125 143

rintingRequisition/GraphicServices

89018

ase complete unshaded areas on rm as applicable.

2. Distribute copies as follows: White and Yellow to Graphic Services. Retain Pink Copies for your records.

3. On completion of order the Yellow copy will be returned with the printed material.

4. Please direct enquiries, quoting requisition number and account number, to extension 3451.

LE OR DESCRIPTION
A Dynamic Fixed Windowing Problem CS-87-56

TE REQUISITIONED: Oct. 15/87

DATE REQUIRED: ASAP

ACCOUNT NO.: 1 2 6 6 1 7 6 4 1

EQUISITIONER - PRINT: Derick Wood

PHONE: x4456

SIGNING AUTHORITY: [illegible]

MAILING INFO — NAME: Sue DeAngelis DEPT.: Computer Science BLDG. & ROOM NO.: MC 6081E [✓ DELIVER] PICK-UP

pyright: I hereby agree to assume all responsibility and liability for any infringement of copyrights and/or patent rights which may arise from the processing of, and reproduction of, any of the materials herein requested. I further agree to indemnify and hold blameless the University of Waterloo from any liability which may arise from said processing or reproducing. I also acknowledge that materials processed as a result of this requisition are for educational use only.

UMBER F PAGES: 21 NUMBER OF COPIES: 150

YPE OF PAPER STOCK: BOND NCR ___ PT. [✓ COVER] BRISTOL SUPPLIED Alpac Ivory ~~140M~~

APER SIZE: 8½ x 11 8½ x 14 11 x 17 10x14 Glosscoat (stock) ~~10 pt.~~

APER COLOUR: WHITE ___ INK: [✓ BLACK] Rolland Tint

RINTING: 1 SIDE ___ PGS. [✓ 2 SIDES] ___ PGS. NUMBERING FROM ___ TO ___

INDING/FINISHING: COLLATING STAPLING HOLE PUNCHED PLASTIC RING

OLDING/ ADDING: 7x10 Saddle Stitched CUTTING SIZE:

pecial Instructions
Special Beaver Cover with Black Ink Format.

PY CENTRE — OPER. NO. / BLDG. / MACH. NO.

ESIGN & PASTE-UP

OPER. NO.	TIME	LABOUR CODE
3	20	D01
		D01
		D01

YPESETTING

	QUANTITY			
AP000001	1	00	12	T01
AP000000				T01
AP000000				T01

ROOF Dylux

PRF 1611800		3	10	P01
PRF				
PRF				

NEGATIVES

	QUANTITY	OPER. NO.	TIME	LABOUR CODE
FLM 0811000	1	2	10	C01
FLM				C01
FLM				C01
FLM				C01
FLM				C01

PMT

PMT				C01
PMT				C01
PMT				C01

PLATES

PLT 111800	1	5	20	P01
PLT 111700	12	11	24	P01
PLT				P01

STOCK

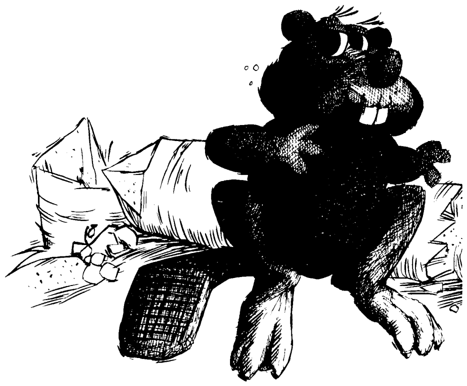
KRO 101400	180	16	20	001
ROL 101400	1080	11	90	001
				001
				001

BINDERY

RNG		1	5	B01
RNG		1	5	B01
RNG		93	15	B01
MIS000000				B01

OUTSIDE SERVICES

$ ___ COST



A Dynamic Fixed Windowing Problem

Rolf Klein
Otto Nurmi
Thomas Ottmann
Derick Wood

Data Structuring Group
Research Report CS-87-56

August 25, 1987

A Dynamic Fixed Windowing Problem [*]

Rolf Klein[†] Otto Nurmi[†] Thomas Ottmann[†]
Derick Wood[‡]

August 25, 1987

Abstract

Given a point set in the plane and a fixed planar region (window) a window query consists of enumerating the points in a translate of the region. A recently presented result demonstrates that there is a *static* data structure, of optimal size, that solves window queries for convex regions in optimal time. We give a data structure, of optimal size, that not only supports window queries in optimal time for, possibly non-convex, polygonal windows, but also allows updating of the point set in optimal time.

1 Introduction

Many new algorithms and data structures have been implemented in the still blossoming area of multidimensional searching and computational geometry. Nevertheless, efficient dynamic solutions for many multidimensional searching problems have still to be discovered. Here we assume that we are given some initial set of objects and we want not only to answer certain queries on the set of objects but also allow the user to insert new objects and delete existing objects from the set.

In this paper, we consider the *dynamic fixed windowing problem for point sets in the plane.* It can be posed as follows: For a given point set P in two-dimensional space and for a given fixed window W, find a data structure to represent P and design algorithms to carry out the following operations

[*]This work was partially supported by grant Ot64/4-2 from the Deutsche Forschungsgemeinschaft and partially under a Natural Sciences and Engineering Research Council of Canada Grant No. A-5692

[†]Institut für Angewandte Informatik und Formale Beschreibungsverfahren, Universität Karlsruhe, Postfach 6980, D-7500 Karlsruhe, West Germany.

[‡]Data Structuring Group, Department of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada.

efficiently: Insertion of points, deletion of points, window-queries, that is, for an arbitrary query translate $W_q = W + q$, report the points in $W_q \cap P$.

Recently Chazelle and Edelsbrunner [4] have presented a solution for the static fixed windowing problem, that is, no insertions or deletions of points are allowed. Their solution is optimal with respect to the query time and the space required by the data structure. They allow arbitrary convex figures as a (fixed shape) window. The solution of [4] can be dynamized by well-known general dynamization methods (see, for example, [3,8,9,11]). However, a much simpler approach works if we do not allow curves in the windows, but only allow *polygons.* This approach leads to an *optimal* solution with respect to query and updating time and the size of the data structure.

In Section 2 we solve the dynamic fixed windowing problem for rectangular windows. (Think of an interactive graphical display screen.) Our solution exploits the features of the priority search trees of McCreight [6]. We, in particular, propose double-ended priority search trees as appropriate structures to implement the solution. In Section 4 we show how to extend this solution to triangular and polygonal windows.

The variants of priority search trees discussed in this paper are of interest in their own right. The algorithms and data structures developed for solving the dynamic fixed windowing problem have direct applications to computer graphics, VLSI-design, and other areas.

