RANGE SEARCHING IN A SET OF LINE SEGMENTS

Mark H. Overmars

RUU-CS-83-6
februari 1983

Rijksuniversiteit Utrecht

Vakgroep informatica

Princetonplein 5
Postbus 80.002
3508 TA Utrecht
Telefoon 030–531454
The Netherlands

vakgroep informatica R.U. Utrecht

RANGE SEARCHING IN A SET OF LINE SEGMENTS

Mark H. Overmars

Technical Report RUU-CS-83-6
februari 1983

Department of Computer Science
University of Utrecht
P.O. Box 80.002
3508 TA Utrecht
the Netherlands

RANGE SEARCHING IN A SET OF LINE SEGMENTS*

Mark H. Overmars

Department of Computer Science, University of Utrecht
P.O. Box 80.002, 3508 TA Utrecht, the Netherlands

Abstract. The range searching (or windowing) problem asks for an accommodation of a set of objects such that those objects that lie (partially) in a given axis-parallel rectangle can be reported efficiently. We solve the range searching problem for a set of n non-intersecting, but possibly touching, line segments in the plane and give a data structure that allows for range queries in $O(\log^2 n+k)$ time, where k is the number of reported answers. The structure is dynamic and allows for insertions and deletions of line segments in $O(\log^2 n)$ time. The structure uses $O(n \log n)$ storage. The related problem of moving the window (range) parallel to one of the coordinate-axes, determining the first line segment that will become visible or stops being visible, is treated as well and similar bounds are obtained.

Keywords and phrases. Range searching, windowing, moving, line segments, segment tree, planar point location.

1. *Introduction*.

Given a set of n objects in d-dimensional space, the range searching problem asks for an accommodation of these objects such that those objects that lie (partially) in a given d-dimensional axis-parallel hyper-rectangle can be computed efficiently. The problem has been considered in great detail for the case in which the objects are d-dimensional points. The best dynamic solutions (i.e., data structures that store the set of points such that new points can be inserted and existing points can be deleted efficiently) known so far were given by Edelsbrunner

----

* This work was supported by the Netherlands Organization for the Advancement of Pure Research.

[1] (based on a structure of McCreight [7]) yielding:

$$\begin{aligned} \text{query time} &: O(\log^{d-1} n+k), \\ \text{insertion time} &: O(\log^{d} n), \\ \text{deletion time} &: O(\log^{d} n), \\ \text{storage} &: O(n \log^{d-1} n), \end{aligned}$$

where k is the number of reported answers, and by Overmars and van Leeuwen [9] (based on a structure of Lueker [6] and Willard [11]) yielding:

$$\begin{aligned} \text{query time} &: O(\log^{d} n+k), \\ \text{insertion time} &: O(\log^{d} n), \\ \text{deletion time} &: O(\log^{d-1} n), \\ \text{storage} &: O(n \log^{d-1} n). \end{aligned}$$

For sets of objects other than points efficient solutions are largely unknown. Only the case in which the objects are axis-parallel hyper-rectangles as well has been treated (see e.g. Edelsbrunner and Maurer [2]).

In the two-dimensional case (i.e., d=2), the range searching problem is essentially the same as the windowing problem considered in computer graphics (see van Leeuwen [10]). In graphical applications the set of objects often consists of non-intersecting but possibly touching line segments (for example a picture of some three-dimensional scene with the hidden line removed) See figure 1 for an example of windowing (i.e., performing a range query on) such a set. The problem of windowing a

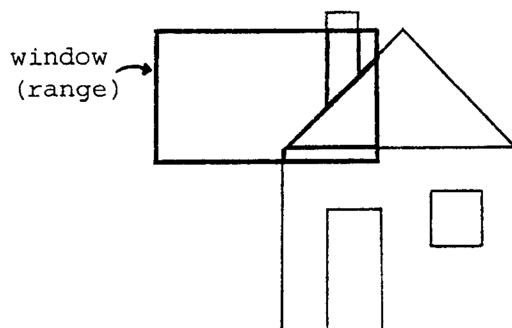

figure 1.

set of line segments in the plane was addressed in Edelsbrunner, Overmars and Seidel [4]. A static solution was given yielding

$$\text{query time} : O(k \log n),$$
$$\text{building time} : O(n \log n),$$
$$\text{storage} : \begin{cases} O(n) & \text{for fixed sized windows,} \\ O(n \log n) & \text{for arbitrary sized windows,} \end{cases}$$

where k is the number of (partially) visible line segments in the window. Although their method is efficient in a number of applications, the query time might become very high when the number of visible line segments is large. A second drawback of the structure is that it is static.

