ's temporarily, using DRESTORE, and then reinsert them again using IRESTORE. But DRESTORE and IRESTORE take $O(\log n)$ time and thus a rotation costs $O(\log n)$ time. So what we need is a binary search tree that can be kept balanced using only a constant number of rotations per update. AVL–trees have this property for insertions but not for deletions. It is interesting that a tree with this property for both insertions and deletions exists and that we can use this tree to keep the update time $O(\log n)$. This tree, which is called a HBB–tree (see [9]) can be kept in balance using only 2 (single) rotations per insertion and 3 rotations per deletion. Therefore, if we use a HBB–tree for our tree T, we can perform updates in P in the stated time. □

We now come to the updating of the set O of obstacle points. Inserting a point $o \in O$ into the structure can be done in the following way:

1. Search with o_x in T and adjust the XMAX– and YMAX–values on the search path. At the end of the search path create a new leaf for o.

2. Walk back along the search path. At every node δ do the following: If $o \succ p_\delta$ then delete p_δ from δ (using DRESTORE(δ)) and reinsert p_δ (in this case no leaf has to be created for p_δ during the insertion).

Deletions of obstacle points can be done in a similar way. Details are left to the reader.

Lemma 8 *A point $o \in O$ can be inserted into and deleted from the structure in time $O(\log^2 n)$.*

Proof: Consider the above algorithm for inserting a point $o \in O$ into the structure: Since DRESTORE and the reinsertion of a point both cost $O(\log n)$ time the algorithm takes $O(\log n)$ time at every node on the path to o_x and thus the algorithm works in time $O(\log^2 n)$. As for the correctness, note that the insertion of an obstacle point can only cause trees T_δ to be incorrect for nodes δ on the search path to o_x. If $T_{lson(\delta)}$ and $T_{rson(\delta)}$ are correct, which we can assume by induction, then the actions taken in step 2 of the algorithm restore T_δ. Balancing can again be done using rotations. The time bound follows. □

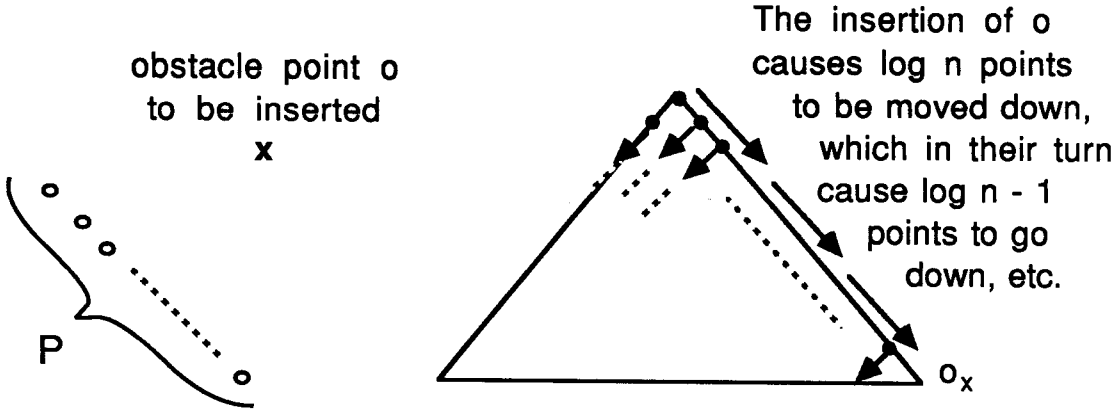

Figure 3: An insertion of an obstacle point that costs $O(\log^2 n)$ time.

Remark: $O(\log^2 n)$ is optimal for an update in O in this structure. Consider, for example, a tree on the point set P depicted in Figure 3, and suppose an obstacle point o has to be inserted that dominates all the points in P. The search path to o_x will end in the rightmost leaf. Now all the points p_δ for δ on this path are no longer maximal there and therefore have to be moved to the left. At these nodes $lson(\delta)$ they have to be inserted. But this means that $p_{lson(\delta)}$ has to be moved down, etc. We see that $\Omega(\sum_{i=1}^{\log n} i) = \Omega(\log^2 n)$ work has to be done.

3.2.2 Domination queries

Now that we have devised a structure for determining the maximal elements between two vertical lines, we return to the domination query problem. Recall from Lemma 4 that a point in P is dominated by a query point q iff it is maximal in the part of $P \cup O$ south-west of q. We store the points of $P \cup O$ as follows: We have a main tree T sorted on (decreasing) y–coordinate of the points in $P \cup O$. This tree will enable us to determine all the points to the south of the query point q. With every node β in T, we associate a structure T_β for maximal elements queries as described above, containing the points of $P_\beta \cup O_\beta$.