2 Rectangular windows

In this section we first introduce some notation and then discuss the case where the given dynamic set of objects is a set of points in the plane and where the given window is a two-dimensional range, that is, a rectangle.

We assume a system of cartesian (x, y)-coordinates in the Euclidean plane. For two points $a = (a_x, a_y)$ and $q = (q_x, q_y)$ the translation of a by q is given by $a + q = (a_x + q_x, a_y + q_y)$ and for a point set A and a point q we have

$$A_q = A + q = \{(a_x + q_x, a_y + q_y) | a \in A\}$$

Let P be a set of n points and let W be a rectangular window given by its left, right, bottom, and top boundaries, that is, $W = (x_l, x_r, y_b, y_t)$. For an arbitrary point q we want to carry out the following operations:

insert(P, q): $P \leftarrow P \cup \{q\}$.

delete(P, q): $P \leftarrow P \setminus \{q\}$.

window$_W(P, q)$: report all points in $P \cap W_q$.

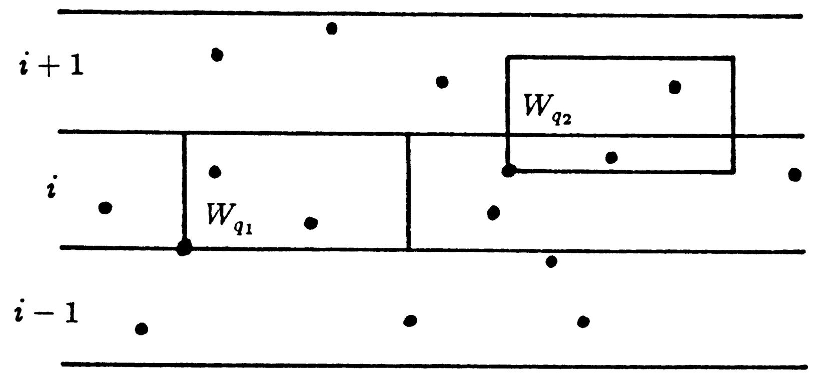


Figure 1: Slabs and groundedness.

A representation of the set P together with algorithms for performing an insert, delete, and window operation on P will be called a *solution* of the dynamic rectangular fixed windowing problem for points (*DRW-problem* for short).

In order to solve the DRW-problem we first, conceptually, slice the Euclidean plane into horizontal slabs of height $Y = \text{height}(W) = y_t - y_b$. We call $s_i = \{p | iY \leq p_y < (i+1)Y\}$ the ith *slab.* If $p \in s_i$, we call i the *slab number* of the point p and denote it by $s(p)$. The slab number of a point can be computed in constant time with respect to the point set.

The decomposition of the plane into slabs of height Y has the following important consequences:

1. For any query point q, W_q intersects at most two adjacent slabs.

2. For any slab s and for any query point q, we have

   (a) either $W_q \cap s = \emptyset$

   (b) or $W_q \cap s \neq \emptyset$ and (W_q is south grounded on s or W_q is north grounded on s).

Here we use the notation of *groundedness* introduced in [4]: For a query region (a window) R and a slab s we say that R is *south grounded* (*north grounded*) on s if the intersection of R with the lower, that is, the southern (the upper, that is, the northern) boundary of the slab s equals the orthogonal projection of R onto this boundary. In Figure 1 W_{q_2} is south grounded on s_{i+1} and north grounded on s_i. Clearly, if the translate W_q is north grounded *and* south grounded on s then $W_q \cap s = W_q$.

The basic idea for solving the DRW-problem is now easily described: To each slab s we associate a pair of priority search trees for the points of $P \cap s$; see Figure ??. The priority search trees of McCreight [6] allow us to

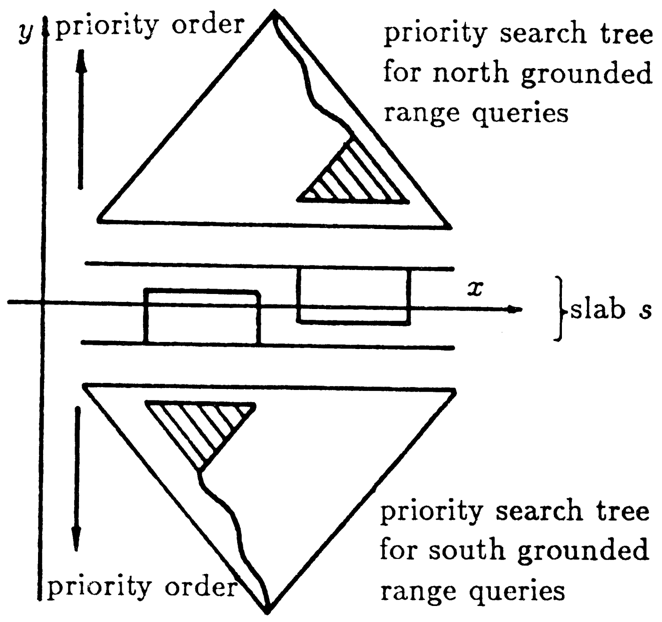

y
priority order
priority search tree for north grounded range queries
x
slab s
priority search tree for south grounded range queries
priority order

Figure 2: Arrangement of priority search trees.

carry out insertions and deletions of points and to report all points falling into a south (north) grounded range. Here we need "(x, y) priority search trees": The trees are leaf search trees for the x-values of the stored points. Each point is stored on a root-to-leaf path to the leaf corresponding to its x-value, according to its y-value which determines the priority order of the points. Because the priority search tree is a leaf search tree for the x-values of the points, it supports range queries for x-intervals; because it is a heap for the y-values of the points, it allows the reporting of all points whose y-values do not exceed a given threshold. Combining both features allows us to answer grounded queries: South grounded queries require us to store the points with smaller y-values closer to the root, that is, the priority order is the increasing y-ordering. North grounded queries require us to store the points with larger y-values closer to the root , that is, the priority order is the decreasing y-ordering. Thus, using a pair of priority search trees for the n points in a slab s allows us to carry out insertions and deletions in time $O(\log n)$ and to answer a north or south grounded range query in time $O(\log n + k)$, where k is the size of the answer. Here we assume that the priority search trees are organized as fully dynamic structures and not as semidynamic ones over a bounded universe (see [6] for details). Note that a priority search tree for n points requires $O(n)$ space. In order to achieve logarithmic performance and linear space in the number N of points we have to avoid any explicit representation of the empty slabs. This can easily be achieved by maintaining the slab numbers of exactly the nonempty slabs in a balanced binary search tree T_s: To each slab number in T_s we associate a pair of priority search trees for exactly those points of P with this slab number. To complete the solution of the DRW-problem we describe how to carry out the required operations.

insert(P, q): Determine the slab number $s(q)$ of q; search the tree T_s for $s(q)$; if $s(q)$ occurs in T_s, insert q into the two priority search trees associated to $s(q)$, otherwise (that is, if $s(q)$ does not occur in T_s) insert $s(q)$ into T_s, create a new pair of priority search trees associated to $s(q)$ both for the only point q.

delete(P, q): Analogous to insertion.

window$_W(P, q)$: Use T_s to determine the, at most two, nonempty slabs that intersect W_q and use the associated priority search trees to report the points in the south — or north — grounded intersection of W_q with these slabs.

