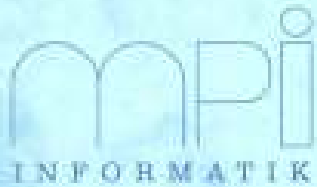
MAX-PLANCK-INSTITUT FÜR INFORMATIK

An Optimal Construction Method for Generalized Convex Layers

Hans-Peter Lenhof Michiel Smid

MPI-I-91-112 August 1991



Im Stadtwald
W 6600 Saarbrücken
Germany

An Optimal Construction Method for Generalized Convex Layers

Hans-Peter Lenhof Michiel Smid*
Max-Planck-Institut für Informatik
D-6600 Saarbrücken, Germany

August 13, 1991

Abstract

Let P be a set of n points in the Euclidean plane and let C be a convex figure. In 1985, Chazelle and Edelsbrunner presented an algorithm, which preprocesses P such that for any query point q, the points of P in the translate $C + q$ can be retrieved efficiently. Assuming that constant time suffices for deciding the inclusion of a point in C, they provided a space and query time optimal solution. Their algorithm uses $O(n)$ space. A query with output size k can be solved in $O(\log n + k)$ time. The preprocessing step of their algorithm, however, has time complexity $O(n^2)$. We show that the usage of a new construction method for layers reduces the preprocessing time to $O(n \log n)$. We thus provide the first space, query time and preprocessing time optimal solution for this class of point retrieval problems. Besides, we present two new dynamic data structures for these problems. The first dynamic data structure allows on-line insertions and deletions of points in $O((\log n)^2)$ time. In this dynamic data structure, a query with output size k can be solved in $O(\log n + k(\log n)^2)$ time. The second dynamic data structure, which allows only semi-online updates, has $O((\log n)^2)$ amortized update time and $O(\log n + k)$ query time.

1 Introduction

We consider the following problem: Let P be a set of n points in the Euclidean plane and let C be a convex figure. Preprocess P such that for any query point q, the points of P in the translate $C + q$ can be retrieved efficiently.

In 1985, Chazelle and Edelsbrunner [6] provided a space and query time optimal solution for this class of point retrieval problems. Their solution uses $O(n)$ space. A query with output size k can be solved in $O(\log n + k)$ time. The preprocessing step, however, has time complexity $O(n^2)$.

In a few special cases alternative solutions have been developed:

- Dynamic fixed polygonal windowing problem: In this case the boundary of the figure C is a polygon. In 1986, Klein et al. [9] presented an optimal dynamic solution for figures with a constant number of boundary edges. Their dynamic data structures use $O(n)$ space. Insertions and deletions can be carried out in $O(\log n)$ time. A query with output size k can be done in $O(\log n + k)$ time.

*This author was supported by the ESPRIT II Basic Research Actions Program, under contract No. 3075 (project ALCOM).

- Fixed radius neighbor problem: In this case C is a disk. The previously best known solutions to this problem are:

	preprocessing	query	space
[5]	$O(n(\log n)^2(\log\log n)^2)$	$O(\log n + k)$	$O(n(\log n \log\log n)^2)$
	$O(n \log n)$	$O(k(\log n)^2)$	$O(n \log n)$
prob.	?	$O(\log n + k)$	$O(n(\log n)^2)$
[6]	$O(n^2)$	$O(\log n + k)$	$O(n)$
[1]	polyn. time	$O(\log n + k)$	$O(n \log n)$
prob.	$O(n(\log n)^2)$	$O(\log n + k)$	$O(n \log n)$

The algorithms in [1] and [5] also handle queries with non-fixed radius.

As mentioned above, Chazelle and Edelsbrunner presented the first space and query time optimal solution. The preprocessing step of their algorithm, however, takes $O(n^2)$ time. During this preprocessing step the Euclidean plane is decomposed into cells. Then, for every non-empty cell, i.e., a cell that contains points of P, a family of so-called layers is constructed. In the worst case, the layers construction method of Chazelle and Edelsbrunner requires $\Omega(n^2)$ time.

In this paper, we introduce a kind of "dual or mirror layers" with respect to the layers, which Chazelle and Edelsbrunner use to build their query data structures. A family of these "dual layers" can be constructed in $O(n \log n)$ time. By walking across the "dual layers" of such a family, we can determine a family of layers in $O(n)$ time. Hence the whole layer computation takes $O(n \log n)$ time. This new construction method for lower and upper envelopes only works in the case of curves, which are translates of a convex curve.

We thus provide the first space, query time and preprocessing time optimal solution for this class of point retrieval problems. We want to emphasize that all these problems can now be solved optimally by one general technique. Besides, this layer construction method gives new dynamic data structures for the above class of point retrieval problems. The data structure, which we use to determine the "dual layers", can also be used to retrieve the points that are contained in the translate $C + q$. On-line insertions and deletions in this query data structure can be carried out in $O((\log n)^2)$ time. A query with output size k can be solved in $O(\log n + k(\log n)^2)$ time.

This update time is improved for so-called *semi-online updates*, as introduced by Dobkin and Suri [8]. (See also [12].) A sequence of updates is called semi-online, if the insertions are on-line, but with each inserted point p, we get an integer d which says that p will be deleted d updates from the moment of insertion. In case $d = \infty$, point p will never be deleted.

In [8, 12], the following is shown. Let D be a static data structure for a decomposable searching problem. Let $S(n)$, $P(n)$ and $Q(n)$ denote the size, the building time and the query time of D, respectively. Then, there exists a dynamic data structure for the same query problem, that allows semi-online updates. This data structure has size $O(S(n))$, a query time of $O(Q(n) \log n)$ and an amortized update time of $O((P(n)/n) \log n)$ per semi-online update. This dynamic data structure is a generalization of the one that is obtained by applying Bentley's logarithmic method, see [2]. It maintains a collection of $O(\log n)$ static structures.

If we apply this result to our static data structure, we get a query time of $O((\log n)^2 + k)$ and an amortized update time of $O((\log n)^2)$. Using fractional cascading [7], the query time is improved to $O(\log n + k)$.

In Section 2, we describe the algorithm of Chazelle and Edelsbrunner. The new construction method for layers will be explained in Section 3. As part of a few concluding remarks in Section 4, we present some applications of the point retrieval algorithm in computer simulations of molecule docking. In these interactive motion planning problems disks with fixed radius appear as projections of the spherical atoms. Besides, we present the results concerning semi-online updates.

2 The algorithm of Chazelle and Edelsbrunner

In this section we explain the algorithm given in [6]. First we introduce in Subsection 2.1 relevant geometric notions and summarize the computational assumptions. In Subsection 2.2 we describe the way the Euclidean plane is divided into cells. In Subsection 2.3 we discuss the query data structure for a cell. Subsections 2.1, 2.2 and 2.3 summarize Sections 2, 3 and 4 of [6].

2.1 Geometric notions and computational assumptions

The Euclidean plane is denoted by E^2. Let $\lambda \in \mathcal{R}$, $v = (v_x, v_y), w = (w_x, w_y) \in E^2$, and $A, B \subseteq E^2$. We use the following notation:

$$\begin{array}{ll} int(A) := \text{interior of } A & cl(A) := \text{closure of } A \\ bd(A) := \text{boundary of } A & conv(A) := \text{convex hull of } A \\ v + w := (v_x + w_x, v_y + w_y) & \lambda v := (\lambda v_x, \lambda v_y) \\ A_v := A + v := \{a + v | a \in A\} & A + B := \{a + b | a \in A, b \in B\} \end{array}$$

Throughout this paper, C is a bounded convex closed figure in E^2. Let L (resp. R) denote the point in $bd(C)$ with minimal (resp. maximal) x-coordinate (If L (resp. R) is not unique, we take the point with maximal (resp. minimal) y-coordinate.). For two different points $v, w \in E^2$ we define

$$I(v, w) := bd((-C)_v) \cap bd((-C)_w),$$

where $(-C) := \{-c | c \in C\}$. The intersection $I(v, w)$ consists of at most two line segments (see Figure 1). Hence $I(v, w)$ can be represented in a constant amount of space.

Definition 1 *The convex closed figure C is called computable, if*

1. *constant time suffices to test for any point $p \in E^2$ whether or not p is contained in C, and*
2. *constant time suffices to compute $I(v, w)$ for any two (potentially infinitesimal close) points v and w in E^2.*

Lemma 1 ([6]) *a) If C is computable, then L and R can be determined in constant time. b) Let C be computable, p a point in E^2 and l a vertical line through p. Constant time suffices to decide whether (1) C is to the right of l or (2) C is to the left of l or (3) $C \cap l \neq \emptyset$ and p is above C or p is contained in C or p is below C.*

We now define the so-called silos and rotated silos as substitutes for $(-C)$ and C.

Definition 2 *Let v be a point, $r_u(v)$ the vertical ray with v as lower endpoint and $r_d(v)$ the vertical ray with v as upper endpoint. We call $S(v) := -C + r_u(v)$ the silo of v and $RS(v) := C + r_d(v)$ the rotated silo of v.*

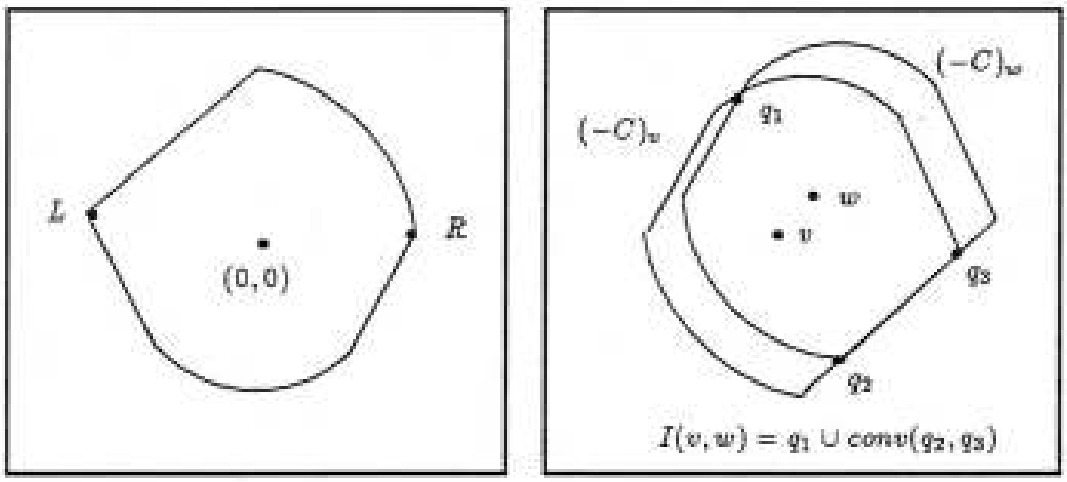


Figure 1: Example of a convex computable figure C and the intersection of two translates $(-C)_v$ and $(-C)_w$.

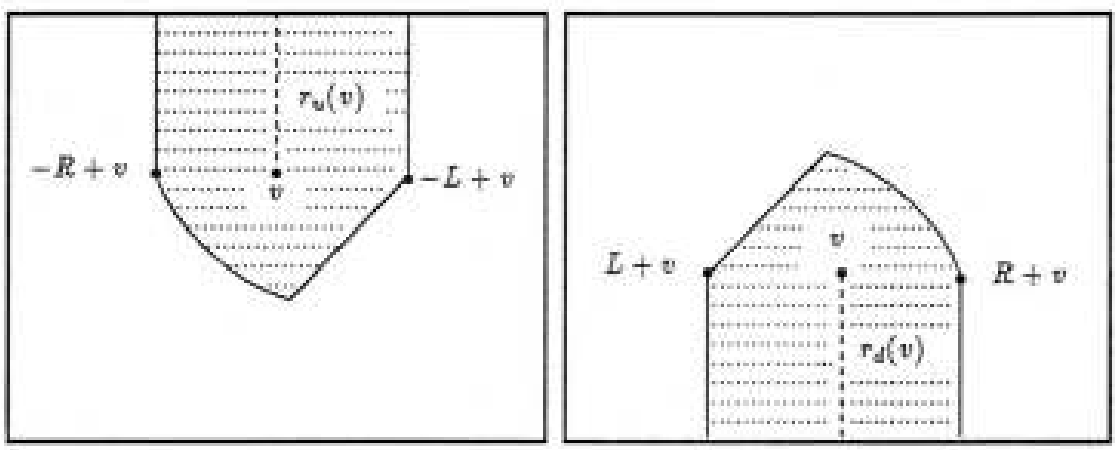


Figure 2: Silo $S(v)$ and rotated silo $RS(v)$.

