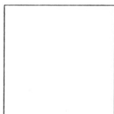
MAX-PLANCK-INSTITUT FÜR INFORMATIK

Dynamic rectangular point location, with an application to the closest pair problem

Michiel Smid

MPI–I–91–101 March 1991



Im Stadtwald
W 6600 Saarbrücken
Germany

Dynamic rectangular point location, with an application to the closest pair problem*

Michiel Smid
Max-Planck-Institut für Informatik
D-6600 Saarbrücken, Germany

March 11, 1991

Abstract

In the k-dimensional rectangular point location problem, we have to store a set of n non-overlapping axes-parallel hyperrectangles in a data structure, such that the following operations can be performed efficiently: point location queries, insertions and deletions of hyperrectangles, and splitting and merging of hyperrectangles. A linear size data structure is given for this problem, allowing queries to be solved in $O((\log n)^{k-1} \log\log n)$ time, and allowing the four update operations to be performed in $O((\log n)^2 \log\log n)$ amortized time. If only queries, insertions and split operations have to be supported, the $\log\log n$ factors disappear. The data structure is based on the skewer tree of Edelsbrunner, Haring and Hilbert and uses dynamic fractional cascading.

This result is used to obtain a linear size data structure that maintains the closest pair in a set of n points in k-dimensional space, when points are inserted. This structure has an $O((\log n)^{k-1})$ amortized insertion time. This leads to an on-line algorithm for computing the closest pair in a point set in $O(n(\log n)^{k-1})$ time. In the planar case, these two latter results are optimal.

1 Introduction

The point location problem is one of the problems in computational geometry that has received considerable attention. In this problem, we have to store a subdivision of k-dimensional space in a data structure, such that for a given query point, we can find the region that contains it. Many data structures have been proposed for this problem, especially for the planar version. See e.g. the books of Preparata and Shamos [8] and Edelsbrunner [4].

In this paper, we consider the case where the subdivision consists of a collection of non-overlapping k-dimensional axes-parallel hyperrectangles, or k-boxes for

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

short. Edelsbrunner, Haring and Hilbert [5] considered the static case of this problem and introduced the skewer tree for solving it. In the present paper, we show how this skewer tree can be adapted such that k-boxes can be inserted, deleted, split and merged. We also equip the skewer tree with dynamic fractional cascading (see Mehlhorn and Näher [6]) to speed up the query algorithm. The result is a data structure of linear size, that allows point location queries to be solved in $O((\log n)^{k-1} \log\log n)$ time, such that the four update operations can be carried out in $O((\log n)^2 \log\log n)$ amortized time. For the special case, where only insertions and split operations have to be supported the $\log\log n$ factors can be omitted.

In the second part of this paper, we apply the skewer tree to obtain an efficient data structure for the closest pair problem. In this problem, we are given a set of points in k-dimensional space and we have to compute, or maintain, the closest pair. For the static case, the closest pair can be computed in $O(n \log n)$ time, which is optimal. (See [8,13].) For the dynamic case, there are data structures by Dobkin and Suri [3] and Smid [10] that can handle semi-online updates in $O((\log n)^2)$ time using linear space, for the planar case. Supowit [12] gives a structure that performs deletions in $O((\log n)^k)$ amortized time using $O(n(\log n)^{k-1})$ space. Finally, Smid [9,11] gives two data structures for fully on-line updates. The first one has linear size and performs updates in $O(n^{2/3} \log n)$ time, whereas the second one has $O(n(\log n)^{k-1})$ size and performs updates in $O((\log n)^{k+2})$ amortized time.

For the case where only points are inserted, no better results are known. In this paper, it is shown how the structure for rectangular point location can be used to obtain a linear size data structure that maintains the closest pair in $O((\log n)^{k-1})$ amortized time per insertion. In the planar case, this gives an optimal data structure.

As an application, this leads to an on-line algorithm that computes the closest pair in a point set in $O(n(\log n)^{k-1})$ time. Again, in the planar case this is optimal.

The rest of this paper is organized as follows. In Section 2, we define the skewer tree for the two-dimensional version of the rectangular point location problem. In that section, we only consider the operations point location, insert and split. The balance condition for this data structure is non-standard, although it resembles that of BB[α]-trees. The method of keeping the skewer tree balanced is a new variation of the partial rebuilding technique. In order to speed up the query algorithm, we equip the skewer tree with a version of dynamic fractional cascading, where only insertions have to be supported.

In Section 3, we analyze the amortized time of the insert and split algorithms. The main difficulty is in proving that the amortized time for insert and split operations is bounded by $O((\log n)^2)$. It turns out that the amortized rebalancing costs dominate the overall update time. In Section 4, we generalize the skewer tree to the k-dimensional case, using standard methods. Then, in Section 5, we show how the skewer tree can be adapted such that delete and merge operations can be supported as well. Since we need fully dynamic fractional cascading here, the time complexities increase by a factor of $O(\log\log n)$.

In the second part of the paper, we consider the dynamic closest pair problem,

where points are inserted. In Section 6, we give a data structure for this problem that uses the skewer tree as a substructure. The data structure maintains a collection of k-dimensional boxes having sides of length at least the current minimal distance. Each box contains a limited number of points. If a point is inserted, we only have to compare the new point with the points that are contained in a constant number of surrounding boxes. If a box contains too many points, it is split into a constant number of boxes, each of which has sides of length at least the current minimal distance.

We finish the paper in Section 7 with some concluding remarks and open problems.

2 Rectangular point location, the planar case

We first consider a special case of the planar rectangular point location problem. We are given a set of n axes-parallel rectangles that do not overlap, i.e., the interiors of the rectangles are pairwise disjoint. Each rectangle has the form $[a_1 : b_1] \times [a_2 : b_2]$. Rectangles may be infinite, i.e., $a_1, a_2 \in \mathcal{R} \cup \{-\infty\}$ and $b_1; b_2 \in \mathcal{R} \cup \{+\infty\}$. The rectangles do not necessarily partition the entire plane. We want to store these rectangles in a data structure such that the following four operations can be carried out efficiently.

Point location: Given a query point p, find the rectangles—if any—that contain p. If p lies on the boundary of a rectangle, there may be several rectangles that contain p. Since the rectangles do not overlap, a query point is contained in at most four rectangles.

Insertion: Insert an axes-parallel rectangle. The new set of rectangles must still be non-overlapping.

Horizontal split: The operation $hsplit(s)$ replaces rectangle $[a_1 : b_1] \times [a_2 : b_2]$ by the rectangles $[a_1 : b_1] \times [a_2 : s]$ and $[a_1 : b_1] \times [s : b_2]$. This operation is defined for bounded as for unbounded rectangles. That is, we assume that $-\infty \leq a_1 < b_1 \leq \infty$ and $-\infty \leq a_2 < s < b_2 \leq \infty$. See Figure 1.

Vertical split: The operation $vsplit(t)$ replaces rectangle $[a_1 : b_1] \times [a_2 : b_2]$ by the rectangles $[a_1 : t] \times [a_2 : b_2]$ and $[t : b_1] \times [a_2 : b_2]$. Again, this operation is defined for bounded as for unbounded rectangles. That is, we assume that $-\infty \leq a_1 < t < b_1 \leq \infty$ and $-\infty \leq a_2 < b_2 \leq \infty$. See Figure 1.

Edelsbrunner, Haring and Hilbert [5] introduced the skewer tree for the static version of this problem.