In this paper we describe an efficient dynamic solution for the range searching problem on a set of non-intersecting line segments yielding

$$\text{query time} : O(\log^2 n+k),$$
$$\text{insertion time} : O(\log^2 n),$$
$$\text{deletion time} : O(\log^2 n),$$
$$\text{storage} : O(n \log n).$$

The structure has a better query time than the structure in [4] when $k>\log n$. In Section 2 we solve the special case in which the range is a horizontal line segment. To this end we use a structure very much related to a structure given by Edelsbrunner and Maurer [3] but used there for solving a quite different searching problem. In Section 3 we will use this special case solution for solving the problem with arbitrary sized ranges.

A related problem asks for an accommodation of the set of line segments such that the first line segment(s) coming in or disappearing from the window when moving the window parallel to one of the coordinate axes can be determined efficiently. This problem is known as the moving problem. In [4] a static solution is given yielding

$$\text{query time} : O(\log n+k),$$
$$\text{building time} : O(n \log n),$$
$$\text{storage} : \begin{cases} O(n) & \text{for fixed sized windows,} \\ O(n \log n) & \text{for arbitrary sized windows,} \end{cases}$$

where k is the number of line segments involved in the first change. In Section 4 we give a dynamic solution for the problem yielding

$$\text{query time} : O(\log^2 n+k),$$
$$\text{insertion time} : O(\log^2 n),$$
$$\text{deletion time} : O(\log^2 n),$$
$$\text{storage} : O(n \log n).$$

In Section 5 we give some conclusions and extensions. In particular, it is shown that the moving problem can be used for obtaining a dynamic solution to the so-called region location problem. Also a number of open problems is listed.

2. A special case.

In this section we solve the range searching problem for a set of line segments for the special case in which the range consists of a horizontal line segment. In Section 3 we will use this solution for solving the general range searching problem. Hence, we have the following problem: given a set V of n non-intersecting (but possibly touching) line segments in the plane and a horizontal query line segment s, determine those segments in V that have at least one point in common with s. We will give a method of storing the segments in V such that queries, insertions and deletions can be performed efficiently. We will solve the problem in three parts. In Subsection 2.1. we solve the problem for a set V of horizontal line segments. Next, in Subsection 2.2. we will locate all segments in V that have one of their endpoints in common with the query segment s. In Subsection 2.3. we locate the remaining segments that have a point in common with s, i.e., we solve the problem for a set of open (i.e., their endpoints not included) non-horizontal line segments. We assume that the set contains no degenerated line segments, i.e., points. When the set does we can treat them separately using the known solutions for range searching in a set of points.

2.1. Horizontal line segments.

We have the following problem: given a set of horizontal line segments V and a query line segment s, determine those segments in V that have a point in common with s. Let each horizontal line segment $s_i \in V$ be given by its leftmost point $(x1_i, y_i)$ and its rightmost point $(x2_i, y_i)$ and let s be given by its leftmost point $(x1,y)$ and its rightmost point $(x2,y)$. We say $s_i < s_j$ when $y_i < y_j$ or $y_i = y_j$ and $x1_i < x1_j$ (and hence, also $x2_i \leq x1_j$). In this order we store the segments in V in the leaves of a balanced binary search tree T. We link the leaves in a doubly linked list. It is clear that horizontal line segments can be inserted and deleted in T in O(log n) time, when V contains n segments, and that T uses O(n) storage. To perform a query with a horizontal line segment s we search for the largest segment s_i in T with $s_i < s$. When s_i and s

have a point in common take $s'=s_i$. Otherwise take s' to be the next larger segment in T (i.e., the neighbor of s_i). If s' and s have no point in common we are done. Otherwise, we report s' and walk along the list of leaves, reporting each segment we pass until we reach a segment s_j that does not have a point in common with s. In this way all horizontal line segments that have a point in common with s are reported exactly once. Such a query clearly takes O(log n+k) time where k is the number of reported answer. Hence, we have solved the first subproblem within the following bounds:

query time : O(log n+k),
insertion time : O(log n),
deletion time : O(log n),
storage : O(n).

2.2. Endpoints.

We have the following problem: given a set of (non-horizontal) non-intersecting line segments V, determine those segments that have one of their endpoints on a horizontal query line segment s. This problem can of course be solved using range searching on a set of points but we have to be careful. As line segments are allowed to touch each other some points might correspond to many different line segments. Most structures for range searching do not allow for multiple points (although they in general can be adapted). We will describe a simpler structure that only works in the restricted case of a horizontal line segment as range.