Given this structure, the maximal elements to the south-west of q can be determined as follows: First we determine the points with y–coordinate $\leq q_y$. These points are exactly the points stored in the trees T_β, for β a node that is right son of a node on the search path to q_y but not on the search path itself (plus, possibly, the point stored in the leaf where the search path ends). Number these nodes $\beta_1, \beta_2, \ldots, \beta_t$ according to the order in which their fathers are encountered when searching with q_y. This means that all points in T_{β_i} have smaller y–coordinates than every point in T_{β_j} if $i > j$. Consider such a structure T_{β_i}. In this structure, which contains only points to the south of q, we can first of all restrict ourselves to the points to the west

of q, thus to the left of the line $r : x = q_x$. Now a point p that is maximal to the left of this line r in T_{β_i} is maximal in the south-west quadrant of q (and thus dominated by q) iff there is no obstacle point stored in some other T_{β_j} that dominates it and that also lies in this quadrant. Such an obstacle would necessarily have greater y–coordinate than p, and thus be stored in a T_{β_j} with $j < i$. So what we do is the following: We search with q_y in T and do a maximal elements query in the structures T_{β_i}. Since we encounter these structures in such a way that, whenever we search in a T_{β_i}, the structures T_{β_j} with $j < i$ have already been encountered it is sufficient to keep track of $XMAX$, which is the maximum x–coordinate of obstacle points to the left of $r : x = q_x$ stored in the T_{β_j}'s encountered so far. Then a maximal elements query must be done with the lines $l : x = XMAX$ and $r : x = q_x$. Keeping track of $XMAX$ can be done using the $XMAX_\delta$–values stored at nodes δ in the T_{β_i}'s, while doing a maximal elements query in T_{β_i}. To be more specific, we have the following algorithm for domination queries:

1. Initialize $XMAX := -\infty$. Search with q_y in T. At every node β on the search path do: If the search path goes to the right then just continue. If the search path goes to the left then do a maximal elements query in $T_{rson(\beta)}$ with the lines $l : x = XMAX$ and $r : x = q_x$. Report the maximal elements thus found as being dominated by q and set $XMAX := \max(XMAX, \max_{o \in O_{rson(\beta)}}\{o_x | o_x \leq q_x\})$.

2. Check the leaf where the search path ends to see if it contains an answer and report if necessary.

Recall from Lemma 6, that a maximal elements query costs time $O(\log n + k)$. Hence the time needed for a domination query is $O(\log^2 n + k)$.

The storage the above structure uses is $O(n \log n)$ and it can be built in $O(n \log^2 n)$ time. Both bounds follow immediately from the corresponding bounds of the associated structures stated in Lemma 6.

Dynamization of the structure can be accomplished using the techniques described in [13]. In [13], Willard and Lueker describe how the rebuilding that is necessary when the main structure is out of balance can be done little by little during several updates, instead of all in one time. Their techniques lead to an update time of $O(U(n) \log n)$, where $U(n)$ denotes the time needed for an update in an associated structure, in our case $O(\log n)$ for an update in P and $O(\log^2 n)$ for an update in O. Hence the total update time becomes $O(\log^2 n)$ for points in P and $O(\log^3 n)$ for points in O. Summarizing the results of this section we have:

Theorem 3 *A structure exists that solves the domination query problem for a set P of points and a set O of obstacle points with a query time of $O(\log^2 n + k)$, where $n = |P| + |O|$ and k is the number of answers. This structure uses $O(n \log n)$ storage and has a preprocessing time of $O(n \log^2 n)$. Updates in P cost $O(\log^2 n)$ time and updates in O $O(\log^3 n)$.*

4 Treating general obstacles

We will now extend the notion of dominance in the presence of obstacles to obstacles of arbitrary size and shape. We will assume that the obstacles are connected, i.e. the connected parts of some object are considered to be separate obstacles.

Definition 4 *Let O be a set of objects in d-dimensional space, and let $p = (p_1, \ldots, p_d)$ and $q = (q_1, \ldots, q_d)$ be two points in E^d. Now p is said to* dominate q with respect to O, *denoted as $p \succ_O q$, iff $p \succ q$ and there is no point r on any obstacle $o \in O$ such that $p \succ r$ and $r \succ q$.*

We will show that it is possible to reduce O to a set O' of points only, such that the dominance pairs do not change. Then the results of the previous sections can be applied. The methods are described only briefly. Details for the All Pairs Problem can be found in [2].