It is evident from the above description that both insertion and deletion take time $O(\log N)$. A window operation can be carried out in time $O(\log N + k)$, where k is the size of the answer. The structure requires $O(N)$ space; therefore, the solution is optimal.

Because both the priority search trees of a slab contain the same x-order the space bound can be improved by a constant factor, without significant influence on the time bound, by constructing only one leaf search tree for the x-coordinates. The points are then stored twice in the tree: once in increasing y-order and once in decreasing y-order. When answering a south grounded query only the fields storing the increasing order are inspected; when answering a north grounded query only the decreasing order is of interest.

In the next section we introduce a combination of priority search trees and the double-ended heaps of [1]. These allow us to store each y-value exactly once thus saving still more space.

Remark: The restriction to a fixed window W is not essential. As long as the height of the window is not smaller than the height of the slabs the same structure can still be used for other rectangular windows. However, the time to carry out the above algorithms now increases with a factor proportional to (window height/slab height).

3 Double-ended priority search trees

The priority search trees of McCreight [6] are leaf search trees for one coordinate of the points and heaps for their other coordinate. Depending on the priority order of the heaps, they provide efficient enumeration of points in a "half-rectangle "of type either $[a, b] \times [c, \infty]$ or $[a, b] \times [-\infty, c]$. By replacing conventional heaps by the double-ended ones of Atkinson, Sack, Santoro, and Strothotte [1] we obtain a compact data structure that provides enumeration of points in both half-rectangles. These trees preserve the efficient updating properties of the ones of [6].

In order to explain the details of the new data structure, we need the following definition.

Definition 3.1 *Let $A = \{x_i | 0 \leq i \leq n\}$ be a point set in $\mathbf{R}$. We say that a permutation π of these points is a* minmax order *if*

$$x_{\pi(2i)} \leq x_{\pi(j)} \text{ when } i = 0, 1, \ldots, \lfloor n/2 \rfloor \text{ and } 2i \leq j \leq n$$

and

$$x_{\pi(2i+1)} \geq x_{\pi(j)} \text{ when } i = 0, 1, \ldots, \lfloor n/2 \rfloor - 1 \text{ and } 2i+1 \leq j \leq n$$

There is only one minmax order if all the x_i's are distinct. An arrangement of these points in a list by π is said to be a minmax list *of A.*

In a minmax list, the first element is the minimum of the whole point set, the second one is its maximum. The third element is the minimum of the rest, the fourth one is the maximum of the rest, and so on.

Now we are ready to define our data structure. Let S be a set of points in $\Re^2$. A *double-ended priority search tree* for these points is a *leaf search tree* for their x-coordinates such that

1. Each point occurs only once,
2. Each point (x, y) occurs at a node on the path from the root to the leaf which corresponds to its x-value.
3. The y-coordinates of the points on a path from the root to any leaf form a minmax list.
4. If no point occurs at some node, then the nodes of the subtree rooted at this node are empty, too.

Let h be the height of the underlying search tree. We count the *level* of nodes starting with level zero at the root. Any point (x, y) can be found in the structure in time $O(h)$ by checking the points along the path to the leaf which corresponds to x (Property 2). Moreover, the points in an upper (lower) "half-rectangle" $[a,b] \times [c,\infty]$ ($[a,b] \times [-\infty,c]$) can be reported in the following manner. Assume that each node p in the tree is of type

```
node = record
            split : real;
            point : pair;
            left : ↑ node;
            right : ↑ node
       end
```

where *split* contains the maximal x-value in the left subtree rooted at $p.left\uparrow$ and

```
pair = record
            x : real;
            y : real
       end
```

Then the following procedure, when invoked for the root of the the tree, enumerates all points in the upper half-rectangle $[a,b] \times [c,\infty]$.

```
procedure Enumerate − uhr(a, b, c : real; p : node);
begin
    if {p.point is defined}
    then if $p.point.y \geq c$ or level(p) mod 2 = 0
    then
    begin
        if $p.point \in [a,b] \times [c,\infty]$
        then report p.point;
        if $p.split \geq a$
        then Enumerate − uhr(a, b, c, p.left ↑);
        if $p.split < b$
        then Enumerate − uhr(a, b, c, p.right ↑);
    end
end
```

In order to deal with a lower half-rectangle $[a,b] \times [-\infty,c]$ line 4 of the above algorithm has to be replaced by

```
    then if $p.point.y \leq c$ or level(p) mod 2 = 1
```

Assume that we are visiting a node p. If p does not hold a point, the subtree rooted at p can be left unvisited (Property 4). If the level of p is odd (even) and $y < c$ ($y > c$) then the subtree at p can be left unvisited too (Property 3). Otherwise, let the point (x, y) be stored in p. If $(x, y) \in [a, b] \times [c, \infty]$ ($[a, b] \times [-\infty, c]$) it is reported.

We see that a node is visited only if the point stored in it, its father, or its grandfather is reported or if the node itself, its father, or its grandfather lie on the path from the root to the leaf cooresponding to a or b, because each point (x, y) in a node that lies strictly between the paths from the root to a and b has its x-value in $[a, b]$. Thus, the points in the query half-rectangle are reported in time $O(h + k)$, where h is the height of the tree and k is the number of reported points.

Before we describe the updating operations we explain two auxiliary operations for maintaining the correct heap structure. Following the terminology of [1] we call them "bubble up" and "trickle down"

Let us suppose that we have a tree which satisfies Properties 1-3 but Property 4 is violated. Let p be an empty non-leaf node such that all the subtrees of p satisfy Property 4. If the level of p is odd(even) we choose a point with the smallest (greatest) y-value among the points of the sons and grandsons of p, store this point at p, and make the corresponding son or grandson empty. Then, we recursively fill the emptied node if one of its sons (when present) is non-empty. This "bubble up" operation needs $O(h)$ time, where h is again the height of the tree. After the operation, the subtree rooted at p satisfies Property 4 and the whole tree Properties 1-3.