Consider Figure 2. The silo $S(v)$ contains the set of points q, such that C_q intersects $r_u(v)$. The boundary $bd(S(v))$ consists of the two rays $-R + r_u(v)$ and $-L + r_u(v)$ and the lower part of $bd((-C)_u)$. The rotated silo $RS(v)$ contains the set of points q, such that $(-C)_q$ intersects $r_d(v)$. The boundary $bd(RS(v))$ consists of the two vertical rays $R + r_d(v)$ and $L + r_d(v)$ and the upper part of $bd((C)_u)$. The boundary of the intersection of the two silos $S(v)$ and $S(w)$ (resp. rotated silos $RS(v)$ and $RS(w)$) is either empty or consists of a single point, a segment or a ray. This boundary can be computed in constant time.

2.2 How to divide the Euclidean plane into cells

In this section we describe how the Euclidean plane is divided into rectangles, such that any translate C_q of C does not intersect more than nine rectangles. We call the rectangles the *cells* of the subdivision. For every cell CE, we construct four query data structures, in which we store the points of $P_{CE} := P \cap CE$.

Let C be a computable figure with non-empty interior and extreme points L and R. We assume that the segment $\bar{s} = conv(R, L)$ is parallel to the x-axis of the coordinate system. The segment $\bar{s}$ decomposes C into two computable figures C_a and C_b. The figure C_a is the upper part, and C_b is the lower part. We assume now that C is C_a and explain the query algorithm for $C = C_a$. The query algorithm for C_b works analogously.

Let M be a point on $bd(C)$ that lies on a tangent $\bar{t} \neq \bar{s}$, parallel to $\bar{s}$ (see Fig. 3). The vertical projection of M onto $\bar{s}$ is denoted by N. We now define two orthogonal vectors $\vec{h} = \frac{1}{2}(R - L)$ and $\vec{v} = \frac{1}{2}(M - N)$, whose lengths $|\vec{v}|$ and $|\vec{h}|$ fix the width and height of a cell. We consider the decomposition $G = \{CE_{ij}\}$ of the Euclidean plane E^2 in cells $CE_{ij} = \{(p_x, p_y) | i|\vec{h}| \leq p_x < (i+1)|\vec{h}|, j|\vec{v}| \leq p_y < (j+1)|\vec{v}|\}$, where i and j range over the integers. We determine the non-empty cells, sort them in lexicographical order, and store them in a balanced binary search tree. It is easy to see that any translate C_q intersects at most nine cells lying in three consecutive rows and columns. Given a translate C_q, we can find the non-empty cells, which are intersected by C_q, in $O(\log n)$ time.

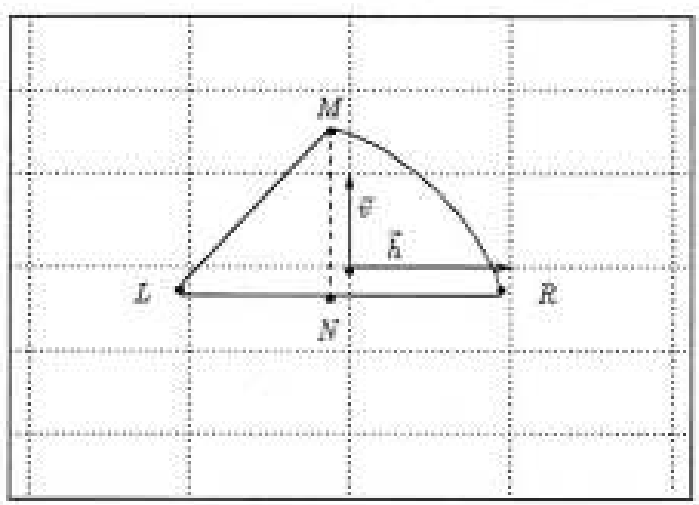


Figure 3: Decomposing C and E^2.

We distinguish between the following kinds of intersection:

Definition 3 *Let CE be a cell and let N = north, E = east, S = south and W = west denote the four edges of $bd(CE)$. We say that C_q is D-grounded, if $C_q \cap D$ equals the orthogonal*

projection of $C_q \cap cl(CE)$ onto D, for $D \in \{N, S, W, E\}$. C_q is said to be grounded if it is D-grounded for at least one assignment of D to N, S, W or E.

Chazelle and Edelsbrunner prove:

Lemma 2 ([6]) *Let q be a point in E^2 and CE a cell of G, such that $C_q \cap CE \neq \emptyset$. Then C_q is grounded with respect to CE.*

Given a query translate C_q and a cell CE, which is intersected by C_q, we can determine in constant time, whether C_q is N, S, W or E-grounded with respect to the cell CE (see [6]).

2.3 The algorithm of Chazelle and Edelsbrunner

For every non-empty cell CE we build four data structures, one for every assignment of D to N, E, S, W. Since all four data structures are constructed in the same way, we only consider the problem "C_q is S-grounded with respect to CE" and show how the query data structure for S-grounded queries is built. Hence we assume for the rest of this section, that the query translates C_q are S-grounded with respect to the non-empty cell CE. In such a query, we want to compute all points $p \in P_{CE} = P \cap CE$ which lie in C_q.

Observation 1 *Point p lies in C_q (resp. $RS(q)$) if and only if $q \in (-C)_p$ (resp. $q \in S(p)$).*

Using Observation 1 we can show the following lemma:

Lemma 3 ([6]) *Let p be a point of P_{CE}. Then, p is in C_q if and only if $q \in S(p)$.*

Lemma 3 tells us, that we can compute all points $p \in P_{CE} \cap C_q$ in the following way: Compute all silos $S(p)$ for $p \in P_{CE}$ which contain q. Note that these are exactly those silos, whose boundaries are intersected by the vertical ray from q towards $y = -\infty$.

How can we find all these silos $S(p)$? We assume that the points $P_{CE} = \{p_1, \cdots, p_m\}$ are sorted in order of increasing x-coordinates. Since the difference of the x-coordinates of any two points in P_{CE} is less than $|\bar{h}|$,

$$U := \bigcup_{1 \leq i \leq m} S(p_i)$$

is connected. The boundary $L(P_{CE}) = bd(U)$, which we call the *S-layer* of P_{CE}, is an unbounded, connected, x-monotone curve. Any vertical line intersects this S-layer in at most one point or ray. We call

$$e_i := [L(P_{CE}) \cap bd(S(p_i))] \setminus \bigcup_{1 \leq j < i} bd(S(p_j))$$

the *edge* e_i of p_i. Since e_i can be empty, not every point $p \in P_{CE}$ contributes to a part of $L(P_{CE})$. If e_i is empty, then p_i is called *redundant* and we define

$$ext(P_{CE}) := \{p \in P_{CE} \mid p \text{ non-redundant}\}.$$

Observation 2 ([6]) *Let $e_{k_1}, \cdots, e_{k_t}$ be the sequence of non-empty edges of $L(P_{CE})$ ordered from left to right.*
(i) For each $1 \leq i \leq t-1$, the maximal x-coordinate of $bd(S(p_{k_i})) \cap bd(S(p_{k_{i+1}}))$ is the x-coordinate of the right endpoint of e_{k_i} and the left endpoint of $e_{k_{i+1}}$.
(ii) $k_i < k_{i+1}$ for $1 \leq i \leq t-1$, i.e., the ordering of the edges coincides with that of P_{CE}.

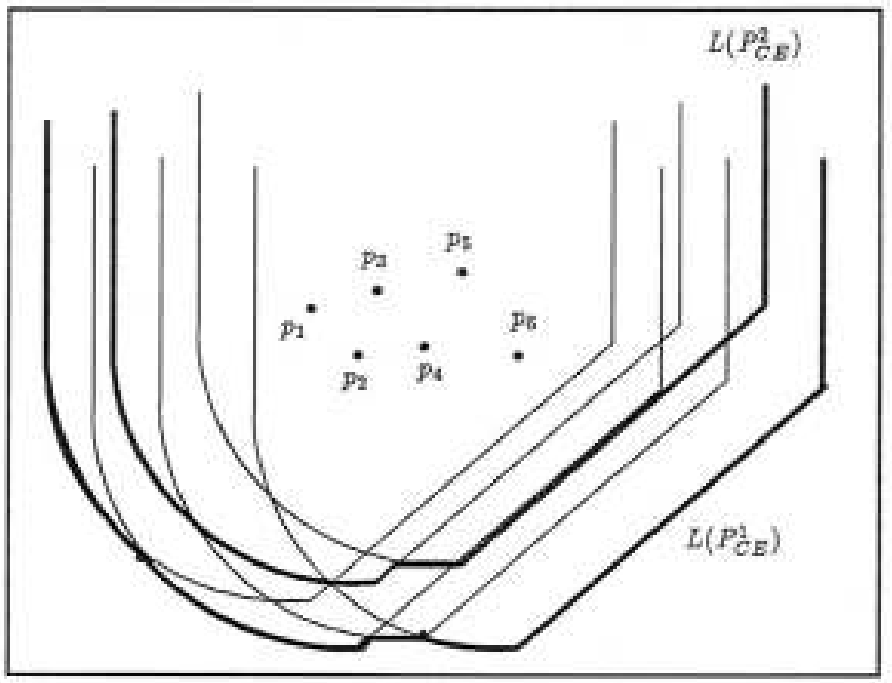


Figure 4: Example for a family of S-layers

Chazelle and Edelsbrunner use Observation 2 to develop an $O(n)$ time construction method for $L(P_{CE})$. But why do they compute $L(P_{CE})$?

Lemma 4 ([6]) *Let $e_{k_1}, \cdots, e_{k_t}$ be the sequence of non-empty edges of $L(P_{CE})$ ordered from left to right. Let C_q be a translate of C that is S-grounded with respect to CE, and $r := r_d(q)$ the vertical ray with upper point q.*
(i) The ray r intersects $L(P_{CE})$ if and only if $C_q \cap P_{CE}$ is not empty.
(ii) If $e_{k_i} \cap r \neq \emptyset$, then p_{k_1} lies in C_q and there are indices i and j, with $i \leq l \leq j$, such that $C_q \cap ext(P_{CE}) = \{p_{k_\alpha} | i \leq \alpha \leq j\}$.

The algorithm for finding all points of $C_q \cap P_{CE}$ works as follows: First, we search for the points of $C_q \cap ext(P_{CE})$. In order to find these points, we search for the edge e_{k_l} that intersects the ray $r := r_d(q)$. (If there is no such edge then $C_q \cap P_{CE} = \emptyset$ and the algorithm stops.) Note that e_{k_l} can be found in $O(\log t)$ time. Then we start at e_{k_l} and walk along $L(P_{CE})$ to the left, until we find an edge $e_{k_{i-1}}$ such that $q \notin S(p_{k_{i-1}})$. Analogously, we walk to the right, again starting at e_{k_l}, until we find an edge $e_{k_{j+1}}$ such that $q \notin S(p_{k_{j+1}})$. During these walks, we report all points $p_{k_i}, p_{k_{i+1}}, \cdots, p_{k_j}$. In this way, we have determined the points of $C_q \cap ext(P_{CE})$ in $O(\log t + |C_q \cap ext(P_{CE})|)$ time.

Let $P^1_{CE} := P_{CE}$ and $P^i_{CE} := P^{i-1}_{CE} \setminus ext(P^{i-1}_{CE})$ for $i > 1$. Suppose we have constructed the S-layers $L(P^i_{CE})$, for $i \geq 1$. If $C_q \cap ext(P^1_{CE}) \neq \emptyset$, we search for all points $p \in C_q \cap ext(P^2_{CE})$ by testing, if the ray r intersects $L(P^2_{CE})$. If r intersects the second S-layer, we walk across $L(P^2_{CE})$ in the same way as described above. We continue to test the S-layers, until we find an S-layer, which is not intersected by the ray r, or until we have checked all non-empty S-layers. Since the family $\mathrm{LS}(P_{CE}) = (L(P^1_{CE}), \cdots, L(P^s_{CE}))$ of non-empty S-layers is nested, an S-layer which lies above a non-intersected S-layer, cannot be intersected by the ray r. Hence, no point represented by such an S-layer, can lie in C_q.

All k_{CE} points $p \in C_q \cap P_{CE}$ are reported in $O(|\text{visited S-layers}| \log m + k_{CE})$ time. Since we have found at least one point in every S-layer, with exception of the last visited, the query time is $O(k_{CE} \log m)$. By applying Chazelle's hive graph [4] to $L_S(P_{CE})$, the query time can be improved to $O(\log m + k_{CE})$. The hive graph connects $L(P^i_{CE})$ with $L(P^{i+1}_{CE})$ in such a way that the knowledge of the edge in $L(P^i_{CE})$ that intersects r, allows us to find the intersecting edge in $L(P^{i+1}_{CE})$ in constant time. $O(m)$ space suffices to store the hive graph of $L_S(P_{CE})$. The hive graph can be constructed in $O(m)$ time (see [4]). Hence the cost of the preprocessing step is dominated by the $O(m^2)$ operations required to construct the family of S-layers. (Note that there are at most m S-layers, each of which is constructed in $O(m)$ time.)