The skewer tree: Let V be a set of non-overlapping axes-parallel rectangles. The skewer tree is recursively defined as follows.

1. If V is empty, the skewer tree is also empty.

2. Assume that V is non-empty. Let l be a vertical line that intersects at least one rectangle of V in its interior. The skewer tree for the set V consists of a

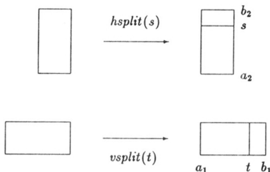


Figure 1: Horizontal and vertical splits.

binary tree—called the *skeleton tree*—in which each node contains additional information:

(a) In the root r of the tree, we store the size of V, the line l and (a pointer to) a balanced binary search tree T_r that is defined as follows. Let V_r be the set of all rectangles in V that are intersected by l in their interiors or that touch l with their right boundaries. Let W_r be the set of y-coordinates of the top and bottom sides of the rectangles in V_r. (Each y-coordinate is represented exactly once.) For convenience, we add $-\infty$ and $+\infty$ to W_r. Then T_r stores the values of W_r in increasing order. With each value s, we store the rectangle *below*(s) resp. *above*(s), which is the rectangle in V_r that has its top resp. bottom side at height s. If *below*(s) or *above*(s) does not exist, then the value of this variable is *nil*.

(b) The root r has two subtrees. The left subtree is a skewer tree for all rectangles in V that lie completely to the left of l. Similarly, the right subtree is a skewer tree storing all rectangles in V that lie completely to the right of l or that touch l with their left boundaries.

The *height* of a skewer tree is defined as the height of its skeleton tree. In order to guarantee that this height is logarithmic in the number of rectangles, we require the following condition.

Balance condition: Let α be a real number such that $1/2 \leq \alpha < 1$. For each node v of the skeleton tree, let n_v be the total number of rectangles that are stored in the subtree of v (including node v itself), and let $d(v)$ be the depth of v in the skeleton tree. (The root has depth 0.) Then we require that $n_v \leq \alpha^{d(v)} n$, where n is the current number of rectangles that are stored in the entire data structure.

Such a skewer tree is called α-*balanced*. If $\alpha = 1/2$, the skewer tree is called *perfectly balanced*.

Remark: A subtree of an α-balanced skewer tree is not necessarily α-balanced.

Also, the root of the skeleton tree always satisfies the balance condition.

The proof of the following lemma is trivial and, therefore, omitted.

Lemma 1 *An α-balanced skewer tree that stores n rectangles has height at most* $\lfloor (\log n)/(\log(1/\alpha)) \rfloor = O(\log n)$.

Point location in an α-balanced skewer tree: Suppose we have to find the rectangles that contain the query point $p = (p_1, p_2)$. Then we start in the root r of the skewer tree. We perform a binary search with the y-coordinate p_2 in the search tree T_r that is stored with r. This gives us two values s and t, such that $s \leq p_2 < t$. If $p_2 > s$ and $p \in above(s)$, then we report this rectangle $above(s)$. Otherwise, if $p_2 = s$, we report those rectangles of $below(s)$ and $above(s)$ that contain p.

If p is contained in the interior of a reported rectangle, then the search procedure is finished. Otherwise, we proceed recursively: If p lies to the left of the line l that is stored in the root, we proceed our search in the left subtree. Otherwise, p lies on or to the right of l, in which case we proceed in the right subtree.

By Lemma 1, the skewer tree has height $O(\log n)$. Therefore, we do a logarithmic number of binary searches with the y-coordinate p_2, each taking $O(\log n)$ time. It follows that the query time is bounded by $O((\log n)^2)$.

We now improve the query time to $O(\log n)$ by using the technique of dynamic fractional cascading. See Chazelle and Guibas [2] and Mehlhorn and Näher [6]. The reader is assumed to be familiar with this data structuring technique. We use the terminology of Mehlhorn and Näher [6].

Dynamic fractional cascading: The *catalogue graph* is the skeleton tree and the range $R(e)$ of each edge e in this catalogue graph is the set of real numbers. The *catalogue* $C(v)$ of a node v in the catalogue graph is the list of y-coordinates that are stored in the binary search tree that is stored with v, extended with $-\infty$ and $+\infty$. Instead of this binary search tree, we store in each node v an *augmented catalogue* $A(v)$ as described in [6]. Note that $C(v) \subseteq A(v)$. These augmented catalogues are implemented as balanced binary search trees. Elements in $C(v)$ are called *proper*, those in $A(v) \setminus C(v)$ are called *non-proper*. Each proper element s contains the values $below(s)$ and $above(s)$ that have the same meaning as before. There are *bridges* between the augmented catalogues of adjacent nodes of the catalogue graph, as described in [6]. Finally, we store in each node v a data structure that solves the SPLIT-FIND problem. This structure is used for finding proper elements that are next to non-proper elements. See [6].

Building a perfectly balanced skewer tree: Given the set V of n non-overlapping rectangles, we first compute the median of the x-coordinates of their right boundaries. Let l be the vertical line having its x-coordinates equal to this median. Then we partition V into three sets $V_<$, V_l resp. $V_\geq$, consisting of the rectangles that lie completely to the left of l, that are intersected by l in their

interiors or touch l with their right boundaries resp. that lie completely to the right of l or touch l with their left boundaries.

Next, we create the root r of the skewer tree, in which we store the line l, the size of V, a list containing the rectangles in V_l in sorted y-order, an empty augmented catalogue $A(r)$ and an empty SPLIT-FIND structure. Finally, we repeat this algorithm for the sets $V_<$ and $V_\geq$, leading to augmented binary trees that become the left and right sons of r.

In this way, we have built the skeleton tree of the final skewer tree. Each node contains a sorted list of the rectangles that have to be stored there. To finish the building algorithm, we do the following for each node v of the skeleton tree: Insert—in increasing order—the y-coordinates of the top and bottom sides of the rectangles that are stored with v, and the values $-\infty$ and $+\infty$, into the augmented catalogue $A(v)$. With each y-value s, we store the corresponding values of $below(s)$ and $above(s)$.

Lemma 2 *An α-balanced skewer tree, equipped with fractional cascading, has size $O(n)$ and can be built in $O(n \log n)$ time.*

Proof: A skewer tree that is not equipped with fractional cascading has size $O(n)$, because the skeleton tree has linear size, and each rectangle is stored only twice, once as an *above*-value and once as a *below*-value. In [6], it is shown that dynamic fractional cascading increases the space complexity by at most a constant factor. This proves that the data structure has linear size.

To prove the bound on the building time, first note that the above algorithm builds a perfectly balanced skewer tree.

Using a linear time median algorithm (see [1]), the skeleton tree and the partition of V such that each rectangle is stored at the correct node can be computed in $O(n \log n)$ time. If n_v rectangles are stored at node v, then it takes $O(\sum_v n_v \log n_v) = O(n \log n)$ time to order these rectangles according to their heights. After this has been done, we have to insert the $O(n)$ y-coordinates into the augmented catalogues. Since we insert these values in increasing order, each value knows its position already. In [6], it is shown that each such insertion takes $O(1)$ amortized time, because the SPLIT-FIND structure needs only constant amortized time per operation. Therefore, this final step of the algorithm takes $O(n)$ time. This proves that the entire building algorithm takes $O(n \log n)$ time. ∎

Point location using fractional cascading: To answer a query with query point $p = (p_1, p_2)$, we use basically the same algorithm as before, except that we only do a binary search in the augmented catalogue that is stored with the root of the skewer tree. Afterwards, we follow bridges and use the SPLIT-FIND structures to locate p_2 in the appropriate augmented catalogues. See [6] for details.