Let us assume then, that we have a tree which satisfies Properties 1-3, and it has a non-leaf node p such that Property 4 is satisfied in the subtree at p. We wish to make p empty in such a way that the whole tree still satisfies Properties 1-3 and the subtrees of p satisfies Property 4. If p is already empty, we are already done. Otherwise, let (x, y) be the point in p. We shift (x, y) downwards in the tree in the direction of the leaf which corresponds to x. This leaf must be empty (Property 1). If the son of p on the path to the x-leaf is empty, (x, y) is stored in that son. Otherwise, we empty the grandson of p recursively and store (x, y) in the emptied node. This "trickle down" operation requires $O(h)$ time, too.

Now we can **insert** a point (x, y) into a tree in the following way: First, we insert a leaf for the x-value (and update the routing information appropriately). After this, we search for the y-value from the minmax list of the nodes along the path from the root to the x-leaf. This node is emptied by the "trickle down" operation and (x, y) is stored in the empty node. An insertion takes $O(h)$ time.

When we want to **delete** a point (x, y) from the tree, we first search for it, empty the corresponding node, and then we fill the emptied node by the

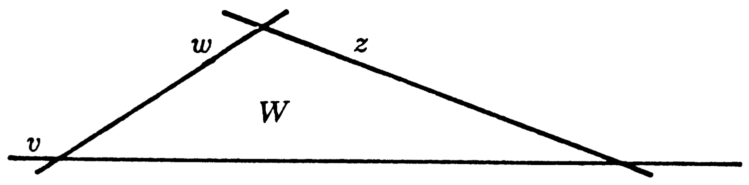


Figure 3: A triangular window.

A deletion takes $O(h)$ time.

We can guarantee the height of the underlying tree to be $O(\log N)$ (N is the number of points) if we choose a balancing scheme for the tree (see for example, [8] for a survey of balanced trees). One rebalancing operation (rotation) changes the links of some non-leaf nodes, but the number of changed links is constant. In order to preserve the minmax order we first empty all the nodes whose links are to be changed, by "trickle down" operations. After the structural changes we again fill them by "bubble up" operations. Thus, a rotation takes $O(\log N)$ time.

If we choose the balancing scheme of Olivié [7] or the one of Bayer [2] we need $O(1)$ rotations after an insertion or a deletion (see Tarjan [10]). Thus we can conclude that our algorithms require $O(\log N + k)$ time for reporting the points in a half-rectangle, $O(\log N)$ time for an update operation, and the data structure requires $O(N)$ space (N is the size of the point set and k the size of the answer to a query).

4 Triangular and polygonal windows

Now let us assume that a fixed triangular window W is given with edges parallel to three lines v, w and z, see Figure 3. We want to extend our solution of the rectangular windowing problem to give a solution of the dynamic triangular fixed windowing problem. Again we are going to decompose the plane into disjoint regions in such a way that every window query W_q will be split into a bounded number of grounded queries for some regions. Further, each grounded query will be of one of a bounded number of types (in the rectangular case the regions are horizontal slabs, any translated window W_q intersects at most two slabs, and the resulting queries are north or south grounded).

A partition into horizontal slabs does not, in general, lead to grounded queries having two parallel edges and a third bordering side because all of the three edges of W_q might intersect one slab. So we choose the cells of the

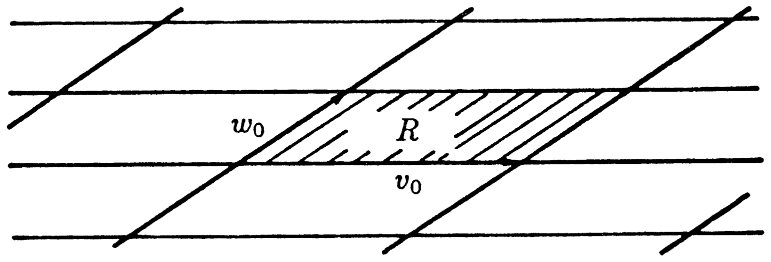


Figure 4: An induced decomposition.

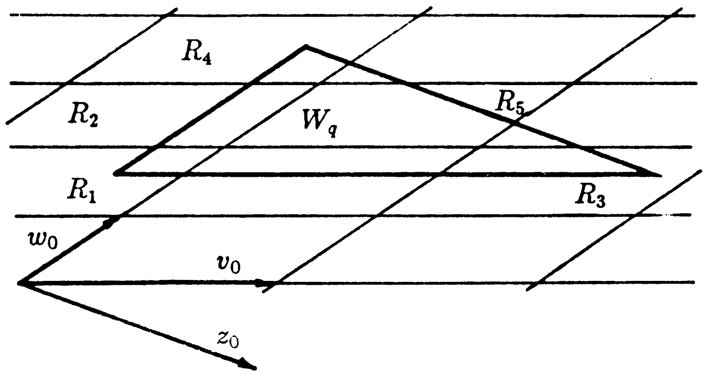


Figure 5: $W_q \cap R_1$ is a north-grounded query.

lattice $\mathbf{Z} \cdot v_0 \oplus \mathbf{Z} \cdot w_0$ as regions, that is the decomposition

$$\Re^2 = \dot{\bigcup_{i,j \in \mathbf{Z}}} R_{i,j}$$

where $R_{i,j} = R + iv_0 + jw_0$, for two vectors v_0 and w_0; see Figure 4. It seems quite natural to choose two of the directions given by W's edges — here v and w — as coordinate axes of the lattice. Now any translate W_q of W will intersect some of the cells. We choose v_0 and w_0 short enough so that at most two edges of W_q are intersected by one cell. Whether the choice of coordinates is indeed wise and how large the cells should be will be discussed later. The intersection of W_q with R_1 (see Figure 5) leads to a north grounded or, more precisely, a $(w = \infty)$-grounded query for this cell *with respect to (w,v)-coordinates.* In Figure 6 v is marked as range coordinate and w as threshold coordinate. Again we can use a (dynamic) priority search tree to answer this query type for R_1 efficiently because priority search trees can be organized with respect to any non-cartesian coordinate system. There is no need for rectangular axes; see for example, [5]. In general, if we are given two nonparallel lines r and t a priority search tree can be organized with respect to r and t that supports *grounded skew*

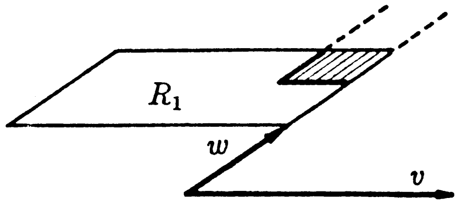


Figure 6: v is a range; w is a threshold.