We can now cite the main result of [6]:

Theorem 1 *Let P be a set of n points in the Euclidean plane E^2 and C a convex computable figure. There exists a data structure, such that $O(k + \log n)$ time suffices to retrieve all k points of P lying in a query translate C_q. The data structure has size $O(n)$ and can be constructed in $O(n^2)$ time.*

3 Fast construction method for S-layers

Let $P_{CE} = P \cap CE = \{p_1, \cdots, p_m\}$ be the sequence of points in cell CE, sorted in order of increasing x-coordinates. We assume that there are no two points in P_{CE} with the same x-coordinate. (In Section 4 we show how degenerate cases can be handled.) We consider again only S-grounded queries. In this section we describe a method to construct the family of S-layers for the point set P_{CE} with respect to the fixed convex computable figure C, which takes $O(m \log m)$ time. In Subsection 3.1 we present a new geometric concept for convex curves, the so-called dual or mirror S-layers. Then we show that the family of S-layers can be constructed from the family of dual S-layers, in $O(m)$ time. In Subsection 3.2 we describe an $O(m(\log m)^2)$ time dual S-layer construction algorithm, which is a modification of Overmars and van Leeuwen's convex layers construction algorithm (see [10]). Besides, we show that this dual S-layer construction method gives new dynamic query data structures for the investigated class of point retrieval problems. In Subsection 3.3 we discuss a modified version of Chazelle's convex layers construction algorithm [3] and we show that this algorithm enables the construction of the dual S-layers with only $O(m \log m)$ operations.

3.1 Dual S-layers

We now define minimal representations for the S-layer $L(P_{CE})$. Recall that the points of $P_{CE} = \{p_1, \cdots, p_m\}$ are sorted by their x-coordinates.

Definition 4 *a) A sequence $R(P_{CE}) = < p_{k_1}, \cdots, p_{k_t} > \subseteq P_{CE}$, which satisfies the following properties, is called a representation system or r-system of the S-layer $L(P_{CE})$:*

- $k_i < k_{i+1}$ *for all* $i = 1, \cdots, t-1$
- $bd(S(p_{k_i})) \cap L(P_{CE}) \neq \emptyset$ *for all* $i = 1, \cdots, t$
- $L(P_{CE}) \subseteq \bigcup_{j=1}^{t} bd(S(p_{k_j}))$.

b) An r-system $R(P_{CE}) = < p_{k_1}, \cdots, p_{k_t} >$ of the S-layer $L(P_{CE})$ with minimal length t is called a minimal r-system of the S-layer $L(P_{CE})$.

In order to prevent the number of variables becoming too large, we redefine the edges e_{k_i} and sets P^i_{CE}. If $R(P_{CE}) = < p_{k_1}, \cdots, p_{k_z} >$ is a minimal r-system for $L(P_{CE})$, then the S-layer $L(P_{CE})$ can be represented by the union

$$L(P_{CE}) = \bigcup_{i=1}^{z} e_{k_i} \quad \text{of the edges} \quad e_{k_i} := [L(P_{CE}) \cap bd(S(p_{k_i}))] \setminus \bigcup_{1 \le j < i} bd(S(p_{k_j})).$$

We say that edge e_{k_i} represents point p_{k_i}. Let $P^1_{CE} := P_{CE}$ and $P^i_{CE} := P^{i-1}_{CE} \setminus R_{i-1}$, where $R_{i-1} := R(P^{i-1}_{CE})$ is a minimal r-system of the S-layer $L(P^{i-1}_{CE})$. Furthermore let $R_S(P_{CE}) = (R_1, \cdots, R_z)$ be a family of minimal r-systems defined recursively in the above way, such that every point p of P_{CE} is contained in one P^i_{CE} for some $1 \le i \le z$. It is easy to see, that the assertions of Lemma 4 are also valid for each S-layer $L(P^i_{CE})$ defined as above. Furthermore each family of S-layers that is constructed in the above way, consists of nested S-layers. Hence, if we know such a family of minimal r-systems for $L(P_{CE})$, we can construct the query data structures used in [6] in $O(m)$ time.

We need some more notions: Let a and b be two points in cell CE with different x-coordinates (we assume $a_x < b_x$). If the silos $S(a)$ and $S(b)$ intersect each other, we call the point in the intersection $bd(S(a)) \cap bd(S(b))$ having the smallest x-coordinate the *si-point* of a and b and the point having the largest x-coordinate the *SI-point* of a and b. We use the notations $si(a,b)$ and $SI(a,b)$ for these intersection points. If the intersection is a unique point, then $si(a,b) = SI(a,b)$. Furthermore we define $int_{si}(a,b) := int(RS(si(a,b)))$, $int^{SI}(a,b) := int(RS(SI(a,b)))$ and $int^{si}_{SI}(a,b) := int(RS(si(a,b))) \cup int(RS(SI(a,b)))$. The intersection

$$e(a,b) := bd(RS(SI(a,b))) \cap \{p = (p_x, p_y) | \, a_x \le p_x \le b_x\}$$

will be called the *dual-edge* or *d-edge* of a and b. The d-edge of a and b is x-monotone and connects a and b. If we would use the boundary $bd(RS(v))$, where v is an arbitrary point in the intersection of the silos a and b, instead of $bd(RS(SI(a,b)))$ in the d-edge definition, we would get the same set. The region

$$reg(a,b) := \{p = (p_x, p_y) | \, a_x < p_x < b_x \wedge \text{p lies (strictly) above } e(a,b)\}$$

will be called the *region* of a and b (see Figure 5).

Lemma 5 *Let a, b, c be three points in cell CE with different x-coordinates. Then $bd(S(c)) \cap bd(S(a) \cup S(b)) = \emptyset$ if and only if $c \in reg(a,b)$.*

Proof: "$\Rightarrow$": Since a, b, c lie in the same cell, every pair of the silos $S(a), S(b), S(c)$ has a non-empty boundary intersection. Assuming wlog $a_x < b_x$, we show that $c \notin reg(a,b)$ implies $bd(S(c)) \cap bd(S(a) \cup S(b)) \neq \emptyset$. If $c_x < a_x$ or $b_x < c_x$, it is obvious that $bd(S(c)) \cap bd(S(a) \cup S(b)) \neq \emptyset$. Otherwise $a_x < c_x < b_x$ and c lies below or on $e(a,b)$. Then, $c \in RS(SI(a,b))$ (see Fig. 5). Observation 1 implies then $SI(a,b) \in S(c)$. Hence $bd(S(c)) \cap bd(S(a) \cup S(b)) \neq \emptyset$.
"$\Leftarrow$": We assume that $a_x < b_x$. Since $c \in reg(a,b)$, Observation 1 implies $SI(a,b) \notin S(c)$. Furthermore the convexity of the silos and the fact that $a_x < c_x < b_x$, guarantee that (1) left of $SI(a,b)$ the boundary $bd(S(c))$ lies (strictly) above $bd(S(a))$ and (2) right of $SI(a,b)$ the boundary $bd(S(c))$ lies (strictly) above $bd(S(b))$. Hence $bd(S(c)) \cap bd(S(a) \cup S(b)) = \emptyset$. ■

We now define a kind of "dual layer" to the S-layer $L(P_{CE})$:

Definition 5
a) Let $DL(P_{CE})$ be the lower envelope of the set of d-edges $\{e(p_i, p_j) | p_i, p_j \in P_{CE}\}$. We call

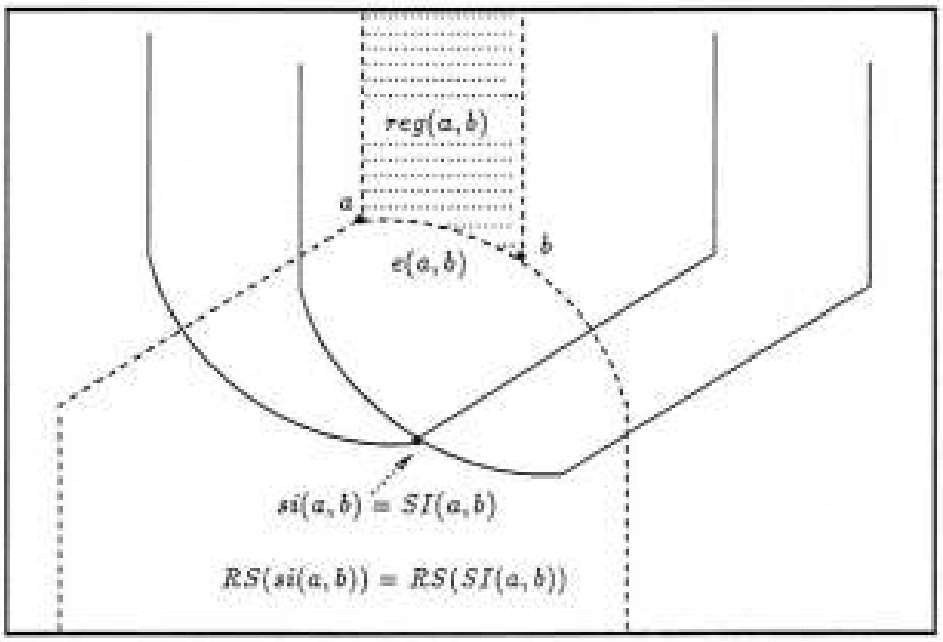


Figure 5: *SI*-point, d-edge and region of two points a and b.

$DL(P_{CS})$ the dual S-layer or dS-layer of P_{CS}.
b) A sequence $DR(P_{CS}) =< p_{k_1}, \cdots, p_{k_t} > \subseteq P_{CS}$, which satisfies the following properties, is called a dual representation system or dr-system of $DL(P_{CS})$:

- $k_i < k_{i+1}$ *for all* $i = 1, \cdots, t-1$
- $e(p_{k_i}, p_{k_{i+1}}) \subseteq DL(P_{CS})$ *for all* $i = 1, \cdots, t-1$
- $\bigcup_{i=1}^{t-1} e(p_{k_i}, p_{k_{i+1}}) = DL(P_{CS})$.

c) A dr-system $DR(P_{CS}) =< p_{k_1}, \cdots, p_{k_t} >$ of $DL(P_{CS})$ with minimal length t is called a minimal dr-system of the dS-layer $DL(P_{CS})$.

The dS-layer is an x-monotone curve. All points $p \in P_{CS}$, which can be element in a minimal dr-system of $DL(P_{CS})$ or has to be an element in all minimal dr-systems of $DL(P_{CS})$, lie on the dS-layer $DL(P_{CS})$. Using Lemma 5 we can easily show, that $< p_{k_1}, \cdots, p_{k_t} >$ is a minimal dr-system for $DL(P_{CS})$ if and only if $< p_{k_1}, \cdots, p_{k_t} >$ is a minimal r-system for the S-layer $L(P_{CS})$.

Let $P_{CS}^1 := P_{CS}$ and $P_{CS}^i := P_{CS}^{i-1} \setminus DR_{i-1}$, where $DR_{i-1} = DR(P_{CS}^{i-1})$ is a minimal dr-system for $DL(P_{CS}^{i-1})$. By computing a family $DL_S(P_{CS}) = (DL(P_{CS}^1), \cdots, DL(P_{CS}^s))$ of dS-layers resp. a corresponding minimal dr-system $DR_S(P_{CS}) = (DR_1, \cdots, DR_s)$, we get a family $R_S(P_{CS}) = DR_S(P_{CS})$ of minimal r-systems, which enables us to build the query data structure for S-grounded queries in $O(m)$ time.

3.2 Convex layers and dS-layers

In this subsection we show that there are important similarities between lower convex hulls and dS-layers. These similarities enable us to construct dS-layers with the same methods that are used to construct lower convex hulls.