Lemma 3 *In an α-balanced skewer tree, equipped with fractional cascading, a point location query can be answered in $O(\log n)$ time.*

Proof: First note that the query algorithm is correct. We only sketch the proof of the time bound. For more details, see [6]. Let $v_0, v_1, \ldots, v_m$ be the path in the skeleton tree that is followed with the x-coordinate of the query point $p = (p_1, p_2)$. Note that $m = O(\log n)$. Clearly, this path can be computed in $O(\log n)$ time. We have to locate the y-coordinate p_2 in each catalogue $C(v_i)$, $i = 1, 2, \ldots, m$, and report all rectangles that contain p. The time needed to locate p_2 in the *augmented* catalogue of the root is bounded by $O(\log n)$. Then we have to perform a FIND-operation to locate p_2 in the catalogue itself. This FIND-operation takes constant time in the worst case. Once p_2 is located in the augmented catalogue of the root, it can be located in the other augmented catalogues $A(v_i)$, $i = 2, \ldots, m$, in $O(1)$ time per augmented catalogue. Finally, by performing FIND-operations, p_2 can be located in each catalogue $C(v_i)$, $i = 2, \ldots, m$, in $O(1)$ time per catalogue. It follows that the total query time is bounded by $O(m + \log n) = O(\log n)$. ■

Next, we give the algorithms for the insert and split operations.

Insertion: To insert a rectangle R into an α-balanced skewer tree, we do the following: We start in the root of the skeleton tree, and follow a path until we reach the first node v such that the vertical line that is stored in this node intersects R in its interior or touches the right boundary of R. In each node that is visited during this walk, we increase the number of rectangles that are stored in its subtree by one. Then we do a binary search in the augmented catalogue $A(v)$ to locate the position where R has to be inserted. Then, we insert the y-coordinates of the top and bottom sides of R—together with the appropriate values for *below* and *above*—into $A(v)$, as described in [6]. If these y-coordinates are present already, we only update the appropriate *below*- and *above*-values.

If we do not find node v, we end our search in a node w of the skeleton tree, one of whose sons is missing—namely the one to which the search wants to proceed. In this case, we give w a new son, i.e., we insert a node u with an empty catalogue $C(u)$ and an edge (w, u) into the catalogue graph, as described in [6]. This will give node u a non-empty augmented catalogue $A(u)$. Then, we store in u a vertical line that intersects the interior of R and the number of rectangles that are stored in u—which is equal to one. Finally, we insert into $A(u)$, the y-coordinates of the top and bottom sides of the rectangle R and the values $-\infty$ and $+\infty$, as described in [6], together with the appropriate *above*- and *below*-values.

The problem of rebalancing the skewer tree is considered below.

Horizontal split: To perform the operation *hsplit*(s) on the rectangle $R = [a_1 : b_1] \times [a_2 : b_2]$, we first search for the node v in the skeleton tree, whose augmented catalogue "contains" R. In each node that is visited during this walk, we increase the number of rectangles that are stored in its subtree by one. Then, we do a binary search in the augmented catalogue $A(v)$ to locate the positions of a_2 and b_2. We replace the rectangle *above*(a_2) resp. *below*(b_2) by the lower resp. upper part of the rectangle R. Finally, we insert the y-coordinate s into $A(v)$, as described in [6],

together with the appropriate values *below*(s) and *above*(s).

Later, we consider the problem of rebalancing the skewer tree.

Vertical split: To perform the operation *vsplit*(t) on the rectangle $R = [a_1 : b_1] \times [a_2 : b_2]$, we again first search for the node v in the skeleton tree, whose augmented catalogue "contains" R. Let l be the vertical line that is stored in v. Assume that l intersects the left part of the rectangle R in its interior or touches its right boundary. (The symmetric case is treated analogously.) We search in the augmented catalogue $A(v)$ for the values a_2 and b_2, and replace the rectangles *above*(a_2) and *below*(b_2) by the left part of R. Then, we use the above insertion algorithm to insert the right part of R into the skewer tree.

The problem of rebalancing the skewer tree is considered below.

Rebalancing the skewer tree: After an insert or split operation, the skewer tree might not satisfy the balance condition anymore. To keep the skewer tree balanced, we use a variation of the partial rebuilding technique (see e.g.[7]):

During the update operation, we have inserted y-coordinates in the augmented catalogue of exactly one node. Starting in that node, we walk back to the root of the skeleton tree and find the highest node w that does not satisfy the balance condition of the α-balanced skewer tree anymore. (Note that the root of the skeleton tree never gets out of balance.) Then we rebuild the complete subtree rooted at the *father of* w as a perfectly balanced skewer tree. More precisely, we do the following:

If the father of w is the root of the skeleton tree, then we rebuild the complete data structure as a perfectly balanced skewer tree. Otherwise, let v be the father of w and let u be the father of v. We delete the edge between u and v from the catalogue graph. (This removes all bridges between the augmented catalogues $A(u)$ and $A(v)$.) Then, we rebuild the SPLIT-FIND structure corresponding to $A(u)$.

Next, we perform the first part of the building algorithm to construct the skeleton tree of a perfectly balanced skewer tree for the rectangles that are stored in the subtree of v. Then we have in each node of this skeleton tree, a sorted list of the y-coordinates that have to be stored there.

We add this skeleton tree S_v to the old skewer tree, as follows: We insert the root of S_v and an edge between u and this root into the catalogue graph. Then we insert the nodes and edges of S_v into the catalogue graph, as described in [6]. (At this moment, all catalogues of the nodes in S_v are empty. The augmented catalogues, however, are non-empty.) For each node of S_v, we insert the y-coordinates of the top and bottom sides of the rectangles that are stored there, and the values $-\infty$ and $+\infty$, into its augmented catalogue, one after another, in increasing order. We also insert the appropriate *above*- and *below*-values.

In Section 3, we analyze the amortized time complexity of the insert and split operations. We mention here that the above update algorithms correctly maintain the α-balanced skewer tree. In particular, it remains true that for each node v in the skeleton tree, all rectangles in its left resp. right subtree lie completely to the left of

l resp. lie completely to the right of l or touch l with their left boundaries, where l is the vertical line that is stored in v. Also, the only nodes that might get out of balance during an update operation must lie on the search path to the node where the update was performed. This is because the value of n only increases. Finally, it is shown in Lemma 6 that after a rebalancing operation, the resulting skewer tree is again α-balanced.

We finish this section by stating the complexity of the α-balanced skewer tree.

Theorem 1 *For the problem of point location in a set of n non-overlapping planar axes-parallel rectangles, there exists a data structure that has a query time of $O(\log n)$, in which insert and split operations take $O((\log n)^2)$ amortized time, that can be built in $O(n \log n)$ time, and that has size $O(n)$.*

Proof: The bounds on the query time, building time and size follow from Lemmas 2 and 3. The bound for the amortized update time will be proved in Section 3. ∎

3 Analysis of the insert and split operations

In this section, we complete the proof of Theorem 1. First, we prove an upper bound on the size of subtrees of the skewer tree.