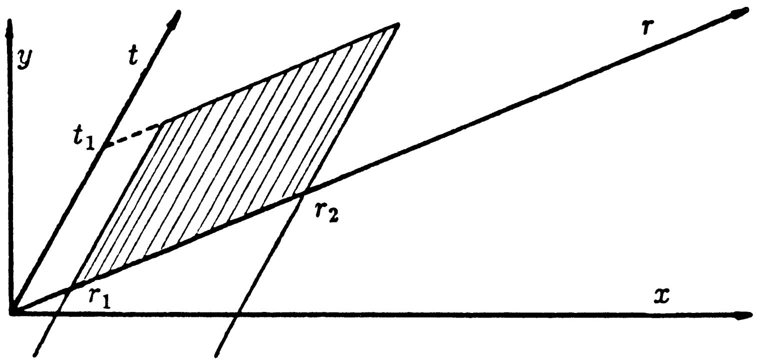


Figure 7: Grounded skew queries.

queries. In (x, y)-coordinates this looks like Figure 7, where a $(t = -\infty)$-grounded skew query is shown for an r-range $[r_1, r_2]$ and a t-threshold t_1. The intersection of W_q and R_2 in Figure 5 may also be regarded as a special case of a $(w = \infty)$-grounded skew query with respect to (w, v)-coordinates. The situation is different in R_3 where the third of W's edges brings a new direction into play. However, if we introduce a third coordinate axis — namely z — parallel to this edge we can regard $W_q \cap R_3$ also as the region of a grounded skew query for R_3, that is, with respect to (v, z)-coordinates as shown in Figure 8. In Figure 8 we have to answer a $(z = -\infty)$-grounded skew query with range coordinate v and threshold coordinate z.

In the same way, $W_q \cap R_4$ is the query region of a $(w = -\infty)$-grounded skew query for R_4 with respect to (z, w)-coordinates; see Figure 9. The situation in R_5 (Figure 10) can be treated as a special case. In the sequel the cells are labelled by the lexicographically ordered index $k = (i, j)$ where i and j are the coordinates of the cell.

Now we solve the *dynamic triangular windowing (DTW)* problem for points in the following way:

To each cell R_k we associate three dynamic priority search trees to hold the points of $P \cap R_k$ with respect to the different coordinate systems. Again we avoid the overhead of empty structures by maintaining a balanced search

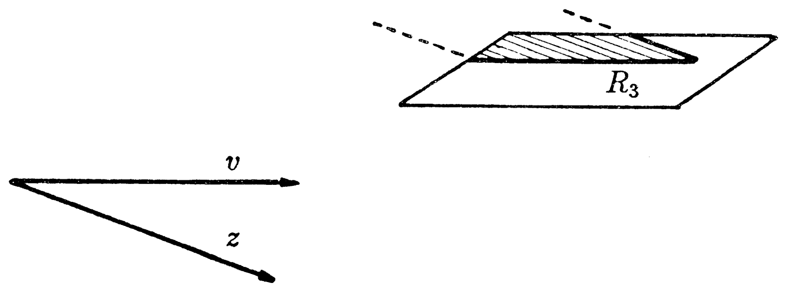


Figure 8: $W_q \cap R_3$ is also a grounded skew query.

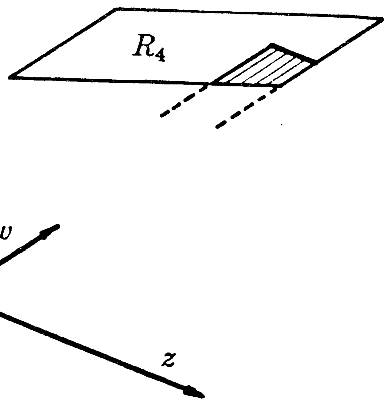


Figure 9: $W_q \cap R_4$ is too.

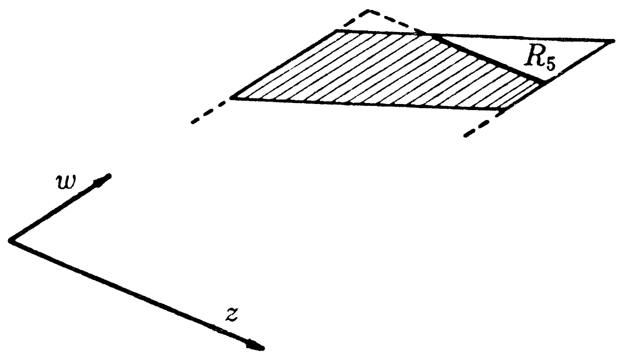


Figure 10: $W_q \cap R_5$ is a special case.

tree T_R for the indices k of the nonempty cells R_k. The operations **insert** (P, q) and **delete** (P, q) are similar to those of the DRW case in Section 2.

window$_T$ (P, q)**:** Compute the bounded number of indices of cells which intersect W_q ; use the tree T_R to determine those of them which are nonempty. For each such cell decide which kind of grounded skew query applies, compute an appropriate range and threshold and use the corresponding priority search tree; report the points transformed into the (x, y)-coordinate system.

Again, our solution requires $O(N)$ space for N points, $O(\log N)$ time for **insert** and **delete** operations, and a **window** operation takes time $O(\log N + k)$. So this solution of the DTW problem is asymptotically both time and space optimal.

The constants hidden in the asymptotic time and space bounds depend on the choice of the partitioning regions. These regions R_k do not necessarily need to be parallelograms; any partition of $\mathbf{R}^2$ into disjoint convex regions R_k will do as long as the following requirements are fulfilled:

1. For any q and all k, the interior of R_k intersects at most two edges of W_q
2. For any q, W_q intersects at most c regions R_k, for some fixed constant c (which may depend on W, of course). Then for any query point q and any region R_k, we have either
   (a) $W_q \cap R_k = \emptyset$ or
   (b) $W_q \cap R_k \neq \emptyset$ and $W_q \cap R_k$ is the query region of a grounded skew query for R_k with respect to one of three fixed coordinate systems.
3. Furthermore, the R_k must be such that
   (a) for any q, the (at most c) region indices k with $W_q \cap R_k \neq \emptyset$ and the intersections of the boundaries of W_q and R_k can be computed in constant time, and
   (b) for any query point p, the region index of p (that is, the k for which $p \in R_k$) can be computed in constant time.