Let's first look at the All Pairs Problem. As noted in the introduction, to say that p dominates q (with respect to the obstacles) is the same as to say that $Rect(p, q)$, the rectangle with p and q as opposite vertices, does not contain (a part of) any obstacle (provided that $p \succ q$, of course). Now suppose that $p \not\succ_O q$. Then there is an obstacle o with non-empty intersection with $Rect(p, q)$. This means that either $o \subseteq Rect(p, q)$, or $o \supseteq Rect(p, q)$ or that o intersects the boundary of $Rect(p, q)$ somewhere. We want to generate an obstacle set O' of points such that we also have $p \not\succ_{O'} q$. The first case, where $o \subseteq Rect(p, q)$, can be handled by adding an arbitrary point on any obstacle in O to O'. The third case is handled by computing for every point p in P the first intersection of the four axis-parallel rays from p with an obstacle in O and adding these intersection points to O'. The second case is the most dificult one. Points lying in the interior of an obstacle could just be removed from P, but this cannot be done for points lying on the boundary of an obstacle. For these points we do the following: we take two very small squares, one with p as south-west vertex and one with p north-east vertex. If such a square has non-empty intersection with the set of obstacles then we add some point of this intersection to O'. Now if the length ϵ of the sides of the squares is taken small enough (namely so small that that the vertical or horizontal open bars around p with width ϵ do not contain any other point of P) then this will solve the second case for all points q that have different coordinates from p. To handle points having some coordinate the same as p we also check if the sides of the square that are incident upon p have a non-empty intersection with the set of obstacles and, if so, add some point of this intersection to O'. Figure 4 shows what points are added to O'. The above discussion shows that $p \not\succ_O q \Longrightarrow p \not\succ_{O'} q$; $p \not\succ_O q \Longleftarrow p \not\succ_{O'} q$ follows directly from the fact that only points on obstacles are added to O'. Thus we have:

Lemma 9 *Let P be a set of points and let O be a set of arbitrary obstacles. Then a set O' of size $O(n)$ consisting of points only exists such that for $p, q \in P$:*
$p \succ_O q \Longleftrightarrow p \succ_{O'} q$.

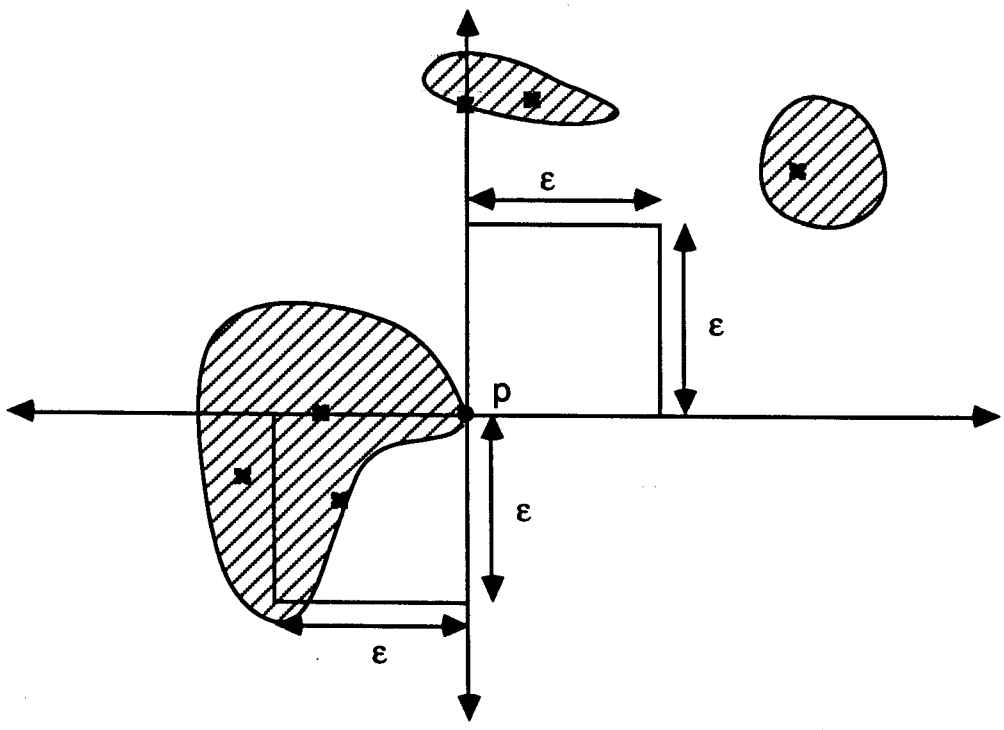

Figure 4: Points added to O'.