Let V' be the set of all endpoints of line segment and let with each point the line segment it belongs to be given. V' might contain many equal points. For points $p_i = (x_i,y_i)$ and $p_j = (x_j,y_j)$ in V' we say $p_i<p_j$ when $y_i<y_j$ or $y_i=y_j$ and $x_i<x_j$. We store all points in V' in this order in the leaves of a balanced binary search tree T. Equal points we store in order of the other endpoint of the segments they belong to. As no two segments have equal endpoints on both sides, this uniquely determines the order of the points. Hence we can locate the two endpoints belonging to a given line segment in O(log n) time. We link the leaves in a doubly linked list. Insertions and deletions of line segments (i.e., of two points) can easily be performed in O(log n) time. To perform a query with a horizontal line segment s we search with (x1,y) (its leftmost point) in T to locate the smallest point $p_i \geq (x1,y)$. When $p_i > (x2,y)$ we are done. Otherwise we report the

segment belonging to p_i and walk along the list of points, reporting the segment belonging to each point we pass, until we reach a point p_j with $p_j > (x2,y)$. In this way all segments with an endpoint on s are reported and, as the set contains no horizontal line segments, each segment is reported exactly once. The query time is clearly bounded by $O(\log n+k)$, where k is the number of reported answers. Hence, we have solved the second subproblem within the following bounds:

query time : $O(\log n+k)$,
insertion time : $O(\log n)$,
deletion time : $O(\log n)$,
storage : $O(n)$.

2.3. Open line segments.

We are left with the following subproblem: given a set V of non-intersecting, non-horizontal open line segments and a horizontal line segment s, determine those line segments in V that have a point in common with s. Let the endpoints of the segments in V, ordered with respect to their y-coordinate be $(x_1,y_1),(x_2,y_2),\ldots(x_m,y_m)$, i.e., $i<j \Rightarrow y_i \leq y_j$ (m being the number of endpoints). We partition the y-axis in the intervals $(-\infty:y_1),[y_1:y_2),[y_2:y_3),\ldots,[y_m:\infty)$, where $[y_i:y_{i+1})$ means the interval between y_i and y_{i+1} with y_i included but y_{i+1} not. See figure 2 for an example. Note that a number of intervals might be empty (when more than one point has a same value in the y-coordinate).

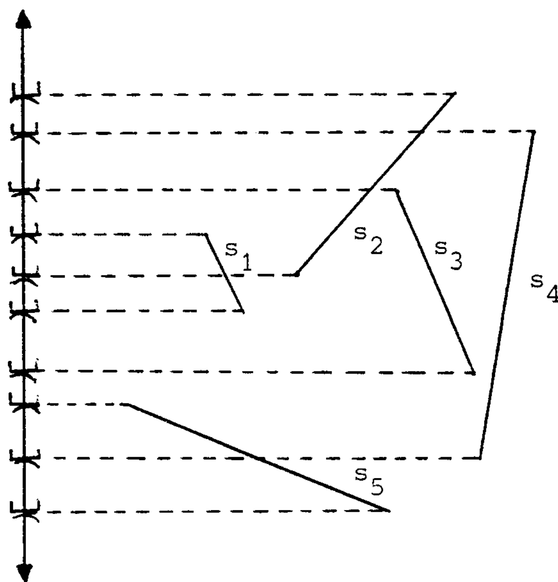

figure 2.

These y-segments we store in the leaves of a BB[α]-tree S. With each internal node β we associate the interval int_β that is the union of the intervals of the leaves in the subtree rooted at β. We augment S by associating with each node β a structure S_β that contains all open line segments in V whose y-projection covers the whole interval int_β but not the whole interval $int_{f(\beta)}$ associated with the father of β (when β is not the root of the tree). See figure 3 for the situation we get in our example. At each node the segments are indicated that have to come in the associated structure. Some care has to be