Observe that with respect to Condition 2(b) we have to avoid situations like those of Figure 11 where $W_q \cap R_k$ is a proper subset of the corresponding skew query region $GSQ \cap R_k$ for region R_k. This is because $p \in P$ would be reported although P is not located in the query window W. Clearly our convexity assumption makes such phenomena impossible. But the situation in Figure 11 also contradicts Condition 1 as can be seen by moving W_q

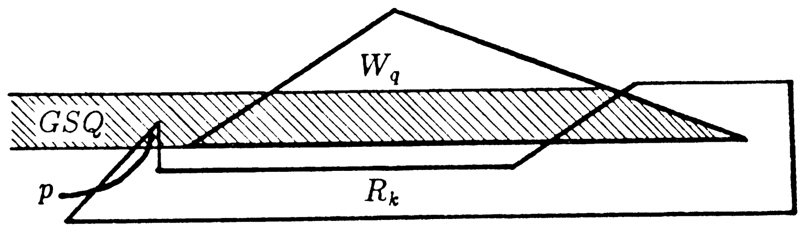


Figure 11: A situation to be avoided.

further downwards. So it remains open how the assumption that all R_k's are convex can be weakened without violating the Condition 2(b). To speed up the calculation of intersections and point locations (Conditions 3(a) and 3(b)) a lattice partition of the plane seems to be an appropriate choice (because the cells are polygons). But it is not clear how large the cells should be made. On the one hand, they must be small enough to satisfy Condition 1 and, on the other hand, time and space performance are improved if we make them as large as possible. For larger cells require fewer data structures storing the point set P, and a translated window W_q intersects fewer cells making querying faster. Furthermore, it is, a priori, not clear which axes should span the lattice.

Let us fix a coordinate system (a,b) and consider (a,b)-oriented cells, that is, parallelograms. Observe that only the relative positioning of the triangle W and the cells is important. So we can, for convenience, consider the triangle as being fixed and translate one such parallelogram, for example, the one lying in the positive quadrant nearest to the origin. Let R denote this parallelogram. From this point of view we may reformulate Condition 1 as:

(1′) For any q, R_q intersects at most two edges of W.

We claim that the parallelogram R satisfies Property 1′ if and only if R can be translated **into** W, that is, if there is some vector q such that $R_q \subseteq W$. To prove the "only if"part, assume that R satisfies Property 1′ and let R sink down from above until it touches or kisses the ground edge g of W; see Figure 12. Note that we do not call this kissing an intersection. If R now intersects the two other edges l and r of W, we can move R down slightly thereby turning the kiss into a third intersection, in contradiction to our assumption. If there are no intersections at all we are finished. So the case of one intersection — say with W's right edge r — is left. Shifting R to the left will either transform this intersection into a kiss without causing any new intersection with l (otherwise moving R back slightly would give a contradiction as above) or stop at the left end of g with R still intersecting r. But in this case R kisses g and intersects or kisses l; so a slight movement

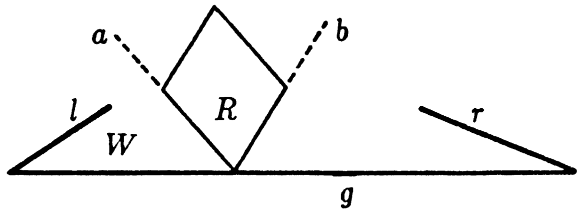


Figure 12: Letting R kiss W.

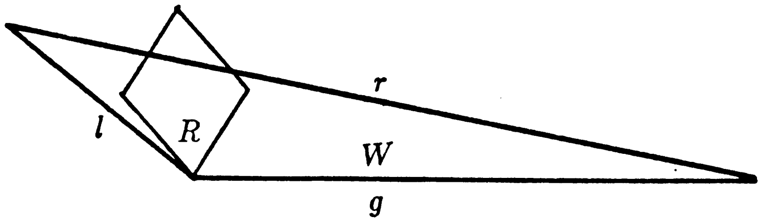


Figure 13: A contradictory position.

will again lead to a contradiction; see Figure 13. To prove the "if"-part, assume $R_q \subseteq W$ for a vector q. If we try to translate R_q further in order to intersect as many of W's edges as possible, we will capture at most two of them and always lose contact with the third. This proves the assertion.

In order to maximize the area of R while maintaining Property $1'$, we, therefore, have to find the maximal (a,b)-oriented parallelogram that can be inscribed in the triangle W. Such a parallelogram must be in contact with all of W's edges; otherwise it can be moved towards an edge it doesn't touch and enlarged; see Figure 14. Two cases now arise. Either W can be translated in such a way that one of the angles of W_q contains an angle of the coordinate axes, or this kind of translation is impossible. In the first

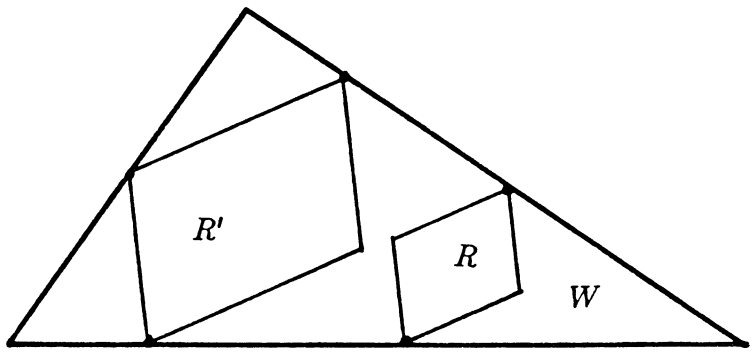


Figure 14: Enlarging R to kiss all edges of W.

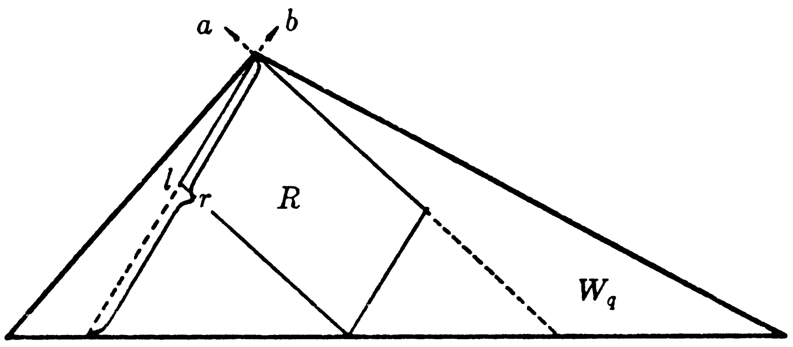


Figure 15: Case 1: W can be translated.