Lemma 4 *Let u be a node in the skeleton tree of an α-balanced skewer tree and let $d(u)$ be the depth of this node. Then the entire subtree of u, i.e., the subtree of the skeleton tree rooted at u, together with the augmented catalogues that are stored in the nodes of this subtree, has size $O(\alpha^{d(u)} n)$. Here, n is the number of rectangles that are stored in the entire skewer tree.*

Proof: Let $st(u)$ denote the subtree of the skeleton tree rooted at u. By definition of α-balancedness, $st(u)$ stores at most $\alpha^{d(u)} n$ rectangles. Therefore, this subtree has at most this number of nodes. So it remains to show that the augmented catalogues that are stored in the nodes of $st(u)$ have size $O(\alpha^{d(u)} n)$.

For a node u, define $f^1(u) := u$, if u is the root of the skeleton tree, and $f^1(u) :=$ the father of u in the skeleton tree, otherwise. For $k \geq 1$, define $f^{k+1}(u) := f^1(f^k(u))$.

We prove that there is a constant c, such that for any positive integer k and any node u of the skeleton tree,

$$\sum_{v \in st(u)} |A(v)| \leq \sum_{i=0}^{k-1} c \left(\frac{6}{a}\right)^i \alpha^{d(u)-i} n \ + \ \left(\frac{6}{a}\right)^k \sum_{v \in st(f^k(u))} |A(v)|. \qquad (1)$$

In this equation, a is a constant that occurs in the analysis of dynamic fractional cascading. See [6]. It will turn out that $a > 6/\alpha$ suffices.

The proof of (1) is by induction on k. The basis of the induction is the case $k = 1$. Let $C(v)$ be the catalogue that belongs to node v, i.e., the list consisting of the proper elements of the augmented catalogue $A(v)$. Let u be a node in the

skeleton tree. Our balance condition guarantees that $\sum_{v\in st(u)} |C(v)| \leq c'\alpha^{d(u)}n$, for some constant c'. Let $B(v,w)$ denote the number of bridges between the augmented catalogues $A(v)$ and $A(w)$. Dynamic fractional cascading guarantees that $B(v,w) \leq 2+(|A(v)|+|A(w)|)/a$, see [6, page 219]. Let E denote the set of edges in the skeleton tree. Note that a non-root and non-leaf node in the skeleton tree is connected to 3 other nodes. We have

$$\begin{aligned}
\sum_{v\in st(u)} |A(v)| &= \sum_{v\in st(u)} |C(v)| + \sum_{v\in st(u)} \sum_{(v,w)\in E} B(v,w) \\
&\leq c'\alpha^{d(u)}n + \sum_{v\in st(u)} \sum_{(v,w)\in E} \left(2 + \frac{|A(v)|+|A(w)|}{a}\right) \\
&\leq c'\alpha^{d(u)}n + 6\,|st(u)| + \frac{3}{a}\sum_{v\in st(u)} |A(v)| + \frac{1}{a}\sum_{v\in st(u)} \sum_{(v,w)\in E} |A(w)| \\
&\leq c'\alpha^{d(u)}n + 6\,\alpha^{d(u)}n + \frac{3}{a}\sum_{v\in st(u)} |A(v)| + \frac{3}{a}\sum_{v\in st(f^1(u))} |A(v)| \\
&\leq c\,\alpha^{d(u)}n + \frac{6}{a}\sum_{v\in st(f^1(u))} |A(v)|,
\end{aligned}$$

for any constant $c \geq c' + 6$.

We have shown that there is a constant c, such that for any node u of the skeleton tree,

$$\sum_{v\in st(u)} |A(v)| \leq c\,\alpha^{d(u)}n + \frac{6}{a}\sum_{v\in st(f^1(u))} |A(v)|. \tag{2}$$

Hence, we have proved (1) for $k = 1$. Now assume that (1) holds for k and any node u. Then

$$\sum_{v\in st(u)} |A(v)| \leq \sum_{i=0}^{k-1} c\left(\frac{6}{a}\right)^i \alpha^{d(u)-i}\,n \;+\; \left(\frac{6}{a}\right)^k \sum_{v\in st(f^k(u))} |A(v)|.$$

Apply (2) to the rightmost summation. Then we get:

$$\begin{aligned}
\sum_{v\in st(u)} |A(v)| &\leq \sum_{i=0}^{k-1} c\left(\frac{6}{a}\right)^i \alpha^{d(u)-i}\,n \;+\; \left(\frac{6}{a}\right)^k \left(c\,\alpha^{d(f^k(u))}\,n \;+\; \frac{6}{a}\sum_{v\in st(f^{k+1}(u)).} |A(v)|\right) \\
&\leq \sum_{i=0}^{k} c\left(\frac{6}{a}\right)^i \alpha^{d(u)-i}n \;+\; \left(\frac{6}{a}\right)^{k+1} \sum_{v\in st(f^{k+1}(u))} |A(v)|.
\end{aligned}$$

In the last line, we used the inequality $\alpha^{d(f^k(u))} \leq \alpha^{d(u)-k}$. This is in fact an equality if $k \leq d(u)$. If $k > d(u)$, then $d(f^k(u)) = 0$, because $f^k(u)$ is the root of the skeleton tree. Since $1/2 < \alpha < 1$, we have in that case $\alpha^{d(f^k(u))} = 1 \leq \alpha^{d(u)-k}$.

This proves that (1) holds for all values of k. Now we can complete the proof of the lemma. First, we bound the first summation on the right-hand side in (1):

$$\sum_{i=0}^{k-1} c\left(\frac{6}{a}\right)^i \alpha^{d(u)-i}\,n \leq c\,\alpha^{d(u)}\,n\sum_{i=0}^{\infty}\left(\frac{6}{a\alpha}\right)^i = O\left(\alpha^{d(u)}n\right),$$

because $a > 6/\alpha$. The second summation on the right-hand side in (1) is bounded above by $(6/a)^k$ times the size of the entire skewer tree. Hence, it is bounded above by $(6/a)^k O(n)$. This holds for every positive k. Take k sufficiently large such that $(6/a)^k O(n) = O(1)$. Then

$$\sum_{v \in st(u)} |A(v)| = O(\alpha^{d(u)} n) + O(1) = O(\alpha^{d(u)} n).$$

Hence, the augmented catalogues of the nodes in $st(u)$ have size $O(\alpha^{d(u)} n)$. This finishes the proof. ∎

In the next lemma, we bound the time needed in a rebalancing operation. Note that this time bound is not a function of the number of rectangles that are stored in the rebuilt subtree. This number can be much smaller.

Lemma 5 *Suppose that during an insert or split operation, we rebuild a subtree with root v. This rebuilding takes $O(\alpha^{d(v)} n \log n)$ time.*

Proof: If v is the root of the skeleton tree, the time bound follows from Lemma 2. So assume that v is not the root. Let m be the number of rectangles that are stored in the subtree rooted at v, and let u be the father of v. Consider the rebalancing algorithm.

By Lemma 9 in [6], the edge between u and v can be deleted from the catalogue graph in $O(|A(u)| + |A(v)|)$ time. The SPLIT-FIND structure for $A(u)$ can be rebuilt in $O(|A(u)|)$ time. (Note that there is no log log-factor here, because we only need a structure for the SPLIT-FIND problem. In the case where also deletions are possible, we need a UNION-SPLIT-FIND structure, introducing a doubly-logarithmic term.)

By Lemma 2, it takes $O(m \log m)$ time to build the skeleton tree, together with the sorted list of rectangles in the nodes, for the m rectangles.