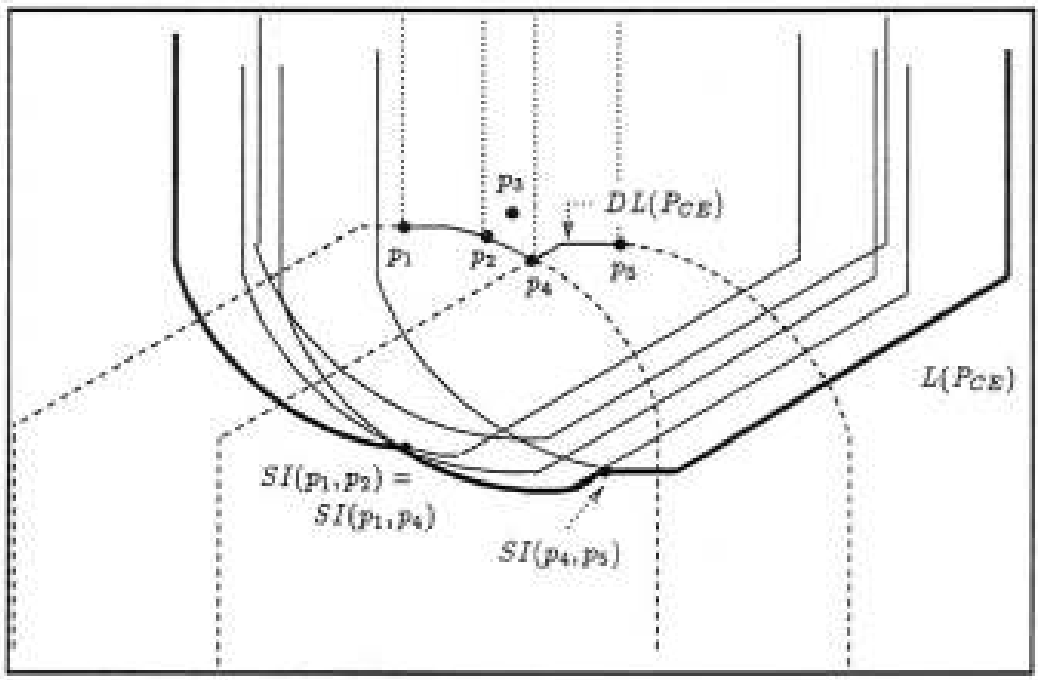


Figure 6: S-layer and dS-layer of a point set $P_{CE} = \{p_1, \cdots, p_5\}$. In the above situation $\{p_1, p_4, p_5\}$ is the minimal dr-systems for $DL(P_{CE})$.

The dS-layer consists of curves, which connect points of the corresponding point set. All points of this point set lie on or above the dS-layer. Given two dS-layers, where all points of the first dS-layer are to the left of all points of the second dS-layer, there is exactly one new d-edge on the dS-layer of all points, which connects the two dS-layers and shares only start- and endpoint with them.

Before going on with the above considerations, we transfer the terms *concave*, *reflex* and *supporting* from the theory of convex hulls to the theory of dS-layers: Let $P^{[l]}_{CE} = \{p_1, \cdots, p_h\}$ and $P^{[r]}_{CE} = \{p_{h+1}, \cdots, p_m\}$ with

$$(p_1)_x < (p_2)_x < \cdots < (p_h)_x < (p_{h+1})_x < \cdots < (p_m)_x$$

be a partition of P_{CE}. Let $DR(P^{[l]}_{CE}) =< p^l_1, \cdots, p^l_s >$ be a minimal dr-system of $DL(P^{[l]}_{CE})$ and $DR(P^{[r]}_{CE}) =< p^r_1, \cdots, p^r_t >$ a minimal dr-system of $DL(P^{[r]}_{CE})$. Let the indices u and v be such that $p^l_i = p_u$ and $p^r_j = p_v$. The d-edge $e(p^l_i, p^r_j)$ is called

- *si-supporting* in p^l_i, if for all $f \in \{1, \cdots, h\}$, $p_f \notin int_{si}(p^l_i, p^r_j)$,
- *si-supporting* in p^r_j, if for all $g \in \{h+1, \cdots, m\}$, $p_g \notin int_{si}(p^l_i, p^r_j)$,
- *si-concave* in p^l_i, if there is a point $p_f \in int_{si}(p^l_i, p^r_j)$, where $f \in \{u+1, \cdots, h\}$,
- *si-concave* in p^r_j, if there is a point $p_g \in int_{si}(p^l_i, p^r_j)$, where $g \in \{h+1, \cdots, v-1\}$,
- *si-reflex* in p^l_i, if there is a point $p_f \in int_{si}(p^l_i, p^r_j)$, where $f \in \{1, \cdots, u-1\}$,
- *si-reflex* in p^r_j, if there is a point $p_g \in int_{si}(p^l_i, p^r_j)$, where $g \in \{v+1, \cdots, m\}$.

Analogously we define *SI-supporting*, *SI-concave* and *SI-reflex*, by replacing $int_{si}(p_i^l, p_j^r)$ by $int^{si}(p_i^l, p_j^r)$ in the above definition. Note that $p_f, p_{f'} \in int_{si}(p_i^l, p_j^r)$, where $f \in \{1, \cdots, u-1\}$ and $f' \in \{u+1, \cdots, h\}$, is not possible, because in this case the d-edge $e(p_f, p_{f'})$ would lie below the d-edges $e(p_{i-1}^l, p_i^l), e(p_i^l, p_{i+1}^l)$. But this would contradict the assumption, that $DR(P_{CE}^{[l]})$ is a dr-system for $DL(P_{CE}^{[l]})$. Therefore, the three cases supporting, concave and reflex are mutually exclusive. The d-edge $e(p_i^l, p_j^r)$ is called a *supporting d-edge* for $DL(P_{CE}^{[l]})$ and $DL(P_{CE}^{[r]})$, if for all $f \in \{1, \cdots, m\}$, $p_f \notin int_{si}^{si}(p_i^l, p_j^r)$. Hence the d-edge $e(p_i^l, p_j^r)$ is a supporting d-edge if and only if $e(p_i^l, p_j^r)$ is si-supporting and *SI*-supporting in both endpoints p_i^l and p_j^r. The endpoints p_i^l and p_j^r of a supporting d-edge are called *supporting* left and right endpoint.

Note that every d-edge $e(p_{k_i}, p_{k_{i+1}})$, which belongs to two neighboring points $p_{k_i}, p_{k_{i+1}}$ of a minimal dr-system, is a supporting d-edge with respect to the corresponding point set.

In order to compute a minimal dr-system of $DL(P_{CE})$, we have to determine the "longest" supporting d-edge for $DL(P_{CE}^{[l]})$ and $DL(P_{CE}^{[r]})$. If we know a supporting d-edge $e(p_i^l, p_j^r)$, we can determine the "longest" supporting d-edge resp. a minimal dr-system of $DL(P_{CE})$ with the following procedure:

Case 1: If $e(p_{i-1}^l, p_{j+1}^r)$ is a supporting d-edge, set $DR(P_{CE}) := < p_1^l, \cdots, p_{i-1}^l, p_{j+1}^r, \cdots, p_t^r >$;

Case 2: If $e(p_{i-1}^l, p_{j+1}^r)$ is not a supporting d-edge and $e(p_{i-1}^l, p_j^r)$, $e(p_i^l, p_{j+1}^r)$ are supporting d-edges, then set $DR(P_{CE}) := < p_1^l, \cdots, p_{i-1}^l, p_j^r, \cdots, p_t^r >$, or set $DR(P_{CE}) := < p_1^l, \cdots, p_i^l, p_{j+1}^r, \cdots, p_t^r >$;

Case 3: If $e(p_{i-1}^l, p_{j+1}^r)$, $e(p_{i-1}^l, p_j^r)$ are not supporting d-edges and $e(p_i^l, p_{j+1}^r)$ is a supporting d-edge, then set $DR(P_{CE}) := < p_1^l, \cdots, p_i^l, p_{j+1}^r, \cdots, p_t^r >$;

Case 4: If $e(p_{i-1}^l, p_{j+1}^r)$, $e(p_i^l, p_{j+1}^r)$ are not supporting d-edges and $e(p_{i-1}^l, p_j^r)$ is a supporting d-edge, then set $DR(P_{CE}) := < p_1^l, \cdots, p_{i-1}^l, p_j^r, \cdots, p_t^r >$;

Case 5: If $e(p_{i-1}^l, p_{j+1}^r)$, $e(p_{i-1}^l, p_j^r)$ and $e(p_i^l, p_{j+1}^r)$ are not supporting d-edges, set $DR(P_{CE}) := < p_1^l, \cdots, p_i^l, p_j^r, \cdots, p_t^r >$.

Before we show how a supporting d-edge can be computed efficiently, we prove that we can determine in constant time, if a d-edge $e(p_i^l, p_j^r)$ is si-supporting, si-concave or si reflex (resp. *SI*-supporting, *SI*-concave or *SI*-reflex).

Lemma 6 *a) A d-edge $e(p_i^l, p_j^r)$ is*

- *si-supporting in p_i^l, if $p_{i-1}^l, p_{i+1}^l \notin int_{si}(p_i^l, p_j^r)$*
- *si-supporting in p_j^r, if $p_{j-1}^r, p_{j+1}^r \notin int_{si}(p_i^l, p_j^r)$*
- *si-concave in p_i^l, if $p_{i+1}^l \in int_{si}(p_i^l, p_j^r)$*
- *si-concave in p_j^r, if $p_{j-1}^r \in int_{si}(p_i^l, p_j^r)$*
- *si-reflex in p_i^l, if $p_{i-1}^l \in int_{si}(p_i^l, p_j^r)$*
- *si-reflex in p_j^r, if $p_{j+1}^r \in int_{si}(p_i^l, p_j^r)$.*

b) If we replace $int_{si}(p_i^l, p_j^r)$ by $int^{si}(p_i^l, p_j^r)$ in a), we get analogous conditions for the properties SI-supporting, SI-concave and SI-reflex.

Proof: a) Let $p_i^l = p_u$. We only have to show the statements for property si-supporting, because all other statements follow by definition. We prove the first statement for property si-supporting. The second statement can be shown analogously. So, assume that $p_{i-1}^l, p_{i+1}^l \notin int_{si}(p_i^l, p_j^r)$. We assume that a point $p_f \in int_{si}(p_i^l, p_j^r)$ with $f < u$ exists. Since $int_{si}^{re}(p_{i-1}^l, p_i^l)$ contains the part of $int_{si}(p_i^l, p_j^r)$, which lies to the left of the vertical line through p_i^l, the point p_f lies in $int_{si}^{re}(p_{i-1}^l, p_i^l)$. But this is a contradiction to the fact, that $e(p_{i-1}^l, p_i^l)$ is a supporting d-edge in $DL(P_{CE}^{[l]})$. Therefore a point $p_f \in int_{si}(p_i^l, p_j^r)$ with $f < u$ does not exist. A similar argument implies, that there is no point $p_f \in int_{si}(p_i^l, p_j^r)$ with $f > u$. Hence the d-edge $e(p_i^l, p_j^r)$ is si-supporting in p_i^l.
b) Analogous to a). ■

Lemma 6 implies that we can determine in constant time, if a given d-edge $e(p_i^l, p_j^r)$ is a supporting d-edge. In the following lemma we show how we can determine a supporting d-edge for the dS-layers $DL(P_{CE}^{[l]})$ and $DL(P_{CE}^{[r]})$ efficiently. Using this lemma, the dS-layer $DL(P_{CE})$ can be computed by a divide-and-conquer algorithm.

Lemma 7
Given the two minimal dr-systems $DR(P_{CE}^{[l]}) = < p_1^l, \cdots, p_s^l >$ and $DR(P_{CE}^{[r]}) = < p_1^r, \cdots, p_t^r >$, we can compute a supporting d-edge of the corresponding dS-layers in $O(\log s + \log t)$ time.

Proof: We assume that the two dr-systems are stored in two arrays. Let $1 \leq i \leq s$ and $1 \leq j \leq t$. We consider the d-edge $e(p_i^l, p_j^r)$. Each of the two points p_i^l and p_j^r can be classified (1) as either si-reflex or si-supporting or si-concave and (2) as either *SI*-reflex or *SI*-supporting or *SI*-concave with respect to the d-edge $e(p_i^l, p_j^r)$. As in the case of ordinary convex hull construction we classify nine possible cases, which are schematically illustrated in Figure 7.