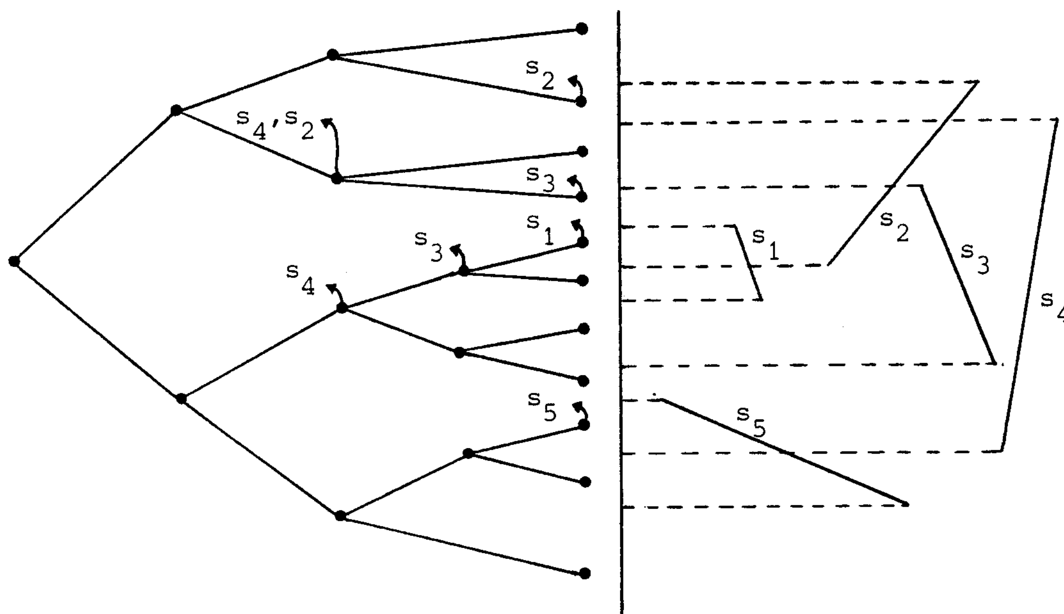

figure 3.

taken at the leaves. In each leaf β with interval $[y_i:y_{i+1})$ we store the segment s_β belonging to (x_i,y_i) when (x_i,y_i) is the lowest endpoint of this segment. Otherwise s_β is empty. We will first show how a query with a horizontal line segment s can be performed. Let the left and right endpoint of s be (x1,y) and (x2,y). We search with y in S. Let the nodes passed be $\beta_1,\beta_2,\ldots$.

<u>Lemma</u> 2.1. Each line segment s_i whose y-projection covers y is contained in exactly one S_{β_j} or it is associated with the leaf β we reach (i.e., s_β).

Proof

When s_i does cover int_β, there must be some node β_j on the search path such that s_i covers the whole interval below β_j but not the whole interval below the father of β_j (s_i can never cover the root of the tree). Then s_i must be in S_{β_j}. So assume s_i does not cover β. Let $int_\beta = [y_i : y_{i+1})$. Then y_i must be an endpoint of s_i. y_i cannot be the highest endpoint of s_i for in that case s_i (that is open) could never cover y. Hence $s_i = s_\beta$ (the segment associated with β).

□

Hence, beside s_β, each line segment that has a point in common with the query line segment s is in exactly one of the S_{β_j} structures. So, to locate all these line segments we can restrict our attention to querying the S_{β_j} structures. Determining whether s_β has a point in common with s can clearly be done in O(1) time. We will show how to structure the S_β structures such that queries on them can be performed efficiently. The y-projection of each segment s_i in S_β covers int_β. We will restrict each segment s_i to the part that lies in the horizontal slab defined by int_β (see figure 4.) As the line segments do not intersect

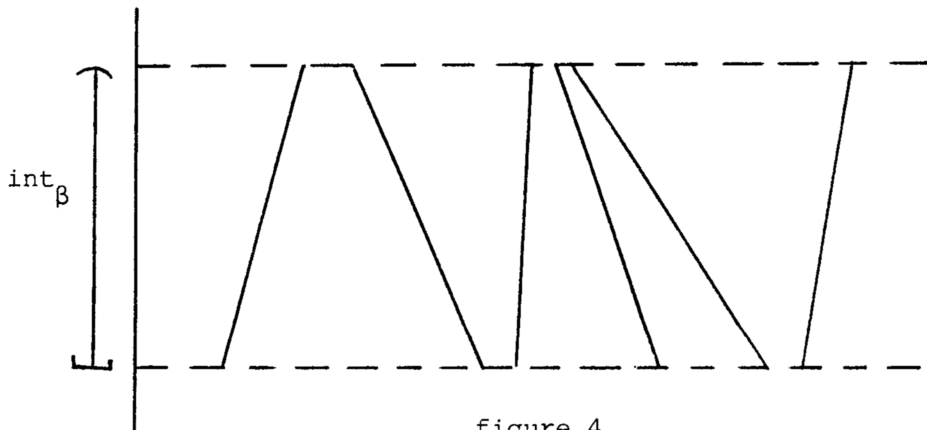

figure 4.

they appear in horizontal order in the slab. We store the segments in this order in a balanced binary search tree. We like them in a doubly linked list. To perform a query with segment s we search with the left endpoint (x1,y) in the structure to locate the first segment s_i that lies right of or contains (x1,y). This is possible because (x1,y) lies in the slab. If s_i lies also to the right of the right endpoint of s we are done. Otherwise we report s_i and walk along the list of segments, reporting each segment we pass, until we reach a segment that lies past the right endpoint of s. Such a query clearly takes O(log n+k)

time where k is the number of line segments reported. As the main structure S has depth O(log n) we have to query at most O(log n) associated S_β structures. Hence the total query time is bounded by $O(\log^2 n+k)$.