Of course, if we actually want to compute this set O' (efficiently) then the obstacles should satisfy some constraints. In [2] constraints are given that make an efficient calculation of O' possible. When, e.g., the obstacles are non-intersecting polygons of bounded degree the reduction can be performed in $O(n \log n)$ time. This leads to:

Theorem 4 *All dominance pairs in a set of points P with respect to a set O of simple, non-intersecting polygons of bounded degree can be computed in time $O(n \log n + k)$, where $n = |P| + |O|$ and k is the number of answers.*

The reduction for the Query Problem is performed in about the same way. When doing a query with some point q, however, we should calculate O' with respect to the set $P \cup \{q\}$. This imposes two problems. First of all the obstacle points to be generated for q cannot be computed in advance. Therefore a data structure is needed that stores the obstacles such that these obstacle points can be computed in an efficient manner. Secondly the value of ε depends on q, so we cannot even compute all obstacle points for the points in P beforehand. Fortunately the case in which ε was needed (namely where $o \supseteq Rect(p, q)$) can now be handled in another way: since we are only interested in points in P that are being dominated, we can just remove points (i.e. they are not present in the data structure) if they lie on the boundary of an obstacle that lies (partially) to its north-east (see Figure 5). The same thing can be done for q: if we find out that q lies on the boundary of an obstacle that lies (partially) to its south-west then we know that q does not dominate any point with respect to the obstacles.

Thus a query is performed as follows: first we check whether q lies on the interior or the north-west boundary of an obstacle. If so, we know that q does not dominate any point. If not, then the first intersection points o_l and o_d of the leftward and downward directed ray from q with an obstacle are determined. Now we could insert

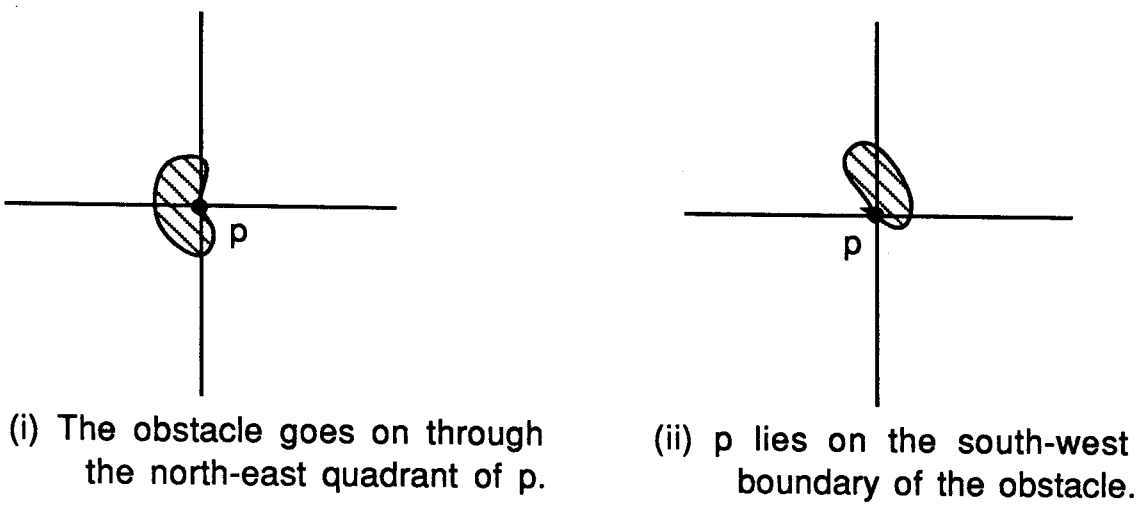

Figure 5: The cases where a point p is removed.

o_l and o_d into the structure and then do a query with q as described in the previous section, but this way the query time becomes $O(\log^3 n + Q(n) + k)$ (where $Q(n)$ is the time needed to compute o_l and o_d), because of the insertion of the obstacle points. Looking at the query algorithm of section 3.2, it can be seen that it suffices to search with both q_y and $(o_d)_y$ in T, thereby restricting ourselves to points between (with respect to y–coordinate) q and o_d, and to initialize $YMAX := (o_l)_x$, thereby restricting to points to the right of o_l. This way the query time is $O(\log^2 n + Q(n) + k)$. Again if O consists of simple, non-intersecting polygons of bounded degree, this approach yields good results:

Theorem 5 *A structure exists that solves the domination query problem for a set P of points and a set O of simple, non-intersecting polygons of bounded degree with a query time of $O(\log^2 n + k)$, where $n = |P| + |O|$ and k is the number of answers. This structure uses $O(n \log n)$ storage and has a preprocessing time of $O(n \log^2 n)$. Insertions in P can be performed in time $O(\log^3 n)$ and deletions in P in $O(\log^2 n)$.*