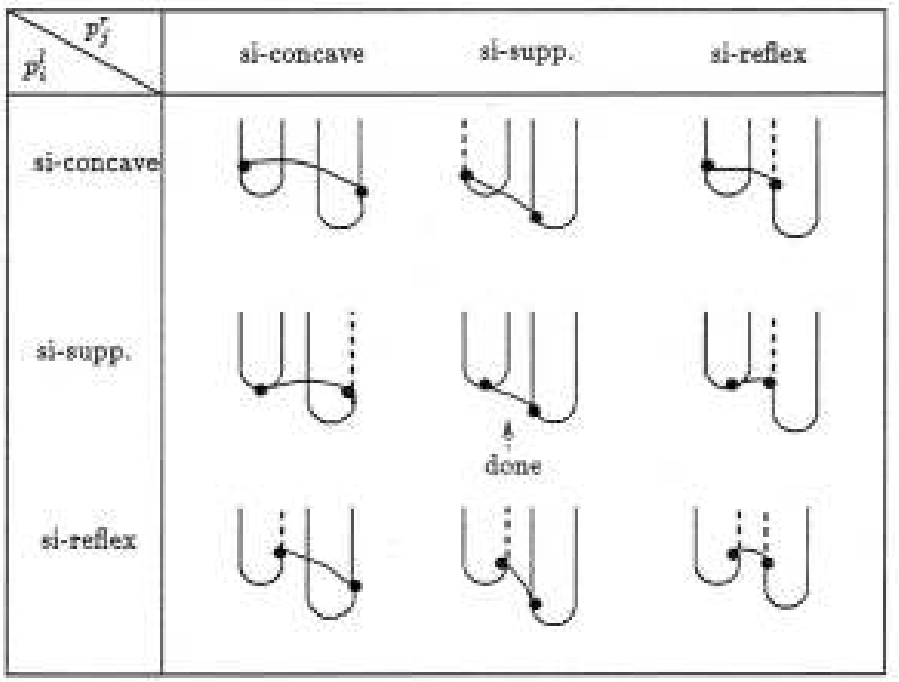


Figure 7: The nine possible cases.

In all cases, in which we have not found a supporting d-edge, we can reduce the number

of candidates for the left or right endpoints of the supporting d-edge. The dashed parts of the dS-layers are those, which can be eliminated from further consideration for containing a supporting point. If $si(p_i^l, p_j^r) \neq SI(p_i^l, p_j^r)$, we classify the d-edge $e(p_i^l, p_j^r)$ with respect to both intersection endpoints and eliminate the parts of the S-layers given by the two classifications. For both intersection endpoints we have nine possible cases, which are illustrated in Figure 7 for si-classification. Since the table for SI-classification is the same as for si-classification and since the dashed parts of the dS-layers, which can be eliminated from further considerations, are also the same, we only consider the si-classification.

(p_i^l, p_j^r) = (si-concave, si-supporting): In this case the set $\{e(p_f^l, p_g^r) | f < i \wedge 1 \leq g \leq t\}$ of d-edges cannot contain a supporting d-edge, because $e(p_i^l, p_j^r)$ lies below or on these d-edges in the range spanned by the x-coordinates of the dS-layer $DL(P_{CE}^{\eta})$ and, hence, all the d-edges in the above set are also si-concave in the left endpoint. Therefore the point set $\{p_f^l | f \leq i\}$ can be removed from the candidate list of left supporting points.

(p_i^l, p_j^r) = (si-supporting, si-concave): In this case the same argument as in the case (si-concave, si-supporting) implies, that we can remove the point set $\{p_g^r | g \geq j\}$ from the candidate set of right supporting points.

(p_i^l, p_j^r) = (si-concave, si-reflex): The fact that the d-edge $e(p_i^l, p_j^r)$ is si-reflex in p_j^r, implies $p_{j-1}^r \notin RS(si(p_i^l, p_j^r))$. Hence all points of the point set $\{p_g^r | g < j\}$ must lie outside the rotated silo $RS(si(p_i^l, p_j^r))$. For any d-edge $e(p_f^l, p_g^r)$, where $f \in \{1, \cdots, s\}$ and $g < j$, there exists always a part of this d-edge, which lies strictly above $e(p_i^l, p_j^r)$. Therefore no d-edge of this set can be supporting. Hence we can remove the point set $\{p_g^r | g < j\}$ from the candidate list of right supporting points.

(p_i^l, p_j^r) = (si-reflex, si-concave): The same argument as in the case (si-concave, si-reflex) implies that we can remove the point set $\{p_f^l | f > i\}$ from the candidate list of left supporting points.

(p_i^l, p_j^r) = (si-reflex, si-reflex): Using again the same argument as in the last two cases, we can remove the point set $\{p_f^l | f > i\}$ from the candidate list of left supporting points and the point set $\{p_g^r | g < j\}$ from the candidate list of right supporting points.

(p_i^l, p_j^r) = (si-supporting, si-reflex): We can remove the point set $\{p_g^r | g < j\}$ from the candidate list of right supporting points.

(p_i^l, p_j^r) = (si-reflex, si-supporting): We can remove the point set $\{p_f^l | f > i\}$ from the candidate list of left supporting points.

(p_i^l, p_j^r) = (si-concave, si-concave): This is the difficult case. Let l_1, l_2 be the vertical lines with $(l_1)_x = (p_i^l)_x$ and $(l_2)_x = (p_1^r)_x$. Let further

$$\Delta = bd(RS(SI(p_i^l, p_{i+1}^l))) \cap bd(RS(si(p_{j-1}^r, p_j^r))).$$

There are three different cases:

- Case 1: $\Delta = \emptyset$.
- Case 2: $\Delta = (\Delta_x, \Delta_y)$ is a point or a vertical line segment.
- Case 3: Δ is a line segment with startpoint $a = (a_x, b_x)$ and endpoint $b = (b_x, b_y)$ $(a_x < b_x)$.

Case 1: $\Delta = \emptyset$. Consider Figure 8. If $bd(RS(SI(p_i^l, p_{i+1}^l)))$ does not intersect line l_2, any rotated silo, whose boundary contains a point of the set $\{p_f^l | f \leq i\}$ and whose interior does not contain a point of the set P_{CE}^{η}, lies on the left side of l_2. Hence the boundary of such a rotated silo does not touch the shaded right region, which contains all points of the set

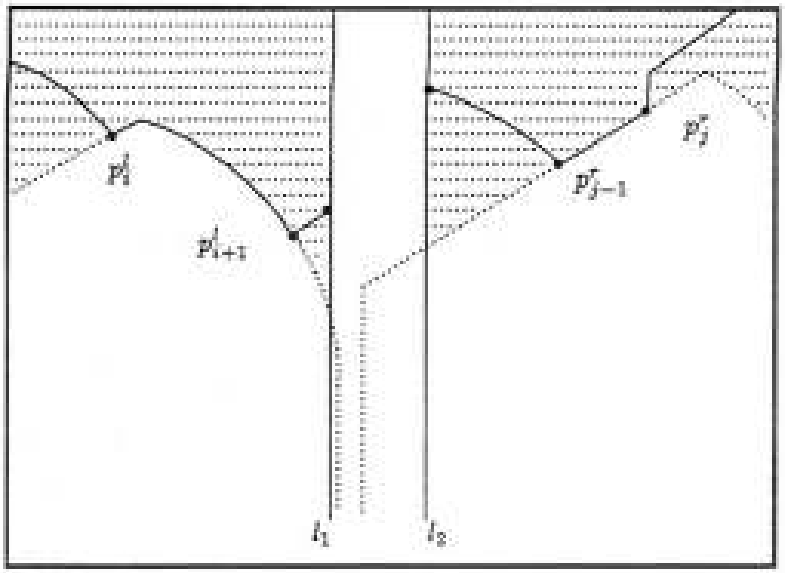


Figure 8: A possible situation in Case 1.

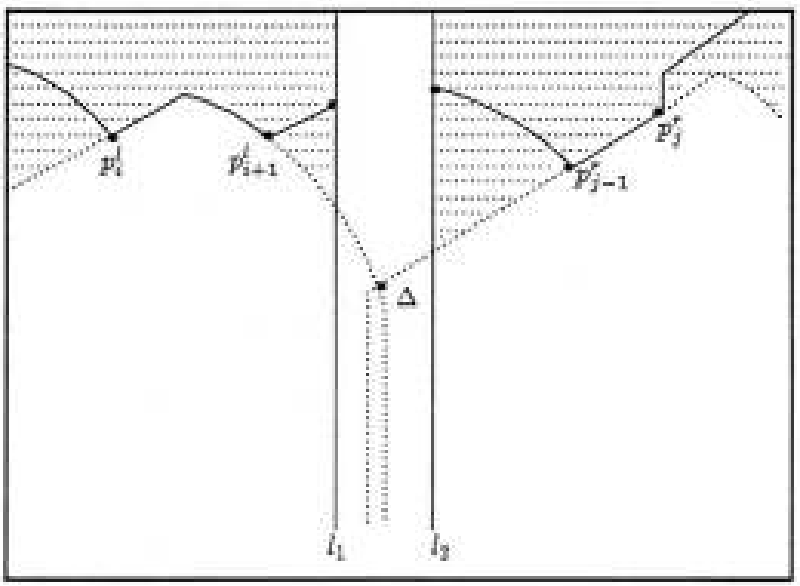


Figure 9: Example for Case 2b.

$P_{CH}^{[r]}$. Therefore the set $\{p_f^l | f \leq i\}$ cannot contain a left endpoint of a supporting d-edge. If $bd(RS(si(p_{j-1}^r, p_j^r)))$ does not intersect l_1, we can eliminate the set $\{p_g^r | g \geq j\}$ from the candidate list of right "supporting points".

Case 2: $\Delta = (\Delta_x, \Delta_y)$ is a point or a vertical line segment with x-coordinate Δ_x. We distinguish three subcases:
Case 2a: If $\Delta_x \leq (l_1)_x$, an argument similiar to Case 1 implies, that the set $\{p_f^l | f \leq i\}$ cannot contain a left endpoint of a supporting d-edge.
Case 2b: See Figure 9. If $(l_1)_x < \Delta_x < (l_2)_x$, the set $\{p_f^l | f \leq i\}$ does not contain a left supporting point and the set $\{p_g^r | g \geq j\}$ does not contain a right supporting point.

Case 2c: If $\Delta_x \geq (l_2)_x$, then the set $\{p_g^r | g \geq j\}$ does not contain a right supporting point.

Case 3: If Cases 1 and 2 do not apply, then Δ is a line segment with startpoint $a = (a_x, a_y)$

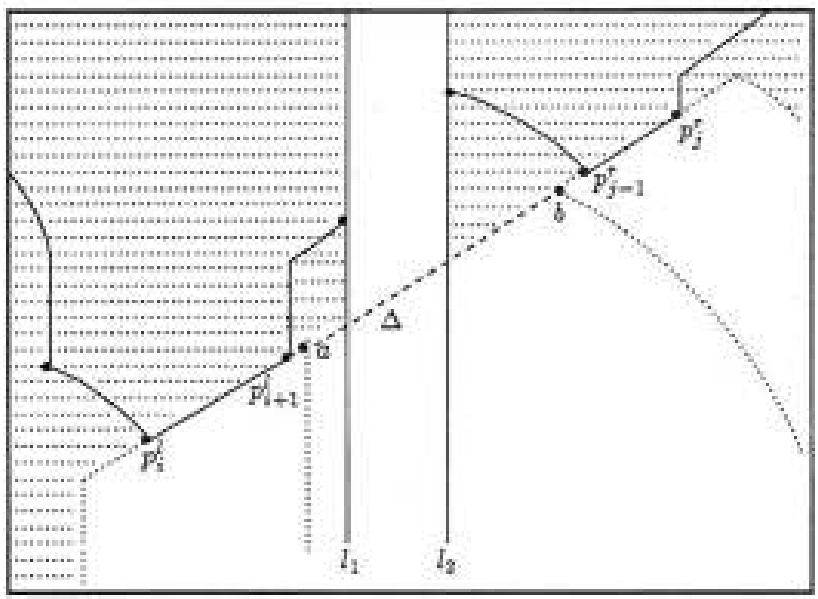


Figure 10: Example for Case 3c.

and endpoint $b = (b_x, b_y)$, where $a_x < b_x$. We distinguish again three subcases:
Case 3a: If $b_x < (l_2)_x$, then the set $\{p_g^r | g \geq j\}$ does not contain a right supporting point.
Case 3b: If $a_x > (l_1)_x$, then the set $\{p_f^l | f \leq i\}$ does not contain a left supporting point.
Case 3c: $b_x \geq (l_2)_x$ and $a_x \leq (l_1)_x$. Assume $e(p_f^l, p_g^r)$ with $g > j$ is a supporting d-edge. Since $(p_i^l, p_j^r) = (\text{si-concave, si-concave})$, p_i^l must lie outside the rotated silo $RS(si(p_{j-1}^r, p_j^r))$. Therefore, the part of $bd(RS(si(p_{j-1}^r, p_j^r)))$ lying to the left of the vertical line through a is contained in $int(RS(SI(p_i^l, p_{i+1}^l)))$. Hence the index f must be greater or equal to i and p_f^l must lie on that part of the line segment from a to b, that is to the left of l_1 (see Figure 10). The fact that all rotated silos are translates of a convex figure, implies the following statement: Consider any rotated silo, whose interior does not contain a point of the set $P_{CS}^{(r)}$ and whose boundary contains the point p_g^r. The part of the boundary of such a rotated silo, lying to the left of the vertical line through p_{j-1}^r, is contained in $int(RS(si(p_{j-1}^r, p_j^r)))$. Hence the boundary of such a silo does not touch the shaded left region, in which all points of the left dr-system lie. Thus p_g^r cannot be the right endpoint of a supporting d-edge and we get a contradiction. Therefore we can eliminate the set $\{p_g^r | g > j\}$ from the candidate list of right supporting points. The same argument implies, that the set $\{p_f^l | f < i\}$ does not contain a left supporting point.