We will now show how to perform updates on the structure. To perform an insertion of a line segment s_i with endpoints $p1_i$ and $p2_i$ we search with both $y1_i$ and $y2_i$ in the main structure S to locate the leaves $y1_i$ and $y2_i$ lie in. Let the leaf $y1_i$ comes in contain the interval $[y_j:y_{j+1})$. We spilt it in two intervals $[y_j:y1_i)$ and $[y1_i:y_{j+1})$ and add them as sons to a new internal node β. Computing the structures that have to be associated to β and the two new leaves can easily be done in O(1) time. Similar for $y2_i$. We also have to insert thé new segment s_i in a number of associated structures. For sometime $y1_i$ and $y2_i$ will follows the same path in the tree but at some node β $y1_i$ will go to the leftson of β and $y2_i$ to the rigthson of β. From this moment on we have to insert s_i in structures S_γ for nodes γ that are rightson of a node on the search path of $y1_i$ or leftson of a node on the search path of $y2_i$ but are not on one of the search paths (see figure 5). This are at most O(log n) structures. Inserting a line segment in a S_j structure can easily be done in O(log n) time. Hence the total amount of time required is $O(\log^2 n)$. There is only one problem. Because we inserted two points in the BB[α] tree S, it may have become out of balance and we have to rebalance it. We will solve this problem below. When we want to delete a line segment s_i we first have to locate the

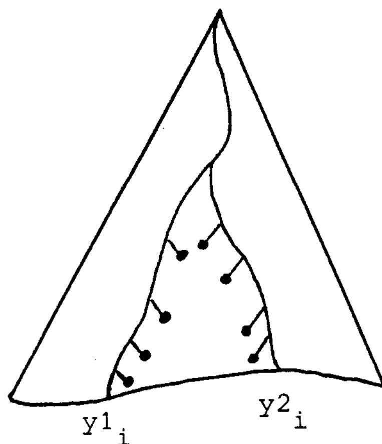

figure 5.

leaves corresponding to its endpoints. This might not be easy for many leaves can correspond to the same point. To this end we use a balanced binary search tree DICT which stores all line segments currently in the set. With each segment in DICT we keep two pointers to the leaves in S corresponding to its endpoints. Clearly queries, insertions and

deletions on DICT can be performed in O(log n) time. (We could have avoided the use of DICT by keeping extra information in the leaves and internal nodes of S but we have not done so for sake of clarity.) Once we have located the two leaves corresponding to the endpoints of s_i we delete them and extend the intervals associated with their brothers. Next we have to delete s_i from S_γ structures that border the search path towards the endpoints in the way displayed in figure 5. Deleting a line segment from a S_γ can easily be done in O(log n) time. Hence, the total deletion time is bounded by $O(\log^2 n)$, provided we are able te rebalance the structure when it becomes out of balance.

To rebalance the augmented BB[α]-tree S we use a technique of Lueker [6] and Willard [11]. We will only give a sketch of the method. Details can be found in [6,11]. BB[α]-trees can be rebalanced by means of single and double rotations at nodes along the path towards the inserted or deleted point. When we perform some rotation we have to reconstruct the structures associated with the nodes involved in the rotation. Let β be some internal node and let n' be the number of leaves (i.e., points) in the subtree rooted at β. Let s_i be a line segment stored in S_β. Then at least one of the endpoints of s_i must lie below the father of β. As S is a BB[α]-tree it follows that the number of line segments in S_β is bounded by O(n'). Building S_β can clearly be done in O(n' log n') time. Willard [11] and Lueker [6] have shown that (when α is properly chosen) when a node β is involved in some rotation, it takes at least $\Omega(n')$ updates in the subtree rooted at β before β is again involved in a rotation. Moreover they show that the work for building the new associated structure S_β can be spread over $\Omega(n')$ updates, meanwhile using the old S_β for query answering. It follows that with each update in the subtree rooted at β we have to do O(log n') = O(log n) work on constructing S_β. This O(log n) work has to be done for each node on the path towards the inserted or deleted point so we need $O(\log^2 n)$ time for rebalancing per update.

<u>Theorem</u> 2.2. There exists a data structure that stores a set of n non-intersecting but possibly touching line segments in the plane such that those segments that have a point in common with a given horizontal line segment can be determined in

$$\text{query time}: O(\log^2 n+k),$$

where k is the number of reported line segments. The structure is dynamic and has

insertion time : $O(\log^2 n)$,
deletion time : $O(\log^2 n)$,
storage : $O(n \log n)$.

Proof

The query time and insertion and deletion time follow from the above discussion. To estimate the amount of storage required, note that at each level of the BB[α]-tree S, each line segment occurs in at most 2 associated structures. As a S_β structure containing n' points takes $O(n')$ storage, the S_β structures associated with nodes on some level of the tree together require $O(n)$ storage. As the depth of S is bounded by $O(\log n)$ the bound on the amount of storage required follows.