Proof: When O consists of simple, non-intersecting polygons of bounded degree, a structure using $O(n \log n)$ storage exists (see [11]) in which o_l and o_d can be computed in time $O(\log^2 n)$ after $O(n \log n)$ preprocessing. To insert a point $p \in P$ into the structure we compute its corresponding obstacle points (these obstacle points can be computed using a similar structure to the one used to compute o_l and o_d). Then p and the obstacle points are inserted into the structure, taking $O(\log^3 n)$ in total. When we delete a point from P we do not delete its corresponding obstacle points. This way the deletion time remains $O(\log^2 n)$. To avoid that the structure will contain too many useless obstacle points, we rebuild the entire structure after $\frac{1}{2}n$ deletions. Using the general techniques of [10] for global rebuilding this gives an extra $O(\log^2 n)$ per update (worst-case).

The above discussion, together with Theorem 3, now gives the desired results. Details are left to the reader. □

Note that the reduction approach makes the structure static with respect to the obstacles: changing O may cause a change in the obstacle points that should be present in the structure for each point in P.

5 Concluding remarks

In this paper a generalization of dominance, namely dominance in the presence of obstacles, has been studied. An optimal $O(n \log n + k)$ solution was given for the All Pairs Problem. Also the query version of the problem was considered. Two structures were presented: a static one with a query time of $O(\log n + k)$ and a fully dynamic one with a query time of $O(\log^2 n + k)$. These structures used $O(n \log^2 n)$ $O(n \log n)$ space respectively. Finally it was indicated how obstacles that were objects instead of points could be handled.

A number of questions remain. E.g., we only considered the two-dimensional case. Can the methods be extended to higher dimensions and with what bounds? For the Query Problem, the solution are not (proved to be) optimal. Would a structure exist using only linear space with a query time of, say, $O(\log^2 n + k)$? Finally, the framework for handling arbitrary obstacles works fine for the All Pairs Problem and for the Query Problem in case the obstacle set is static, but it makes a query structure based on this principle static with respect to the obstacles. It is interesting to look for other solutions to the query problem, in which the obstacles themselves are stored, so that it can be made dynamic with respect to the obstacles.

Acknowledgement

We would like to thank Derick Wood (University of Waterloo, Canada) for the helpful discussions on the All Pairs Problem during his stay in Utrecht in November '87.

References

[1] Bentley, J.L., and D. Wood, An optimal algorithm for reporting intersections of rectangles, *IEEE Transactions on Computers*, C-29 (1980), pp. 571-579.

[2] de Berg, M.T., and M.H. Overmars, Dominance in the presence of obstacles, *Proc. 14th International Workshop on Graph-Theoretic Concepts in Computer Science WG'88,* 1988, to apppear.

[3] Chazelle, B., and L.J. Guibas, Fractional cascading: I. A data structuring technique, *Algorithmica* 1 (1986), pp. 133-162.

[4] Güting, R.H., O. Nurmi and T. Ottmann, The direct dominance problem, *Proc. 1st ACM Symp. Computational Geometry,* 1985, pp. 81-88.

[5] Knuth, D.E., The art of computer programming, Vol. 3: sorting and searching, *Addison-Wesley, Reading, Mass.*, 1973.

[6] McCreight, E.M., Priority search trees, *SIAM J. Computing* 14 (1985), pp.257-276.

[7] Mehlhorn, K., Data structures and algorithms 3: multi-dimensional searching and computational geometry, *Springer-Verlag*, 1984.

[8] Munro, J.I., M.H. Overmars and D. Wood, Variations on visibility, *Proc. 3rd ACM Symp. Computational Geometry,* 1987, pp. 291-299.

[9] Olivié, H.J., Half-balanced binary search trees, *RAIRO Theoretical Informatics* 16 (1982), pp. 51-71.

[10] Overmars, M.H., *The Design of Dynamic Data Structures*, Lecture Notes in Computer Science 156, Springer Verlag, 1983.

[11] Overmars, M.H., Range searching in a set of line segments, *Proc. 1st ACM Symp. Computational Geometry*, 1985, pp. 177-185.

[12] Overmars, M.H., and D. Wood, On rectangular visibility, *J. Algorithms* 9 (1988), pp. 372-390.

[13] Willard, D.E., and G.S. Lueker, Adding range restriction capability to dynamic data structures, *Journal of the ACM* 32 (1985), pp. 597-617.