In all cases, in which we do not find a supporting d-edge, a portion of one or both dr-systems can be eliminated. By testing always a pair of points lying in the "middle" of the remaining chains, we can find a supporting d-edge in $O(\log s + \log t)$ time. ∎

We are now able to describe the whole dS-layer construction algorithm, which is a slightly modified version of the (lower) convex layer construction algorithm of Overmars and van Leeuwen [10]. We store the point set $P_{CS} = \{p_1, \cdots, p_m\}$ in the leaves of an augmented balanced binary search tree T, sorted by their x-coordinates. This tree T is identical to the

dynamic data structure used in [10]. Let u denote a node of T with left son v and right son w. Let $P(v)$ (resp. $P(w)$) be the points of P stored in the subtree rooted at v (resp. w). During the construction of tree T we determine a minimal dr-system of $DL(P(u))$. We assume that a concatenable queue Q_v is associated to node v, in which the chain of a minimal dr-system of $DL(P(v))$ is stored, and that a concatenable queue Q_w is associated to node w, in which the chain of a minimal dr-system of $DL(P(w))$ is stored. The concatenable queues enable us to carry out the following operations in $O(\log m)$ time:

- to locate the supporting points using Lemma 7,
- to split the chains associated to v (or w) in the fragment, which belongs to the chain of u, and the remaining subchain,
- to concatenate the fragments of left and right minimal dr-system to the concatenable queue, which belongs to the computed minimal dr-system $DR(P(u))$.

The concatenable queue, which belongs to the minimal dr-system $DR(P(u))$, is associated to the node u. Furthermore we store information about the "bridge", which connects the two original parts of the concatenable queue. The remaining subchains are associated to v and w. Besides we store the bridge between the two fragments of Q_v in node v and the bridge between the two fragments of Q_w in node w. For details about information stored in the nodes of tree T, see [10]. The tree T uses $O(m)$ space.

dS-layer construction algorithm

(1) Construct the augmented tree T;
(2) $z := |P_{CE}|$;
(3) $P_T := P_{CE}$;
(4) **while** $(z \neq 0)$ **do**

Determine and store the minimal dr-system $DR(P_T)$ associated to $root(T)$;
Remove the points represented by $DR(P_T)$ one by one from T;
$z := z - |DR(P_T)|$;
$P_T := P_T \setminus DR(P_T)$

od

The tree construction in Step 1 takes $O(m \log m)$ time. Determing the minimal dr-system and storing this minimal dr-system can be done in $O(|DR(P_T)|)$ time. Removing the points represented by $DR(P_T)$ costs $O(|DR(P_T)|(\log m)^2)$ time. Thus the whole while-loop can be carried out in $O(m(\log m)^2)$ time. Hence the whole dS-layers construction for cell CE with $|P_{CE}| = m$ can be done in $O(m(\log m)^2)$ time.

We summmarize now:

Theorem 2 *Let P be a set of n points in the Euclidean plane E^2. There exists a data structure of size $O(n)$, such that $O(\log n + k)$ time suffices to retrieve all k points of P lying inside a query translate C_q of a convex computable figure C. The data structure can be constructed in $O(n(\log n)^2)$ time.*

Proof: Decomposing the Euclidean plane into cells, distributing the points of P in their corresponding cells and storing the non-empty cells in sorted order in a binary search tree

can be done in $O(n \log n)$ time. Then we have to construct the four query data structures for every non-empty cell CE. If $|P_{CE}| = m$, the construction of a family of d$*$-layers ($* \in \{N, S, O, W\}$) costs $O(m(\log m)^2)$ time. Building the four query data structures for cell CE with the help of these dual layer families can be done in $O(m)$ operations. Hence the preprocessing for all non-empty cells can be carried out in $O(n(\log n)^2)$ time.

Since the new data structures, which we use to construct the dual layers, has size $O(n)$, all data structures together use $O(n)$ space. The query time is the same as in [6], because we use the same query data structure. ∎

It is worthwhile to mention that we also get dynamic data structures for this class of retrieval problems, because we can use the dynamic data structure T to search for all points of cell CE in a query translate C_q, which is S-grounded with respect to CE. Insertions and deletions in the dynamic data structure T can be done in $O((\log |P_T|)^2)$ time, where P_T is the point set stored in T (see [10]). We describe now a simple way to find all points stored in T, which lie in the query translate C_q: Search the edge of the S-layer $L(P_T)$, which lies below or above the query point q. This search can be carried out in the concatenable queue associated to the root of T with $O(\log |P_T|)$ operations. If the point p, which belongs to the above edge, does not lie in C_q, we are ready in this cell. If the point lies in C_q, we store p in a queue called REMEMBER, delete p from T and search again. We continue to do this until we have found all points $p \in C_q$ stored in the original tree T. Afterwards we restore T by inserting all points $p \in$ REMEMBER in T again. This retrieval operation takes $O(\log |P_T| + k(\log |P_T|)^2)$ time, where k is the number of points stored in P_T, which lie in C_q.

Theorem 3 *Let P be a set of n points in E^2 and C a convex computable figure. There exists a dynamic data structure which stores the point set P and which uses $O(n)$ space, such that $O(\log n + k(\log n)^2)$ time suffices to retrieve all k points of P lying inside a query translate C_q. The data structure can be dynamically maintained at a worst-case cost of $O((\log n)^2)$ per insertion and deletion.*

3.3 An improved dS-layers construction algorithm

Let $P_{CE} = P \cap CE = \{p_1, \cdots, p_m\}$ be the sequence of points in cell CE, sorted in order of increasing x-coordinates. We assume wlog that there are no two points in P_{CE} with the same x-coordinate. We consider again only S-grounded queries.

In the last subsection we saw that we can construct the augmented balanced binary search tree T for P_{CE} in $O(m \log m)$ time. In Step 4 of the dS-layer construction algorithm, we delete points from T. Each deletion costs $O((\log m)^2)$ time. Hence all deletions together can be done in $O(m(\log m)^2)$ time. Chazelle showed in [3] that the deletions involved in the computation of the convex layers can be batched together, such that all deletions can be done in $O(m \log m)$ time. We prove now, that we can use a slightly modified version of Chazelle's convex layer construction algorithm to reduce the preprocessing time for our data structure to $O(m \log m)$.

We store the points $p_1, \cdots, p_m$ in the leaves of the balanced binary search tree T presented in the last subsection. The subset of points stored at the leaves of a subtree $T(u)$ with root u, is denoted by $P(u)$. Let $DL(P(u))$ be the dS-layer of $P(u)$ and $DR(P(u))$ a minimal dr-system of $DL(P(u))$. Connecting P_{CE} by the set

$$E_d = \bigcup_{u \in T} \{e(a,b) | a, b \text{ neighboring points in } DR(P(u))\}$$

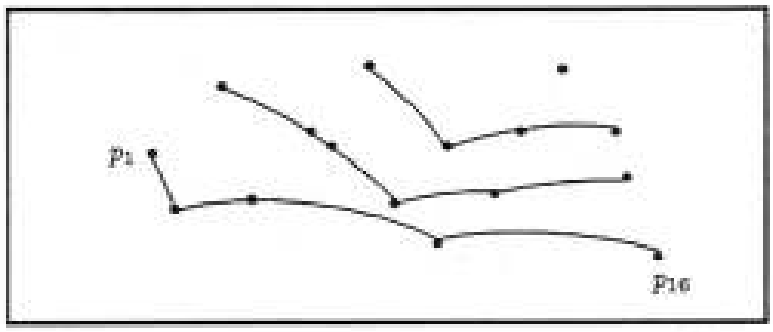


Figure 11: Dual S-layers of 16 points $\{p_1, \cdots, p_{16}\}$. The corresponding figure C is a disk.

of d-edges, we get a planar embedding of the graph $G = (V, \mathcal{E})$ with nodes $V = P_{CS}$ and edges $\mathcal{E} = \{(a, b) | e(a, b) \in \mathcal{E}_d\}$. The connected acyclic planar graph G is called the *dS-graph* of P_{CS}. We use the notation G also for the two-dimensional embedding of the graph. The d-edges of G are in one-to-one correspondence with the nodes of the tree T (see Figure 12). Each node $u \in T$ corresponds to the "longest" supporting d-edge of G, which connects the dS-layers of node u's children. Assume v and w are the children of node $u \in T$, then u

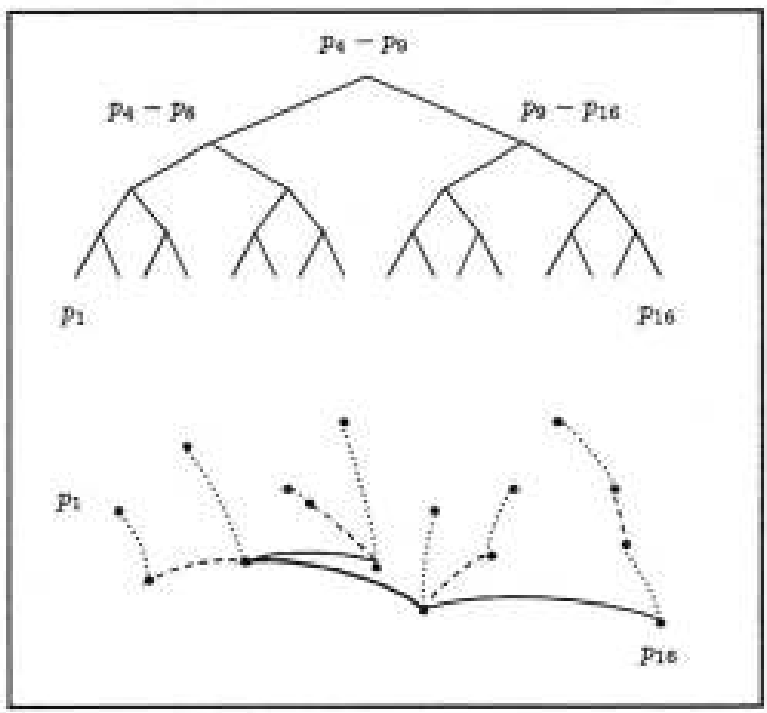


Figure 12: Tree T and the dS-graph for the point set of Figure 11.

corresponds to the "longest" supporting d-edge of $DL(P(v))$ and $DL(P(w))$.

The dS-graph G is represented by an adjacency list structure. We endow each vertex $p \in P_{CS}$ with a list $V(p)$, which contains the names of the adjacent vertices. Each list $V(p)$ consists of two sublists $VL(p)$ and $VR(p)$, defined as follows: $VL(p)$ (resp. $VR(p)$) contains the vertices adjacent to p, which have smaller (resp. larger) x-coordinates than p. The points in $VL(p)$ and $VR(p)$ are sorted with respect to the corresponding d-edges from bottom d-edge to top d-edge. Each vertex p has a pointer to the bottom d-edge of $VL(p)$

and a pointer to the bottom d-edge of $VR(p)$ (see Figure 13). The sorting of the d-edges in $VL(p)$ and $VR(p)$ can be carried out by considering the intersection points of the d-edges with vertical lines near vertex p. If we have d-edges $e(a,p)$ and $e(b,p)$, which deliver the same intersection point with the selected vertical line, we sort these d-edges by considering the x-coordinates of the corresponding points a and b.

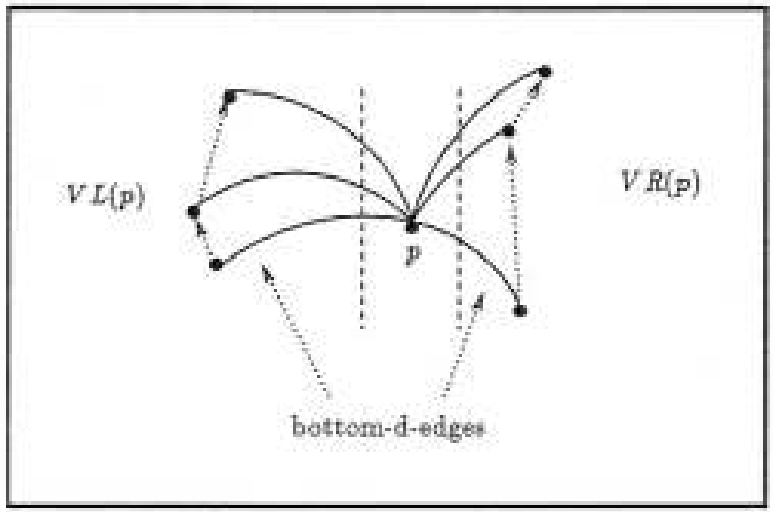


Figure 13: Adjacency list structure for vertex p and the bottom d-edges.