□

3. Range searching.

We will now solve the general range searching problem for a set of non-intersecting line segments. A line segment lies (partially) in a range r if and only if at least one of the following two conditions holds:

(i) s has a point in common with the boundary of r,

(ii) the mid-point of s lies in r.

A line segment may clearly satisfy both conditions. To find those line segments in a set V that lie (partially) in a range r we first locate those segments that satisfy condition (i) and next we locate those that satisfy condition (ii). We split the boundary of r in its four line segments, s_t the top line segment, s_b the bottom line segment, s_l the left line segment and s_r the right line segment. The segments in V that have a point in common with s_t or s_b can be determined using the structure described in the previous section. One can easily construct a structure, completely similar to the one described in Section 2 for locating those line segments that have a point in common with a given vertical line segment. On this structure we perform a query with s_l and s_r. As a line segment in V might have more than one point in common with the boundary of the range some care has to be taken that segments are not reported more than once. See the remark below. Hence, the k line segments that have a point in common with the boundary of the range r can be located in $O(\log^2 n+k)$ time.

The line segments that satisfy condition (ii) can be located by an ordinary range query with r on the set of mid-points. Note that, as line segments are not allowed to intersect and were not degenerated

to points, all mid-points are different. Hence, we can use one of the structures for range searching described in the introduction yielding e.g. a query time of $O(\log n+k)$, an update time of $O(\log^2 n)$, using $O(n \log n)$ storage. As noted above, some care has to be taken that line segments are reported only once. Using the method described, some line segment might be located up to five times (e.g., when it goes through two corners of the range and has its mid-point in the range). This can be avoided in the following way. When we find some line segment s_i to be reported, we first check whether it will be found later again. As we perform the five queries (with s_t, s_b, s_l and s_r and the range query) in a fixed order, this can be done in $O(1)$ time by looking at the way s_i intersects the range r. When s_i will not be found later again we report it now. Otherwise we do not report it.

Combining Theorem 2.2. with the results known for ordinary range searching we obtain the following result:

Theorem 3.1. The range searching (windowing) problem for a set of n non-intersecting but possibly touching line segments in the plane can be solved within

query time : $O(\log^2 n+k)$,
insertion time : $O(\log^2 n)$,
deletion time : $O(\log^2 n)$,
storage : $O(n \log n)$,

where k is the number of reported answers.

4. Moving.

As indicated above the range searching problem is the same as the windowing problem considered in graphics. After windowing a set of objects it is often required that the window can be moved over the set of objects in the plane, keeping track of the objects currently visible within the window. We will only allow moving of the window parallel to one of the coordinate-axes. We must be able to locate the first change in the set of visible objects. With a change we mean one of the following cases:

(i) a line segment that is not visible becomes partially visible,
(ii) a line segment that is partially visible becomes fully visible,
(iii) a line segment that is fully visible becomes partially visible,

(iv) a line segment that is partially visible becomes invisible,

(v) a line segment that is partially visible start or stops intersecting some bordering line of the window.