To add this skeleton tree $st(v)$ to the complete skewer tree, we insert nodes and edges into the catalogue graph. By Lemma 9 in [6], this takes time proportional to

$$|A(u)| + |A(v)| + \sum_{x \in st(v)} \sum_{(x,y) \in E} (|A(x)| + |A(y)|),$$

where E is the set of edges in $st(v)$. This expression is bounded by the summation $O(\sum_{x \in st(u)} |A(x)|)$, which—by Lemma 4—is bounded by $O(\alpha^{d(u)} n) = O(\alpha^{d(v)} n)$.

Finally, we insert the $O(m)$ y-coordinates into the augmented catalogues. Since we know the position where each y-coordinate is inserted, and because we only have a SPLIT-FIND structure, each of these insertions takes $O(1)$ amortized time. It follows that the total time for this final step of the algorithm is bounded by $O(m)$.

We have shown that the total time for the rebalancing operation is bounded by $O(m \log m + \alpha^{d(v)} n + m)$, which is bounded by $O(\alpha^{d(v)} n)$, because—by the balance condition—$m \leq \alpha^{d(v)} n$. (Note that we rebuild the subtree rooted at v, because the subtree rooted at one of its sons was the highest node out of balance. Therefore, the upper bound on m holds.) ∎

Next, we show that the rebalancing algorithm indeed results in an α-balanced skewer tree, and that expensive rebuilding operations seldom occur. If during an update, node w is the highest node that is out of balance, then we say that node w *causes* the rebalancing operation.

Lemma 6 *The rebalancing algorithm correctly rebuilds an α-balanced skewer tree. Furthermore, let w be a node in the skeleton tree of an α-balanced skewer tree, and assume that during the current update, w causes a rebalancing operation. Let n be the number of rectangles that are stored in the entire data structure at this moment. Then, if node w causes a rebalancing operation again, there must have been at least $(2\alpha - 1)/(2\alpha)\,\alpha^{d(w)}\,n$ updates in the subtree of w.*

Proof: Since node w causes the rebalancing operation, we rebuild the subtree rooted at its father v. (Note that the root of the skeleton tree never causes a rebalancing operation. Therefore, node v exists.) Let m be the number of rectangles that are stored in the subtree of v.

At the moment of the rebalancing operation, node v is not out of balance. Therefore, $m \leq \alpha^{d(v)}\,n$.

Consider a node $u \neq v$ in the skeleton tree of the rebuilt subtree. Since we rebuild a structure as a perfectly balanced skewer tree, there are at most $(1/2)^{d'(u)}\,m$ rectangles in the subtree rooted at u. Here, $d'(u)$ is the depth of u in the subtree rooted at v. It follows that the number of rectangles that are stored in the subtree of u is at most

$$\left(\frac{1}{2}\right)^{d'(u)} m \leq \left(\frac{1}{2}\right)^{d'(u)} \alpha^{d(v)} n = \left(\frac{1}{2\alpha}\right)^{d'(u)} \alpha^{d'(u)+d(v)} n = \left(\frac{1}{2\alpha}\right)^{d'(u)} \alpha^{d(u)} n \leq \frac{1}{2\alpha}\alpha^{d(u)} n,$$

because $d'(u) \geq 1$.

In particular, this number is at most $\alpha^{d(u)} n$. This proves that after the rebalancing operation, the resulting data structure is again α-balanced.

Consider the update where node w causes a rebalancing operation again, and let n' be the total number of rectangles at this moment. Note that $n' \geq n$. At this moment, the subtree rooted at w stores more than $\alpha^{d(w)}\,n' \geq \alpha^{d(w)}\,n$ rectangles. We saw above that after the previous rebalancing operation caused by w, its subtree contained at most $(1/(2\alpha))\alpha^{d(w)} n$ rectangles. It follows that the number of updates in the subtree of w since the previous rebalancing operation must be at least

$$\alpha^{d(w)}\,n - \frac{1}{2\alpha}\,\alpha^{d(w)}\,n = \frac{2\alpha - 1}{2\alpha}\,\alpha^{d(w)}\,n.$$

Note that $(2\alpha - 1)/(2\alpha) > 0$, because $1/2 < \alpha < 1$. This proves the lemma. ∎

Lemma 7 *In an α-balanced skewer tree, insert and split operations can be performed in $O((\log n)^2)$ amortized time.*

Proof: First, we consider the time needed in the update algorithms before a new leaf is added to the skeleton tree and before rebalancing is done. To perform an insert or split operation, we walk down the skeleton tree and do a constant number of binary searches in at most two augmented catalogues. This takes $O(\log n)$ time. Then, the information of the *below*- and *above*-values can be adapted in $O(1)$ time. Finally, we insert a constant number of y-coordinates in an augmented catalogue. In [6], it is shown that this takes $O(1)$ amortized time, because we only have a SPLIT-FIND structure. Hence, this part of the update algorithms takes $O(\log n)$ amortized time.

Consider a fixed node w in the skeleton tree, and consider a sequence of updates that occur in the subtree of w, from the moment that w causes a rebalancing operation until the next moment that w causes a rebalancing operation. (The first moment is included in this sequence, the second one not.) By Lemma 6, this sequence has length $\Omega(\alpha^{d(w)}n)$. During this sequence, w is responsible for one rebuilding of the subtree rooted at its father v. By Lemma 5, this rebuilding takes $O(\alpha^{d(v)}n \log n)$ time. During this sequence, several leafs may have been added to the subtree of w. The time to add these leaves is bounded above by the time to build the entire subtree rooted at w. Hence, this time is also bounded by $O(\alpha^{d(v)}n \log n)$. It follows that this node w contributes an amortized time to the rebalancing complexity that is bounded by

$$O(\alpha^{d(v)}n \log n)/\Omega(\alpha^{d(w)}n) = O(\log n).$$

During an update we visit $O(\log n)$ nodes in the skeleton tree—not counting here nodes visited during rebalancing operations—each contributing $O(\log n)$ amortized time to the update time. This proves that the total amortized update time is bounded by $O((\log n)^2)$. ■

This concludes the analysis of the update algorithms for the skewer tree. Hence, the proof of Theorem 1 is completed.

4 Generalization to higher dimensions

We now generalize the results obtained so far to the k-dimensional case. A *k-dimensional box*, or *k-box* for short, is an axes-parallel hyperrectangle of the form

$$[a_1 : b_1] \times [a_2 : b_2] \times \ldots \times [a_k : b_k],$$

where $a_i \in \mathcal{R} \cup \{-\infty\}$ and $b_i \in \mathcal{R} \cup \{\infty\}$, $i = 1, \ldots, k$. In the k-dimensional rectangular point location problem, we are given a set of non-overlapping k-boxes, on which the following operations have to be performed.

Point location: Given a query point p in k-space, find the boxes—if any—that contain p. Since the boxes do not overlap, a query gives at most 2^k answers.

Insertion: This operation inserts a k-box into the set. The new set of boxes must still be non-overlapping.

Split operation: The operation *i-split*(s) replaces the box $[a_1 : b_1] \times \ldots \times [a_k : b_k]$ by the two boxes

$$[a_1 : b_1] \times \ldots \times [a_{i-1} : b_{i-1}] \times [a_i : s] \times [a_{i+1} : b_{i+1}] \times \ldots \times [a_k : b_k]$$

and

$$[a_1 : b_1] \times \ldots \times [a_{i-1} : b_{i-1}] \times [s : b_i] \times [a_{i+1} : b_{i+1}] \times \ldots \times [a_k : b_k].$$

This operation is defined for $1 \leq i \leq k$ and $a_i < s < b_i$.