First we describe one simple way to compute the dS-graph G in $O(m \log m)$ time. We assume that we know the dS-graphs of $\{p_1, \cdots, p_{\lfloor m/2 \rfloor}\}$ and $\{p_{\lfloor m/2 \rfloor + 1}, \cdots, p_m\}$. Using the extra pointers to the bottom d-edges, we can easily run across the chains $\{p_1^l, \cdots, p_s^l\}$ and $\{p_1^r, \cdots, p_t^r\}$ of the computed dr-systems. In order to compute a "longest" supporting d-edge e_{l_s} for the two dS-layers, we go ahead as follows: We set $e_{l_s} = e(p_1^l, p_j^r)$ for $j = 1, 2, ...$, until we get a d-edge $e(p_1^l, p_j^r)$, which is supporting in p_j^r. If $e(p_1^l, p_j^r)$ is also supporting in p_1^l, we use the procedure introduced in Subsection 2.2 to determine the "longest" supporting d-edge and add this d-edge to the dS-graph G. If the d-edge $e(p_1^l, p_j^r)$ is not supporting in p_1^l, we move the first endpoint p_1^l to $p_2^l, p_3^l \cdots$, until we find a point p_i^l, such that $e(p_i^l, p_j^r)$ is supporting in p_i^l. The d-edge $e(p_i^l, p_j^r)$ is now supporting in p_i^l, but in general not in p_j^r. If the d-edge is not supporting in p_j^r, we move again the right endpoint p_j^r to $p_{j+1}^r, p_{j+2}^r, \cdots$. In this way we move both endpoints around their dS-layers until we find a "longest" supporting d-edge e_{l_s}. Since this construction method requires $O(m)$ operations for every level of the tree T, the whole dS-graph can be constructed in $O(m \log m)$ time.

Starting in the leftmost or rightmost vertex of the dS-graph G and following the extra pointers to the bottom d-edges, we can find a minimal dr-system $DR(P_{CB})$ for $DL(P_{CB})$. We store this dr-system and then we remove all points $p \in DR(P_{CB})$ from the dS-graph G.

Before we describe in detail how points will be deleted from G, we briefly introduce the geometrical concept, on which Chazelle's deletion method is based. In order to remove the vertex p from G and in order to reshape the dS-graph G, we move the vertex $p = (p_x, p_y)$ on the vertical ray $l_p := \{(p_x, y) | y \geq p_y\}$ towards $y = \infty$. By moving the point towards $y = \infty$, the d-edges adjacent to p will be pulled up and will be removed one by one. The d-edges adjacent to p have to be considered in the order in which they appear as supporting d-edges in the path from leaf p to the root of T (see Figure 14).

The deletion of a point p lying on the current dS-layer will now be described in detail. Let

$v_1, \cdots, v_l$ be the nodes of T, which lie on the path from leaf p to the root of T. Every node v_i corresponds to a "longest" supporting d-edge of two dS-layers. Since p lies in the current dS-layer of all points present in G, p lies on one of these two dS-layers. Let $w_1, \cdots, w_k$ be the subsequence of $v_1, \cdots, v_l$, that corresponds to supporting d-edges with p as an endpoint (see Figure 14). We distinguish between *p-left supporting* d-edges $e(w_i, p)$, where $w_i \in VL(p)$,

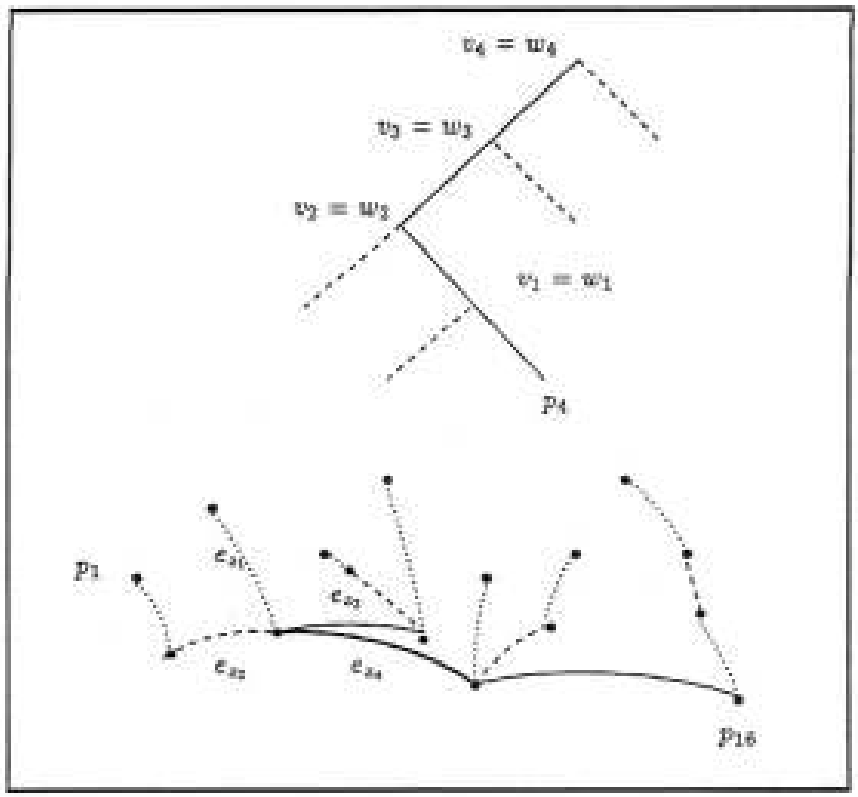


Figure 14: Leaf-to-root path for point p_4 of Figure 11 and the corresponding d-edges in the dS-graph.

and *p-right supporting* d-edges $e(p, w_j)$, where $w_j \in VR(p)$. Let $e_{z_1}, \cdots, e_{z_k}$ be the sequence of supporting d-edges corresponding to the sequence $w_1, \cdots, w_k$ of nodes. Since the d-edges with endpoint p have to be removed in this "leaf-to-root" order, we have to merge the lists $VL(p)$ and $VR(p)$. This merging can be done in $O(l)$ steps.

For any node w_i let $G(w_i)$ denote the dS-subgraph of G, that belongs to the subtree of T rooted at w_i. In order to remove p from G, we have to update the sequence of dS-graphs $G(w_1), \cdots, G(w_k)$ in this order. We assume that we have already removed p from the dS-graphs $G(w_1), \cdots, G(w_{i-1})$ and we want to update $G(w_i)$. Furthermore we assume wlog, that the supporting d-edge $e(p, c)$ corresponding to w_i is a p-right supporting d-edge. Let $e(a, p)$ and $e(p, b)$ be the last p-left supporting d-edge and the last p-right supporting d-edge, which we have pulled up. Without loss of generality we can assume, that $e(a, p)$ and $e(p, b)$ exist. In this situation the new part of $G(w_{i-1})$ lies between a and b above the composed curve $(e(a, p), e(p, b))$ (see Figure 15).

Let $e(a, a'), \cdots, e(b', b)$ be the sequence of d-edges between a and b, which lie on the dS-layer of all points represented in $G(w_{i-1})$. Hence a' is the vertex of the current dS-layer $DL(P(w_{i-1}))$ following a in counterclockwise order and b' is the vertex following b in clockwise order. Let c' denote the vertex of the right dS-subgraph following c in clockwise order (see

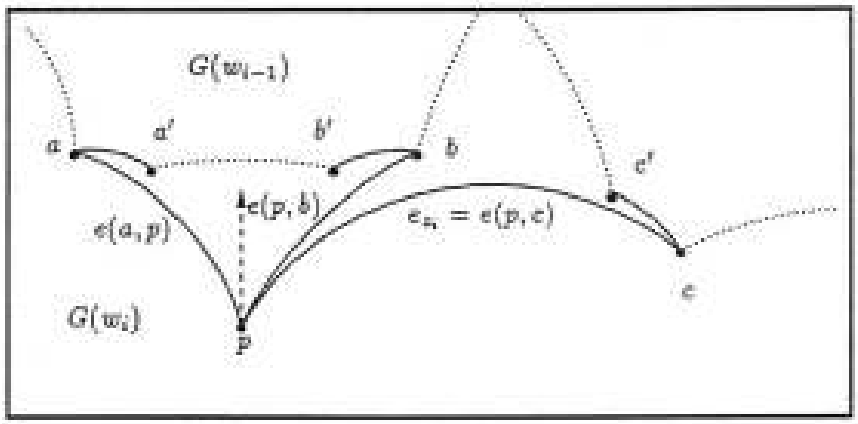


Figure 15: Schematical illustration of the deletion process.

Figure 15). Updating $G(w_i)$ means pulling up the vertex p until it disappears from the dS-layer $DL(P(w_i))$. During this process we stop at every point on the vertical line l_p, where a supporting d-edge has to be exchanged by another "longest" supporting d-edge, and change the dS-graph.

Let IS_a, IS_b, IS_c be the intersection points of the vertical line from p towards $y = \infty$ with the three boundaries $bd(RS(SI(a,a')))$, $bd(RS(si(b',b)))$ and $bd(RS(si(c',c)))$. (We consider $SI(\ ,\ )$, if both points lie to the left of l_p, and $si(\ ,\ )$, if both points lie to the right of l_p. If one point lies to the left and the other to the right of l_p, we can choose $si(\ ,\ )$ or $SI(\ ,\ )$. If the intersection is a line segment, we choose the point with maximal y-coordinate.) The first placement for p, where a supporting d-edge has to be exchanged, is that point of these three intersection points, which has minimal y-coordinate. Therefore we compute the three intersection points and sort them in order of increasing y-coordinates. If for example IS_c is the intersection point with smallest y-coordinate, the point p reaches first IS_c on his way towards $y = \infty$. At this point we have to replace the d-edge $e(p,c)$ by the d-edge $e(p,c')$ in the dS-graph. Hence we replace c by c', compute the intersection point $IS_{c'}$ of the

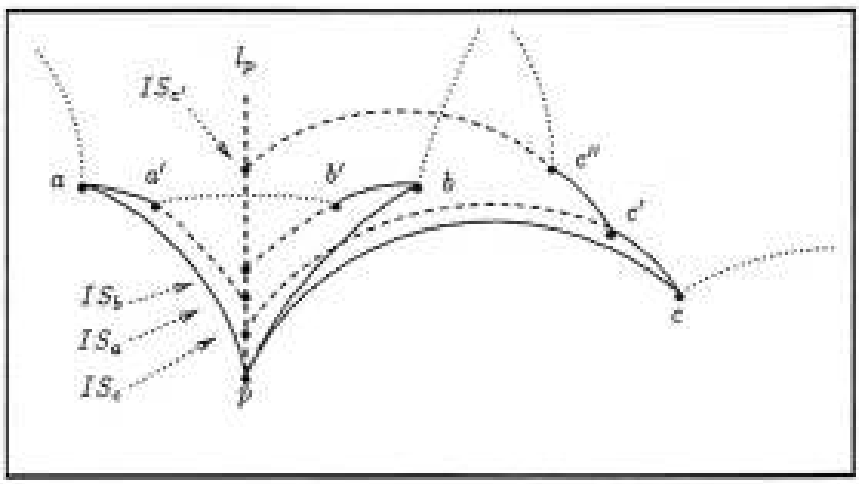


Figure 16: Determing the next placement for point p.

vertical line through p with the boundary $bd(RS(si(c',c'')))$ and insert the new intersection point in the sorted sequence of intersection points (see Figure 16). The first element in this

sorted sequence always determines the next placement for p. At every placement of p with corresponding endpoints a, b, c we have to test, if $e(a, c)$ or $e(b, c)$ is a supporting d-edge for the two dS-layers. If we have found a "longest" supporting d-edge, we change the dS-graph $G(w_i)$ and start to remove p from $G(w_{i+1})$. In this way we handle all supporting d-edges with endpoint p. If $v_i \notin \{w_1, \cdots, w_k\}$, then p is not an endpoint of the supporting d-edge associated to v_i. Therefore this supporting d-edge does not change, and no additional work is required.