See figure 6 for examples of the cases. A change occurs when either

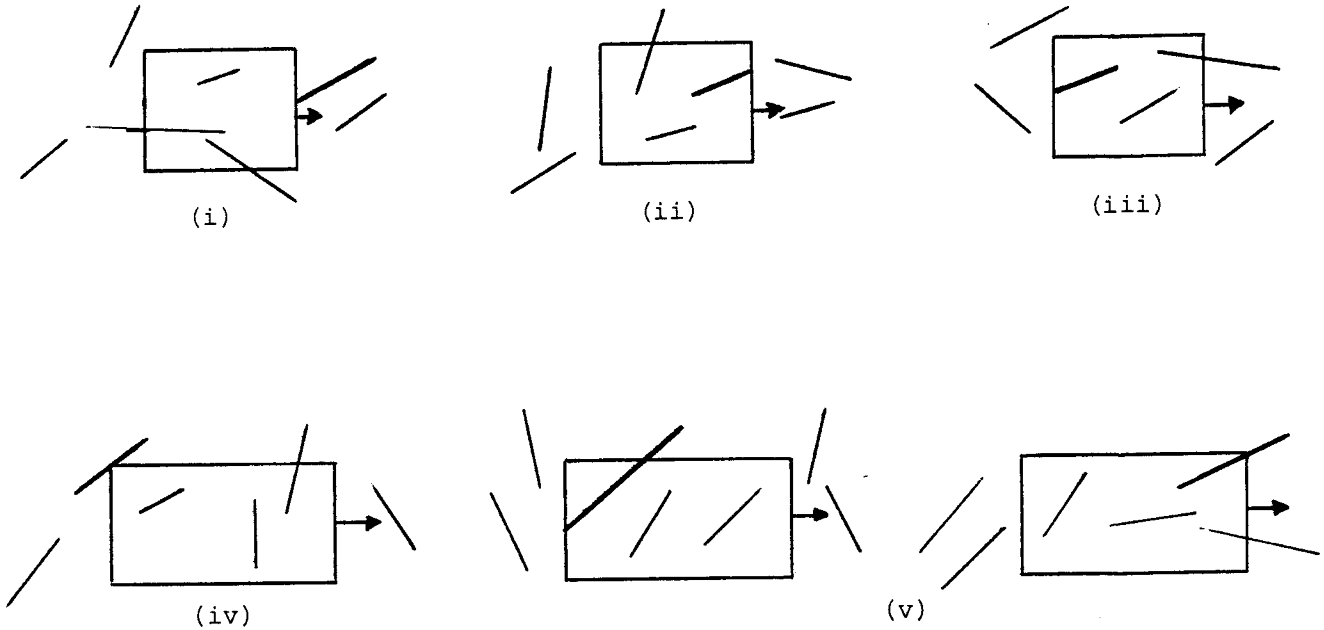

figure 6.

one of the four corners of the window encounters a line segment somewhere in between the endpoints or the left or right border encounters an endpoint of a line segment. We will only describe the method for moving the window to the right. The method can easily be adapted to solve the problem for any of the other three directions. We have reduced the problem to the following two subproblems: (i) given a set of non-intersecting line segments on a point p, determine the first line segment we encounter (not in an endpoint) when moving p to the right, and (ii) given a set of points and a vertical line segment s, determine the first point(s) we encounter when moving s to the right. The structure described in Section 2.3. can easily be adapted to solve the first subproblem. Consider the horizontal line segment s_p that starts at p and runs to the right upto infinity. We search with this segment in the structure S, but, when we search in some associated structure S_β we do not report all line segments that have a point in common with s_p but only the first one, or, if the first segment has p as the common point, we report the second one. This can easily be done and the query time becomes $O(\log^2 n)$. In this way we obtain O(log n) line segments that have a point in common with s_p. These O(log n) segments we check to locate

the one that is hit first when moving p to the right. This adds another O(log n) to the query time. Performing this query for all four corners we find the segment that is first hit by one of the corners of the window in $O(\log^2 n)$ time. (Possibly upto 4 line segments are hit at the same moment. In such cases we report them all.) Let this segment be s_i and let the distance we have to move the window to hit s_i be δ. We will now solve subproblem (ii) but we have to be careful. The left or right border of the window might hit many points at the same moment. These points will be reported all which might take a lot of time. This is no problem when they are indeed involved in the first change of the situation visible in the window but, when this happens after the window has moved more than δ we should not spend this amount of time now. Therefore we will first determine only one of the points we first hit when moving the left and right border to the right. Let us first consider the left border s_l. To determine one of the first points s_l hits when moving it to the right we can as well ask for one of the leftmost points in the range with leftborder s_l, running upto infinity, that is not on s_l itself. The structure of Edelsbrunner [1] can be modified to give such a point in O(log n) time. Let the distance from s_l to the point found be δ_l. In a similar way we can determine δ_r that is the distance the right border s_r has to move to reach a point. Let us look at the minimum of δ, δ_l and δ_r. When δ is the minimum, we report the first line segment s_i hit by one of the corners. When δ_l is the minimum, we again perform a query with the range starting at s_l, upto infinity, and report all leftmost points in the range. For each point we report all line segments that are incident with it. This can easily be done in O(log n+k) time were k is the number of line segments involved in the first change. Similar when δ_r is the minimum. This leads to the following result:

Theorem 4.1. There exists a data structure that stores a set of n non-intersecting line segments in the plane such that for each window the first k line segments involved in the first change when moving the window parallel to one of the coordinate-axes can be determined in

query time : $O(\log^2 n+k)$.

The structure has

insertion time : $O(\log^2 n)$,
deletion time : $O(\log^2 n)$,
storage : $O(n \log n)$.