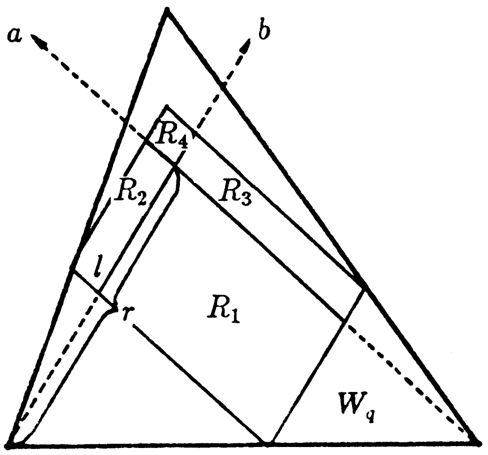


Figure 16: Case 2: W cannot be translated.

case the situation looks like that shown in Figure 15. The upper vertex of R is fixed, because it has to touch both of the upper edges. The left vertex l may vary in the range denoted by r, and the bottom vertex must lie on the ground edge of W between the a and b axes. In the second case, we can translate W so that the origin of the (a, b)-coordinate system falls inside W_q and the axes intersect two of W_q's vertices; see Figure 16. These axes divide R into disjoint parts R_1, R_2, R_3, and R_4. The upper vertex of R_1 is the origin in the (a, b)-coordinate system. The left vertex of R_1, l, can vary in the interval r, the part of the negative b-axis contained in W_q. For each choice of l in r, all vertices of R are uniquely determined; three of them lie on different edges of W_q. It is easy to see that the ratio of the widths of R_1 and R_3 in direction of the b-axis is constant. The same holds for R_1, R_2, and the a-axis. Thus, the areas of R_2, R_3, and R_4 are linear functions of the area

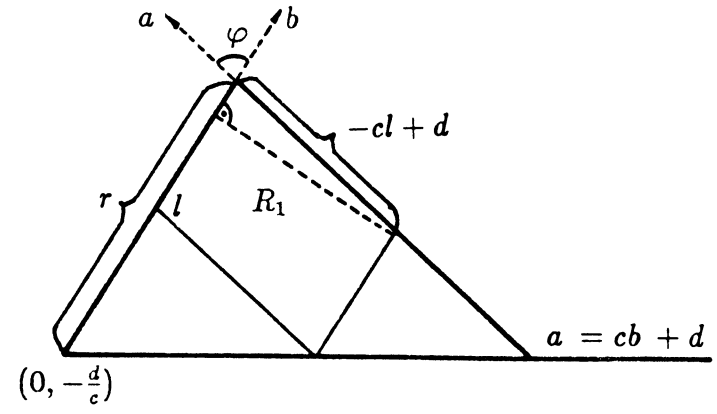


Figure 17: The quadratic optimization problem.

of R_1. Therefore, both cases reduce to the simple quadratic optimization problem depicted in Figure 17, that is, to maximize the area

$$A(l) = l(cl + d)\sin\varphi$$

of R_1 for l in $[-d/c, 0]$, where c and d are constants from the equation $a = cb+d$ of the ground edge of W_q and φ is the angle between the coordinate axes. Since

$$\frac{\mathrm{d}A(l)}{\mathrm{d}l} = \sin\varphi(2cl + d)$$

the maximum occurs at $l = -\frac{d}{2c}$. So the maximal R_1 has — with respect to (a, b)-coordinates — half the ground size and half the height of W_1 (Figure 18). Therefore the maximal R_1 has half the area of the triangle W_1. Note that this also holds for $R_1 \cup R_2$ and $W_1 \cup W_2$.

To summarize, we have shown how to find a maximal parallelogram with Property 1 for any pair of prescribed coordinate axes a and b. If we are free to choose them, as is the case in the DTW-problem, we must select, say, a parallel to one of W's edges and b must be chosen in such a way that it is not contained in the two angles of W adjacent to a, in order to avoid Case 1. Giving R half the ground size and half the height with respect to a, b leads to a maximal parallelogram which has Property 1'; it is half as large as the triangular window W. Furthermore, this choice is also optimal with respect to the maximal number of regions a translate W_q can intersect. It is not hard to see that this number is bounded by eight (see Figure 5) and no other lattice having Property 1 can do better. Note that we may use a rectangular grid as a partition of $\Re^2$ whenever it seems convenient. As before in the DRW case of Section 2, our data structure enables us to answer triangular window queries with respect to not only the given window W but also to any other fixed triangular window W' if

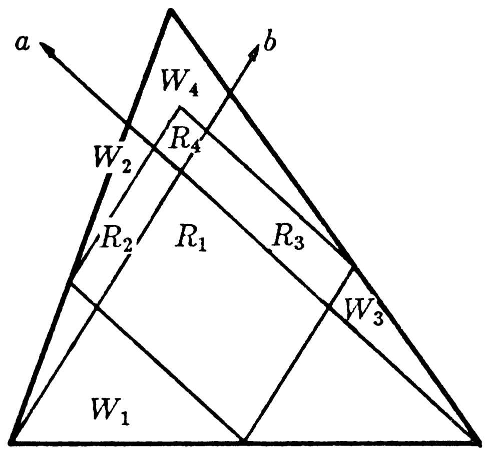


Figure 18: The maximal R_1.

- the edges of W' have the same orientation as those of W and
- the cell R of the lattice chosen for W can be inscribed in W'

It is clear that this solution leads immediately to an asymptotically optimal solution to the *dynamic fixed polygonal windowing (DPW)* problem for points. We triangulate the window and, hence, reduce the DPW problem to a fixed number of DTW problems. These may be solved separately using different lattices. Each point $p \in P$ is stored three times per lattice and each window query leads to at most eight grounded skew queries per lattice.

However, it is more space efficient to decompose the given polygon into convex subpolygons. For a convex m-gon PG we can proceed as follows: Choose a lattice such that Condition 1 is satisfied, that is, in any translate PG_q each cell intersects at most two edges of PG_q and when a cell intersects two edges they must be adjacent in PG. To each nonempty cell associate m priority search trees that support grounded skew queries with respect to the m pairs of adjacent edges of PG.

Note that, for $m > 3$, it is no longer the case that a parallelogram satisfies Condition 1$'$ if it can be inscribed in PG; see Figure 19. To find an optimal lattice seems to be somewhat more complicated than in the triangular case. However, a (good) cell-decomposition can be easily found as follows. Fix a pair of coordinates a, b, triangulate PG, and construct for each triangle the maximal parallelogram with respect to a and b as shown above. Now take the minima of all the widths and heights as the width and height of the desired parallelogram.