In this way we remove all points $p \in DR(P_{CE})$ from the dS-graph G. Then we determine a minimal dr-system $DR(P^2_{CE})$ of the dS-layer $DL(P^2_{CE})$, where $P^2_{CE} := P_{CE} \setminus DR(P_{CE})$. This can be done by running across the path of the current dS-graph G, which starts in the leftmost or rightmost vertex, and following the extra pointers to the bottom d-edges. After we have determined and stored $DR(P^2_{CE})$, we remove all points in $DR(P^2_{CE})$ from the dS-graph G. Then we determine a minimal dr-system $DR(P^3_{CE})$ of the dS-layer $DL(P^3_{CE})$, where $P^3_{CE} := P^2_{CE} \setminus DR(P^2_{CE})$. We continue in this way, until we get $P^s_{CE} = \emptyset$.

It can easily be seen that the above deletion procedure and the whole dS-layer construction algorithm work correctly. We leave out here the complexity analysis, because it is almost identical to that given in [3].

We summarize now:

Lemma 8 *A family of minimal dr-systems for the set P_{CE} with $|P_{CE}| = m$ can be computed in $O(m \log m)$ time using $O(m)$ space.*

Lemma 8 implies that the whole preprocessing for the entire data structure requires $O(n \log n)$ operations. Hence, we get the main result of this paper:

Theorem 4 *Let P be a set of n points in E^2 and C a convex computable figure. There exists a data structure that stores the point set P, such that $O(\log n + k)$ time suffices to retrieve all k points lying inside a query translate C_q. The data structure has size $O(n)$ and can be constructed in $O(n \log n)$ time.*

We close this subsection with a few examples of convex computable figures C.

In the case C is a disk, we don't need to decompose C into upper and lower parts, because both parts lead to the same decomposition of E^2.

Corollary 1 *Let P be a set of n points in the Euclidean plane E^2 and let C be a disk. In $O(n \log n)$ time we can preprocess P so that for any query point q, all k points of P, which lie in the query translate C_q, can be retrieved in $O(\log n + k)$ time. The query data structure has size $O(n)$.*

Other shapes which C may assume, are for example triangles, rectangles, ellipses or hybrid convex figures bounded by a constant number of analytic curves.

Our method also works for non-bounded convex computable figures. In such cases we only have to modify the decomposition of the Euclidean plane. If C is a hyberbola for example, we only have one single cell: the whole Euclidean plane. After a suitable basis transformation we need only one single query data structure, for example the query data structure for S-grounded queries.

If C is a convex m-gon, the primitive operations like intersection computing and point-inside-or-outside-test can be done in $O(\log m)$ time.

Corollary 2 *Let P be a set of n points in the Euclidean plane and let C be a convex m-gon. In $O(n \log n \log m + m \log m)$ time we can preprocess P so that for any query point q, all k points of P, which lie in the query translate C_q, can be retrieved in $O(\log n + (k+1) \log m)$ time. The query data structure has size $O(n + m)$.*

The algorithm uses only $O(n+m)$ space, because vertices of the dS-graph only contain pointers to their adjacent vertices, and not to the sequence of polygon edges. A similar result was given by Klein et al.[9]. Their query algorithm, however, has running time $O(m \log n + k)$.

4 Some concluding remarks

1) How to handle points with the same x-coordinate?
If there are points with the same x-coordinate, we sort these points in order of increasing y-coordinates and store the sorted sequence of points in a queue. Hence, for every x-coordinate, there is one queue. When we start the computation of the family of dS-layers for S-grounded queries, we take the first point from every queue and start the construction with this point set P^*. Whenever a point $p \in P^*$ has become an element of a computed minimal dr-system, we remove p from P^*, we pick up the next point of the queue (if the queue is not empty) to which p has belonged, and we include this point into P^*. Here, including into P^* means that we insert this point in the data structures used in the preprocessing.

These additional sorting, inserting and deleting operations do not change space or preprocessing time bounds.

2) Faster dynamic data structures
The static data structure of Theorem 4 has a building time $P(n) = O(n \log n)$, size $S(n) = O(n)$, and a query time $Q(n) = O(\log n + k)$. Since the point retrieval problem is a decomposable searching problem, we can apply the techniques of Dobkin and Suri [8], and Smid [12]. This gives data structures that can handle semi-online updates in $O(\frac{P(n)}{n} \log n) = O((\log n)^2)$ amortized time. These data structure still have a building time of $O(n \log n)$ and a size of $O(n)$. The query time becomes $O(Q(n) \log n) = O((\log n)^2 + k)$, which is better than that of Theorem 3.

Using fractional cascading, the query time can be improved to $O(\log n + k)$: The data structure consists of $O(\log n)$ static structures for sets of sizes $O(2^i)$, $i = 0, 1, \ldots, \log n$. To answer a query, we query each static structure separately. Each such query starts with a binary search in the outermost layer. Since for each binary search the query point is the same, we can connect these outermost layers using fractional cascading. More precisely, we copy elements from a structure for $O(2^i)$ points to a structure for $O(2^{i-1})$ points, for $i = \log n, \ldots, 1$. The search starts in the structure for $O(2^0)$ points. Once we have located the query point in the $(i-1)$-th structure we use the fractional cascading information to locate the point in the i-th structure. (See [7] for details.)

During an update, we rebuild static structures for sets of sizes $2^0, \ldots, 2^l$, for some l. This rebuilding takes time $O(2^l \log 2^l)$. Then we copy elements from the i-th structure to the $(i-1)$-th structure, for $i = l+1, l, \ldots, 1$, and construct the fractional cascading information for these new static structures. This takes only $O(2^l)$ time. Hence, the amortized update time for this improved structure remains bounded by $O((\log n)^2)$.

3) Collision tests for molecules
We give an application of our result in the field of molecular simulation in computational chemistry. Testing whether and when a molecule A in motion collides with another fixed molecule B are two important problems, which arise in molecular modelling or docking simulation. Especially the following problems are of interest:

- *Trans*(A, B): Given a translation direction for molecule A, test whether molecule A collides with the fixed molecule B. If A collides with B, then determine, when the collision takes place.

- *Rot*(A, B): Given a rotation axe and a (rotation) direction for molecule A, test whether molecule A collides with the fixed molecule B. If A collides with B, then determine, when the collision takes place.

We use the following "atom model": Each atom a of a molecule is a sphere with center $c(a) = (a_x, a_y, a_z)$ and radius r_a, where r_a is the van der Waals radius of atom a. Let A and B be two molecules, which consists of n and m atoms resp. We assume for simplicity that all n atoms of molecule A have radius r_A and all m atoms of molecule B have radius r_B. Furthermore we assume that the shapes of the molecules are fixed. (Hence we don't allow distorsions of the shapes, our model supports only docking simulation based on the lock-and-key model. It would be desirable of course, to have molecular dynamics tools, which could carry out energy minimization and could determine possible new shapes for the molecules, if the molecules are so close to each other, that they influence each other. The best tool would be an algorithm, that carries out the docking process with fitting of the shapes automatically. The user has to place the molecules, such that the possible active sites are close together. Then the algorithm guides the molecules in a local energy minimum, in which perhaps molecule A has docked at the active side of molecule B. Such an algorithm would also support the induced fit model.). Since there are lower bounds for the minimal distance between to different atoms, a sphere with fixed radius can contain only a constant number of atoms or atomic nuclei.

We now consider the problem *Trans*(A, B). Instead of determing collision between atoms in A and B, we test for collisions between the centers of all n atoms in A and the spheres, which we obtain, by blowing up every atom in B to a sphere with radius $r_A + r_B$.

Let pr be the projection parallel to the translation direction into a fixed plane, which is orthogonal to the translation direction. In order to test for collision, we carry out a space sweep in the translation direction with a plane E, which is orthogonal to the translation direction. First we sort all atoms of A and B with respect to the following order "$<$": $a < \bar{a}$, if the sweep plane reaches the center of atom a before it reaches the center of $\bar{a}$. We move the plane in the translation direction and we stop, when the plane reaches a center of any atom. If we stop at the center of an atom $a \in A$, we insert $pr(c(a))$ in the dynamic point retrieval query structure of Theorem 3, where we take for C a disk with radius $r_A + r_B$. If we stop at the center of an atom $b \in B$, we do the following test: We test whether there is a point in the dynamic data structure, which lies in the circle with radius $r_A + r_B$ and center $pr(c(b))$. If we find a point, which lies in this circle, molecule A will collide with molecule B.

Sorting the points in the translation direction takes $O((n+m)\log(n+m))$ operations. The space sweep and the construction of the query data structure costs $O(n(\log n)^2 + m\log n)$ time. Hence the whole collision test takes $O(n(\log n)^2 + (n+m)\log(n+m))$ time.

If we want to determine when A collides with B, we have to modify the above algorithm slightly. Whenever we stop at a center of an atom $b \in B$ during the sweep, we have to retrieve all atoms A_b of A, whose projected centers lie in the sphere with radius $r_A + r_B$ and center $pr(c(b))$. For all these atoms of A we determine, when they collide with atom b. Besides we have to determine the collision points (if existing) of all pairs (a, b^*), where $a \in A_b$ and $b^* \in B_b := \{\bar{b} \in B \mid |c(\bar{b}) - c(b)| \leq 2(r_A + r_B)\}$. We compare every computed collision point with the minimal one we have found until now. If we find a new minimum, we store the corresponding collision atom pair and the new minimum. After all these tests have been done, we remove the point set corresponding to A_b from the query data structure and go to the next atom in the sweep direction.

Note that there are only a constant number of atoms in B_b. Hence, if we know the set B_b, the whole operation for atom b can be done in $O(|A_b|(\log n)^2)$ time. If we assume that

we know the sets B_b for all $b \in B$, the space sweep can be carried out in $O(n(\log n)^2 + (n+m)\log(n+m))$ time. (We only have to compute the sets B_b once. Computing the distances between all pairs of atoms in B and determing a neighbor list for each atom is the simplest way to get the sets B_b. But there are a few more efficient methods to determine the sets B_b.)

Note that the collision detection for rotations can be handled in the same way and that we get the same upper bounds as in the case of translations. If we carry out a collision test for a rotation, we choose a sweep plane, which contains the rotation axe, and rotate this plane around this axe. The centers are sorted according to the rotation angles, when the rotated plane hits the centers. In this case the projection pr is the rotation around the rotation axe in a fixed plane, which contains the rotation axe.

References

[1] A. Aggarwal, M. Hansen and T. Leighton. *Solving Query-Retrieval Problems by Compacting Voronoi Diagrams.* PROC. OF THE 16TH ANNUAL SYMP. OF THEORY OF COMPUTING (1990), pp. 331-340.

[2] J. L. Bentley. *Decomposable Searching Problems.* INFORM. PROC. LETT. vol. 8 (1979), pp. 244-251.

[3] B. Chazelle. *On the Convex Layers of a Planar Set.* IEEE TRANSACTIONS ON INFORMATION THEORY, vol. 31, no. 4 (1985), pp. 509-517.

[4] B. Chazelle. *Filtering Search: A New Approach to Query-Answering.* SIAM J. OF COMP. vol. 15 (1986), pp. 703-724.

[5] B. Chazelle, R. Cole, F.P. Preparata and C. Yap. *New Upper Bounds for Neighbor Searching.* INFORMATION AND CONTROL, vol. 68 (1986), pp. 105-124.

[6] B. Chazelle and H. Edelsbrunner. *Optimal Solutions for a Class of Point Retrieval Problems.* J. SYMBOLIC COMPUTATION, vol. 1 (1985), pp. 47-56.

[7] B. Chazelle and L. Guibas. *Fractional Cascading: I, A Data Structuring Technique; II, Applications.* ALGORITHMICA, vol. 1 (1986), pp. 133-191.

[8] D. Dobkin and S. Suri. *Maintenance of Geometric Extrema.* JOURNAL OF THE ACM, vol. 38 (1991), pp. 275-298.

[9] R. Klein, O. Nurmi, T. Ottmann and D. Wood. *A Dynamic Fixed Windowing Problem.* ALGORITHMICA, vol. 4 (1989), pp. 535-550.

[10] M.H. Overmars and J. van Leeuwen. *Maintenance of Configurations in the Plane.* J. COMPUT. SYST. SCI. vol. 23 (1981), pp. 166-204.

[11] F.P. Preparata and M.I. Shamos. *Computational Geometry: an Introduction.* Springer-Verlag New York-Berlin-Heidelberg-Tokyo (1985).

[12] M. Smid. *Algorithms for Semi-online Updates on Decomposable Problems.* PROC. 2ND CANADIAN CONF. ON COMPUTATIONAL GEOMETRY, 1990, pp. 347-350.