The data structure for solving this problem is a direct generalization of the skewer tree of the preceding sections. The static version, without fractional cascading, was introduced in [5].

The k-dimensional skewer tree: Let V be a set of non-overlapping k-boxes. For $k = 2$, a 2-dimensional skewer tree for the set V was defined in Section 2. Let $k > 2$. If V is empty, the skewer tree is also empty.

Assume that V is non-empty. Let $\sigma : x_1 = \beta_1$ be a hyperplane in k-space. Let V_-, V_0, resp. V_+ be the set of boxes $[a_1 : b_1] \times \ldots \times [a_k : b_k]$ in V such that $b_1 < \beta_1$, $a_1 < \beta_1 \leq b_1$ resp. $\beta_1 \leq a_1$. The hyperplane is assumed to be chosen such that V_0 is non-empty.

The k-dimensional skewer tree for the set V is an augmented binary search tree—called the *skeleton tree*—having the following form:

1. There is a root that contains the size of V, the hyperplane σ and (a pointer to) a $(k-1)$-dimensional skewer tree for the set V_0', which is obtained from V_0 by deleting in each k-box the first interval.

2. The root contains pointers to its left and right sons, which are k-dimensional skewer trees for the sets V_- and V_+.

The k-dimensional skewer tree contains 2-dimensional skewer trees as substructures. In these 2-dimensional structures, the *below*- and *above*-values are k-boxes instead of planar rectangles. We equip the 2-dimensional structures with dynamic fractional cascading.

Balance condition: Let $1/2 \leq \alpha < 1$. A k-dimensional skewer tree storing n boxes is called *α-balanced*, if for each node v of the skeleton tree, the subtree of v—which is a k-dimensional skewer tree—stores at most $\alpha^{d(v)} n$ boxes, and if the $(k-1)$-dimensional skewer tree that is stored with v is also α-balanced. Here, $d(v)$ is the depth of v in the skeleton tree.

If $\alpha = 1/2$, the skewer tree is called *perfectly balanced.*

The building algorithm for the skewer tree is similar to that in Section 2 and is left to the reader.

Lemma 8 *A k-dimensional skewer tree storing n boxes has size $O(n)$. A perfectly balanced skewer tree can be built in $O(n \log n)$ time.*

Proof: Let $S(n,k)$ denote the size of the skewer tree. Then, by Theorem 1, $S(n,2) = O(n)$. For $k > 2$, we have

$$S(n,k) = O(1) + S(n_0, k-1) + S(n_-, k) + S(n_+, k),$$

for some $0 < n_0 \leq n$ and $n_-, n_+ \geq 0$, such that $n_0 + n_- + n_+ = n$. Using this relation, it follows easily that $S(n,k) = O(n)$.

Similarly, the building time $T(n,k)$ satisfies $T(n,2) = O(n \log n)$, and for $k > 2$,

$$T(n,k) = O(n) + T(n_0, k-1) + T(n_-, k) + T(n_+, k),$$

for some $n_0 > 0$ and $0 \leq n_-, n_+ \leq n/2$, such that $n_0 + n_- + n_+ = n$. It follows that $T(n,k) = O(n \log n)$. ∎

Point location: Let $p = (p_1, \ldots, p_k)$ be a query point. If $k = 2$, we use the query algorithm of Section 2 that uses fractional cascading.

Assume that $k > 2$. We do a query with point $p' = (p_2, \ldots, p_k)$ in the $(k-1)$-dimensional skewer tree that is stored in the root. For each $(k-1)$-box found, we check whether p lies in the corresponding k-box. If it does, we report the k-box. Let $\sigma : x_1 = \beta_1$ be the hyperplane stored in the root. If $p_1 < \beta_1$, we do a query with p in the left subtree of the root, using the same algorithm recursively, unless this subtree is empty, in which case the query stops. Otherwise, if $p_1 \geq \beta_1$, we do a query with p in the right subtree of the root, unless it is empty.

Lemma 9 *A point location query in an α-balanced k-dimensional skewer tree can be solved in $O((\log n)^{k-1})$ time.*

Proof: First note that the query algorithm is correct. Let $Q(n,k)$ denote the query time. Then, by Theorem 1, $Q(n,2) = O(\log n)$. Let $k > 2$. Since the skeleton tree has height $O(\log n)$, and since each $(k-1)$-dimensional query gives at most $2^{k-1} = O(1)$ answers, we have $Q(n,k) = O(\log n) Q(n, k-1)$. This shows that $Q(n,k) = O((\log n)^{k-1})$. ∎

The constant factor in the query time can be exponential in k. This, however, only occurs if the query point lies on the boundary of many boxes. If the query point lies in the interior of a box, the constant will be small.

Insertion: Suppose we want to insert the k-box $B = [a_1 : b_1] \times \ldots \times [a_k : b_k]$. If $k = 2$, we use the algorithm of Section 2. So assume that $k > 2$. We search in the skeleton tree for the first node v, such that the hyperplane $\sigma_v : x_1 = \beta_1$ stored there, satisfies $a_1 < \beta_1 \leq b_1$. In each node that is visited during this walk, we increase the number of boxes that are stored in its subtree by one.

If node v exists, then we insert the $(k-1)$-box B'—which is obtained from B by deleting the first interval—into the $(k-1)$-dimensional skewer tree that is stored in v, using the same algorithm recursively.

If v does not exist, we end in a node w one of whose sons—the one to which the search wants to proceed—is missing. In this case, we give w a left or right subtree—depending on the position of B w.r.t. the hyperplane stored in w—which is a k-dimensional skewer tree for box B.

The problem of rebalancing is considered later.

Split operation: Let $1 \leq i \leq k$. Suppose we want to perform the operation i-*split*(s) on the k-box $B = [a_1 : b_1] \times \ldots \times [a_k : b_k]$. If $k = 2$ and $i = 1$, we use the vertical split algorithm of Section 2. If $k = i = 2$, we use the horizontal split algorithm of Section 2.

Assume that $k > 2$. We search in the skeleton tree for the first node v, such that the hyperplane $\sigma_v : x_1 = \beta_1$ stored there, satisfies $a_1 < \beta_1 \leq b_1$. If $i > 1$, we increase in each visited node the number of boxes stored in its subtree by one.

Next, if $i > 1$, we perform the operation i-*split*(s) on the $(k-1)$-box B' in the $(k-1)$-dimensional skewer tree that is stored in v, using the same algorithm recursively. Here, B' is obtained from B by deleting the first interval.

Otherwise, $i = 1$. Assume that $a_1 < \beta_1 \leq s$, i.e., the "left" part of the k-box to be split is intersected by the hyperplane σ_v in its interior or touches σ_v with its "right" boundary. (The case $s < \beta_1 \leq b_1$ is treated analogously.) We search for box B in the $(k-1)$-dimensional skewer tree T_v that is stored with v. (Note that B is stored twice in exactly one 2-dimensional skewer subtree of T_v, once as a *below*-value and once as an *above*-value.) Then we replace the two occurrences of B by the "left" part of the box. Finally, we use the above insertion algorithm to insert the "right" part of B into the data structure.

The problem of rebalancing is treated below.