Again, our construction enables us to answer queries with respect to any fixed convex polygon PG' as long as its corresponding m edges are parallel

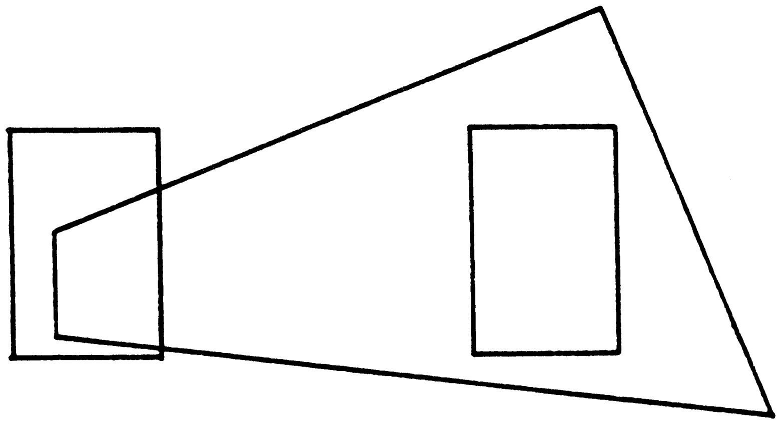


Figure 19: Condition 1′ is not necessarily implied.

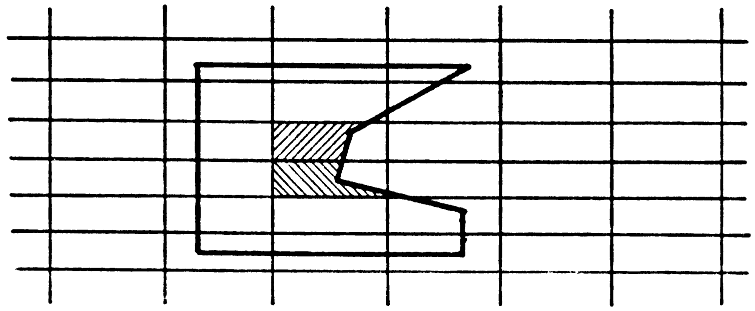


Figure 20: Convexity appears to be necessary.

to those of PG and Condition 1 is fulfilled.

Let S be the set of orientations of the edges of PG. If each nonempty cell is associated with $\binom{m}{2}$ priority search trees corresponding to all pairs of edges of PG we can also answer all fixed S-oriented (see [5]) convex polygonal window queries as long as Condition 1 is not violated. But convexity seems to be essential because priority search trees do not support queries of type "cell \ grounded skew region" (see Figure 20) which may arise in the non-convex case. Note that we can combine two priority search trees to give one double-ended priority search tree (as we did in the rectangular case in Section 3) whenever there are two vertices whose adjacent edges are pairwise parallel as shown in Figure 21. The queries concerning vertices r and s can be treated as $(w = -\infty)$-, respectively, $(w = \infty)$-grounded skew queries and answered by consulting one double-ended priority search tree.

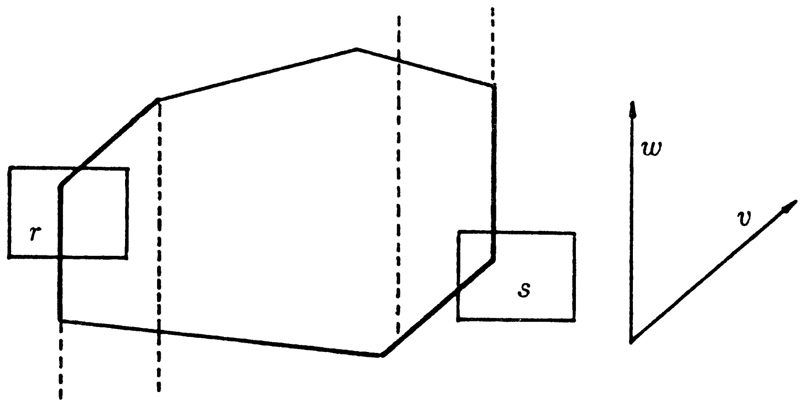


Figure 21: Combining two priority search trees.

5 Conclusions

In Sections 2 and 3 we have proved the following theorem:

Theorem 5.1 *The dynamic fixed polygonal windowing problem can be solved in space $O(N)$ and time $O(\log N)$ for* **insert** *and* **delete** *operations and $O(\log N + k)$ time for* **window** *operations, where N is the number of points and k is the size of the output.*

For fixed windows this solution is asymptotically time and space optimal. If the number of edges of the query window is a parameter of the problem we see that the time and space bounds grow linearly with this parameter. How the constants of the time and space bounds can be minimized was discussed in some detail in the rectangular and triangular case.

Unfortunately, the approach of using priority search trees seems not to work if the window is not a polygon. So we leave open the question of an optimal solution for the dynamic fixed circular windowing problem for points, and, more generally, for the dynamic fixed windowing problem, where the window is an arbitrary region in the plane.

References

[1] M.D. Atkinson, J.-R. Sack, N. Santoro, and Th. Strothotte. Minmax-heaps, order statistics trees and their application to the coursemarks problem. *Communications of the ACM*, 29:996–1000, 1986.

[2] R. Bayer. Symmetric binary B-trees: data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.

[3] J.L. Bentley and J.B. Saxe. Decomposable searching problems I: static-to-dynamic transformation. *Journal of Algorithms*, 1:301–358, 1980.

[4] B. Chazelle and H. Edelsbrunner. Optimal solutions for a class of point retrieval problems. In *Proceedings of the 12th International Colloquium on Automata, Languages, and Programming*, pages 80–89, 1985.

[5] R.H. Güting. Dynamic C-oriented polygon intersection searching. *Information and Control*, 63:143–163, 1984.

[6] E.M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14:257–276, 1985.

[7] H.J. Olivie. A new class of balanced binary search trees. *RAIRO Informatique théoretique*, 16:51–71, 1982.

[8] M.H. Overmars. *The Design of Dynamic Data Structures.* Springer-Verlag, New York, 1983.

[9] M.H. Overmars and J. van Leeuwen. Two general methods for dynamizing decomposable searching problems. *Information Processing Letters*, 12:168–173, 1981.

[10] R.E. Tarjan. Updating balanced search trees in $O(1)$ rotations. *Information Processing Letters*, 16:253–257, 1983.

[11] J. van Leeuwen and D. Wood. Dynamization of decomposable searching problems. *Information Processing Letters*, 10:51–56, 1980.