5. Conclusions and extensions.

We have solved the range searching (windowing) problem for a set of non-intersecting line segments dynamically within a query time of $O(\log^2 n+k)$, an update time of $O(\log^2 n)$ using $O(n \log n)$ storage, where k is the number of reported answers. This result can be extended to work for sets of other objects as well. For the moment we call a convex object simple if its intersection with any horizontal or vertical line segment can be determined in $O(1)$ time. Some examples of simple convex objects are line segments, circles, convex polygons with a bounded number of edges, etc. The structure described in Section 2 can easily be adapted to work for sets of non-intersecting simple convex objects as well. Again we build a segment tree out of the y-projections of all objects and we associate structures to internal nodes in essentially the same way. (Some care has to be taken when objects touch each other.) In this way we can find all objects that have a point in common with the boundary of the range. To find the objects that lie completely in the range we choose in each object a point (for example its leftmost point) and perform a range query on those points. We can again extend the class of possible objects by allowing objects to be composed of a bounded number of simple convex objects. Some examples are the boundary of a triangle (composed of three line segments), a star (composed of some triangles) etc. These objects we split in their simple convex parts and these parts we store in the structure we described. The only problem is that we have to take care that each object visible in the range is reported only once. This can easily be done by checking each time an object is reported whether it will be reported later again (because some other part of it is visible as well). This takes $O(1)$ time by looking at the way the object is visible within the range. If it will be reported later we do not report it now, otherwise we do. We can go on extending the class further. For example, the objects need only be convex with respect to the x- and y-direction, i.e., each horizontal and vertical line segment with endpoints in the object should lie completely in the object. This allows objects like the boundary of a circle (composed of four parts that are convex with respect to the x- and y-direction), circle arcs, etc. For sets of these kinds of objects we can still obtain a query time bound of $O(\log^2 n+k)$, an update time bound of $O(\log^2 n)$ using $O(n \log n)$ storage.

As a second problem we solved the moving problem for a set of line segments in $O(\log^2 n+k)$ query time, $O(\log^2 n)$ update time, using $O(n \log n)$ storage, where k is the number of line segments involved in the first

change. This result can be extended to sets of other objects in the same way as described above.

The moving problem has an important application, namely: dynamic point location. Given a subdivision of the plane in polygonal regions, the point location problem asks for the region a given query point p lies in. The problem was solved statically by Kirkpatrick [5] who obtained a query time of $O(\log n)$, a building time of $O(n \log n)$ using $O(n)$ storage. Note that the region p lies in is fully determined by the first line segment we hit when moving p to the right. Hence, keeping with each line segment (and endpoint) the region to the left of it, we can determine the region p lies in by using the moving technique described in Section 4 with p as (degenerated) range. The query time becomes $O(\log^2 n)$. The structure allows for insertions and deletions of line segments in the subdivision, but some care has to be taken. When we create new regions or combine old ones, we have to revise the regions stored at the bounding line segments. This might take a lot of time when regions are large. Therefore we must assume that each region has only a bounded number of line segments on its boundary. In this case we can insert and delete line segments (creating and combining regions) in $O(\log^2 n)$ time.

A number of open problems remains. An important (theoretical) question is whether the techniques can be extended to work in higher dimensional space as well. Another important question concerns the concept of zooming. It asks for determining the first change of the set of objects visible when the window is reduced or enlarged. Edelsbrunner, Overmars and Seidel [4] solve the problem statically for fixed shaped windows. Overmars and Edelsbrunner [8] solve the problem dynamically for arbitrary windows but their technique is only applicable to sets of points. The problem for sets of line segments remains open. A third question is whether the query time of $O(\log^2 n+k)$ can be reduced to $O(\log n+k)$ (as for range searching on a set of points).

References.

[1] Edelsbrunner, H., A note on dynamic range searching, Bull. of the EATCS 15 (1981), 34-40.

[2] Edelsbrunner, H. and H.A. Maurer, On the intersection of orthogonal objects, Inform. Proc. Lett. 13 (1981) 177-181.

[3] Edelsbrunner, H. and H.A. Maurer, A space-optimal solution of general region location, Theor. Comp. Sci. 16 (1981) 329-336.

[4] Edelsbrunner, H., M.H. Overmars and R. Seidel, Some methods of computational geometry applied to computer graphics, to appear.

[5] Kirkpatrick, D.G., Optimal search in planar subdivisions, SIAM J. Computing 12 (1983) 28-35.

[6] Lueker, G.S., A transformation for adding range restriction capability to dynamic data structures for decomposable searching problems, Techn. Rep. 129, Dept. of Computer Science, University of California, 1979.

[7] McCreight, E.M., Priority search trees, Techn. Rep. CSL-81-5, XEROX Palo Alto research centre, 1981.

[8] Overmars, M.H. and H. Edelsbrunner, Zooming with arbitrary sized windows, to appear.

[9] Overmars, M.H. and J. van Leeuwen, Worst-case optimal insertion and deletion methods for decomposable searching problems, Inform. Proc. Lett. 12 (1981), 168-173.

[10] van Leeuwen, J., Graphics and computational geometry, Les Mathematiques de l'Informatique, Colloq. AFCET, 1982, 159-165.

[11] Willard, D.E., The super-B-tree algorithm, Techn. Rep. TR-03-79, Aiken Computer Lab., Harvard University, 1979.