Rebalancing: After the update has been carried out, we walk back to the root of the skeleton tree and we find the highest node w that does not satisfy the balance condition anymore. Then we rebuild the complete subtree rooted at the father of w as a perfectly balanced skewer tree.

Lemma 10 *In an α-balanced k-dimensional skewer tree, insert and split operations can be performed in $O((\log n)^2)$ amortized time.*

Proof: Let $I(n,k)$ denote the amortized insertion time. According to Theorem 1, $I(n,2) = O((\log n)^2)$. Let $k > 2$. Note that Lemma 6 also holds in the k-dimensional case. Consider the insert algorithm. It takes $O(\log n)$ time to search for node v. Each node on the search path contributes $O(\log n)$ amortized time to the rebalancing costs. (This can be proved in exactly the same way as in Lemma 7.) If node v exists, we spend at most $I(n,k-1)$ time in the skewer tree that is stored with v. Otherwise, if v does not exist, we add a skewer tree that stores the new box. In the same way as in the proof of Lemma 7, we can bound the amortized time for adding this skewer tree by $O(\log n)$. It follows that $I(n,k) = O((\log n)^2) + I(n,k-1)$. Using this relation, it follows easily that $I(n) = O((\log n)^2)$. Similarly, let $S(n,k)$ denote

the amortized time for a split operation. Then, $S(n,k) = O((\log n)^2) + I(n,k)$. Therefore, $S(n,k) = O((\log n)^2)$. This proves the lemma. ∎

This concludes the analysis of the k-dimensional skewer tree. The next theorem summarizes the results of this section.

Theorem 2 *For the problem of point location in a set of n non-overlapping k-dimensional boxes, there exists a data structure with a query time of $O((\log n)^{k-1})$, in which insert and split operations take $O((\log n)^2)$ amortized time, that can be built in $O(n \log n)$ time, and that has size $O(n)$.*

5 A fully dynamic data structure

Until now, we considered the case where the update operations are insertions and splits. In this section, we show how the structure can be adapted to allow deletion and merge operations to be performed as well. Since we need fully dynamic fractional cascading for this case, the time complexities increase by a factor of $O(\log \log n)$.

As before, we are given a set of non-overlapping k-boxes. Besides the operations point location, insertion and i-*split*(s), there are the following two operations:

Deletion: This operation deletes a k-box from the set.

Merge operation: The operation i-*merge* takes two k-boxes

$$[a_1 : b_1] \times \ldots \times [a_{i-1} : b_{i-1}] \times [a_i : s] \times [a_{i+1} : b_{i+1}] \times \ldots \times [a_k : b_k]$$

and

$$[a_1 : b_1] \times \ldots \times [a_{i-1} : b_{i-1}] \times [s : b_i] \times [a_{i+1} : b_{i+1}] \times \ldots \times [a_k : b_k],$$

and merges them together to obtain the new k-box $[a_1 : b_1] \times \ldots \times [a_k : b_k]$. This operation is defined for $1 \leq i \leq k$ and $a_i < s < b_i$.

The data structure for this fully dynamic problem is the α-balanced k-dimensional skewer tree, adapted as follows. Instead of SPLIT-FIND structures, we store data structures for the UNION-SPLIT-FIND problem with the augmented catalogues. In order to keep the skewer tree balanced, we require that each node v—except the root—stores at most $\alpha^{d(v)} n_0$ boxes in its subtree, where n_0 is the number of boxes that are present at the start of the algorithm. Hence, the value of n_0 is kept fixed during a sequence of updates. After $n_0/2$ updates, the complete data structure will be rebuilt. Then, the value of n_0 is set to the number of boxes that are present at that moment.

In this adapted skewer tree, the operations point location, insertion and i-*split*(s) are performed as before, except that we now use the UNION-SPLIT-FIND structure. The rebalancing algorithm is also the same as before, except that we rebalance as soon as a non-root node v contains more than $\alpha^{d(v)} n_0$ boxes in its subtree. The operations deletion and i-*merge* are given below.

Deletion: Suppose we want to delete the k-box $B = [a_1 : b_1] \times \ldots \times [a_k : b_k]$. If $k = 2$, we search in the skeleton tree for the first node v, such that the vertical line l stored with v intersects B in its interior or touches B on its right boundary. During this search, we decrease in each visited node the number of boxes that are stored in its subtree by one. Then we delete the y-coordinates of the top and bottom sides of B from the augmented catalogue $A(v)$, and we adapt some *below*- and *above*-values. Note that if this rectangle was the last one that intersects l, then node v itself is not deleted. In this case, the catalogue $C(v)$ will only contain the values $-\infty$ and $+\infty$. The augmented catalogue $A(v)$, however, will contain more elements.

Otherwise, if $k > 2$, we search in the skeleton tree for the first node v, such that the hyperplane $\sigma_v : x_1 = \beta_1$ stored there, satisfies $a_1 < \beta_1 \leq b_1$. During this search, we decrease in each visited node the number of boxes that are stored in its subtree by one. Then we delete the $(k-1)$-box B'—which is obtained from B by deleting the first interval—from the $(k-1)$-dimensional skewer tree that is stored in v, using the same algorithm recursively.

After the deletion, the data structure will still satisfy the balance condition, because n_0 is kept fixed. Therefore, no rebalancing operation is necessary.

Merge operation: Suppose we want to perform the operation *i-merge* on the two boxes $[a_1 : b_1] \times \ldots \times [a_i : s] \times \ldots \times [a_k : b_k]$ and $[a_1 : b_1] \times \ldots \times [s : b_i] \times \ldots \times [a_k : b_k]$.

First suppose that $k = i = 2$. Then we have to perform the inverse of a horizontal split: We search for the first node v in the skeleton tree whose vertical line intersects the two rectangles in their interiors or touches their right boundaries. During this search, we decrease in each visited node the number of boxes that are stored in its subtree by one. Then we delete the y-coordinate s from the augmented catalogue $A(v)$, and we adapt some *below*- and *above*-values.

If $k = 2$ and $i = 1$, we do the inverse of a vertical split: We search in the skeleton tree with both rectangles, until we reach the first node v whose vertical line l intersects one of the two rectangles in its interior or touches its right boundary. Assume that l intersects the left rectangle in its interior or touches its right boundary. (The symmetric case is handled analogously.) Then we replace the two occurrences of this left rectangle in the augmented catalogue $A(v)$—once as a *below*-value and once as an *above*-value—by the new rectangle. Next, we delete the right rectangle from the skewer tree using the above algorithm.

Now assume that $k > 2$. We search in the skeleton tree for the first node v, such that the hyperplane $\sigma_v : x_1 = \beta_1$ stored there, satisfies $a_1 < \beta_1 \leq b_1$. If $i > 1$, we decrease in each visited node the number of boxes that are stored in its subtree by one.

Then, if $i > 1$, we perform the operation *i-merge* on the two $(k-1)$-boxes—that are obtained by removing the first interval—in the $(k-1)$-dimensional skewer tree that is stored with v, using the same algorithm recursively.

Otherwise, $i = 1$. Assume that $a_1 < \beta_1 \leq s$. (The case $s < \beta_1 \leq b_1$ is treated analogously.) We search for the "left" box B in the $(k-1)$-dimensional skewer tree T_v that is stored in v. (This box is stored twice in a 2-dimensional skewer

subtree of T_v, once as a *below*-value and once as an *above*-value.) Then we replace the two occurrences of this "left" box by the merged box. Finally, we use the above deletion algorithm to delete the "right" box from the data structure.

Also in this case, no rebalancing operation needs to be performed.

Theorem 3 *For the problem of point location in a set of n non-overlapping k-dimensional boxes, there exists a data structure with $O((\log n)^{k-1} \log\log n)$ query time, in which insert, delete, split and merge operations take $O((\log n)^2 \log\log n)$ amortized time, that can be built in $O(n \log n \log\log n)$ time, and that has size $O(n)$.*

Proof: The heights of the various binary trees that are contained in the skewer tree are all bounded by $O(\log n_0)$. Since we rebuild the entire data structure after $n_0/2$ updates, the current number of boxes—n—satisfies $n_0/2 \leq n \leq 3n_0/2$. Therefore, the heights of all binary trees are bounded by $O(\log r)$.

The proofs of the complexity bounds are basically the same as those in Sections 2-4. The $O(\log\log n)$ terms come from the fact that we use a UNION-SPLIT-FIND structure.

Rebuilding of the data structure after $n_0/2$ updates adds only $O(\log n \log\log n)$ to the amortized update time. ∎

6 Maintaining the closest pair in a point set

In this section, we show how the skewer tree can be used to obtain an efficient data structure that maintains the closest pair in a point set if points are inserted. The method works for an arbitrary L_t-distance. Let $p = (p_1, \ldots, p_k)$ and $q = (q_1, \ldots, q_k)$ be two points in k-dimensional space. Then the L_t-distance $d_t(p, q)$ between p and q is defined by

$$d_t(p,q) := \left(\sum_{i=1}^{k} |p_i - q_i|^t \right)^{1/t},$$

if $1 \leq t < \infty$, and for $t = \infty$, it is defined by

$$d_\infty(p,q) := \max_{1 \leq i \leq k} |p_i - q_i|.$$

In the rest of this section, we fix t, and we measure all distances in the L_t-metric. We write $d(p,q)$ for $d_t(p,q)$.

Before we define the data structure, we prove some results that are needed in the analysis.

Lemma 11 *Let V be a set of points in k-dimensional space, and let the distance of a closest pair in V be at least equal to δ. Let l be a positive integer. Then any k-dimensional cube having sides of length $l\delta$ contains at most $(lk+l)^k$ points of V.*

Proof: Assume w.l.o.g. that the k-cube has the form $[0 : l\delta]^k$. Partition this cube into $(lk + l)^k$ subcubes

$$[i_1\delta/(k+1) : (i_1+1)\delta/(k+1)] \times \ldots \times [i_k\delta/(k+1) : (i_k+1)\delta/(k+1)],$$

where the i_j's are integers such that $0 \leq i_j \leq lk + l - 1$, for $1 \leq j \leq k$.

Assume that the cube contains at least $(lk + l)^k + 1$ points of V. Then one of the subcubes contains at least two points of V. These two points have a distance that is at most equal to the L_t-diameter of this subcube. This diameter, however, is at most $k \times \delta/(k+1) < \delta$. This contradicts the fact that the minimal distance of V is at least δ. ∎

Lemma 12 *Let V be a set of points in k-dimensional space, and let δ be the distance of a closest pair in V. Let B be a k-dimensional box that contains more than $(2k+2)^k$ points of V. For $i = 1, \ldots, k$, define m_i resp. M_i as the minimal resp. maximal i-th coordinate of any point in $V \cap B$. Then there is an i, such that $M_i - m_i > 2\delta$.*

Proof: Assume that $M_i - m_i \leq 2\delta$ for all $i = 1, \ldots, k$. Then, there is a k-cube B' having side lengths 2δ that contains all points of $V \cap B$. By the previous lemma, however, the cube B' contains at most $(2k+2)^k$ points of V. This is a contradiction. ∎

6.1 The closest pair data structure

Now we are ready to introduce the data structure that maintains the closest pair in a point set.

The data structure: Let V be a set of n points in k-dimensional space. If $\log n \leq 2(1 + 1/k)^{k/(k-1)}$, the data structure consists of the closest pair (P, Q) and the distance δ between these points.

Assume that n is such that $\log n > 2(1 + 1/k)^{k/(k-1)}$. Then, the data structure consists of the following.

1. There is a pair (P, Q) that maintains the current closest pair.

2. There is a variable δ that maintains the distance $d(P, Q)$.

3. At each moment, k-space is partitioned into non-overlapping k-boxes. Each k-box in this partition has sides of length at least δ. Each k-box of the partition contains at least one and at most $(2k)^k(\log n)^{k-1}$ points of V.

4. The k-boxes of the partition are stored in an α-balanced k-dimensional skewer tree of Section 4 that can handle insert and split operations. With each box, we store a list of those points in V that are contained in this box. (These points are stored in an arbitrary order. If a point is on the boundaries of several boxes, then it is stored in only one of these boxes.)

Remark: The choice of the constant $2(1+1/k)^{k/(k-1)}$ will become clear later. Note that

$$\frac{k}{k-1}\ln(1+1/k) = 1/k + O((1/k)^2).$$

Therefore,

$$(1+1/k)^{k/(k-1)} = \exp\left(1/k + O((1/k)^2)\right) = \quad + 1/k + O((1/k)^2).$$

Hence, if we forget about the quadratic term, the data structure is built if $\log n > 2 + 2/k$, or

$$n > 2^{2+2/k} = 4e^{(2\ln 2)/k} = 4 + \frac{8\ln 2}{k} + O((1/k)^2).$$

First, we show how this data structure can be built. In [8,13], it is shown how the closest pair and their distance can be computed in $O(n \log n)$ time, using $O(n)$ space. In Section 4, we have shown how a perfectly balanced skewer tree can be built in $O(n \log n)$ time. So it remains to be shown how the partition of k-space into k-boxes can be computed. We give a recursive algorithm that computes this partition.

The partitioning algorithm: Let V be a set of n points in k-space, where $k \geq 1$. Let δ be the distance between a closest pair in V. This variable δ is a global variable, i.e., in recursive calls it does not get a new value.

If $|V| = 1$, then the partition consists of one k-box, namely the entire space. So assume that $|V| > 1$.

Order the points of V with respect to their last coordinates. Let p be a smallest point in this ordered set. Let a_1 be the last coordinate of point p. Let $i \geq 1$, and assume that $a_1, \ldots, a_i$ are computed already.

If there is a point in V having a last coordinate lying in the half-open interval $(a_i : a_i + \delta]$, then we set $a_{i+1} := a_i + \delta$. Otherwise, we set a_{i+1} to the value of the last coordinate of a first point in the ordered set V that lies "to the right" of the hyperplane $x_k = a_i$. If there are no points to the right of this hyperplane, then a_{i+1} is not defined, and the construction of the a_j's stops.

This gives a sequence of intervals $(-\infty : a_1], (a_1 : a_2], \ldots, (a_l : \infty)$ for some l. Let $a_0 := -\infty$ and $a_{l+1} := \infty$. Partition V into subsets $V_0, \ldots, V_l$, where V_i contains those points of V that have their last coordinates in the interval $(a_i : a_{i+1}]$.

If $k = 1$, this is the desired partition of 1-space into 1-boxes, together with the corresponding partition of V.

Assume that $k \geq 2$. For $i = 0, 1, \ldots, l$, do the following. Use the same algorithm recursively to compute a partition of $(k-1)$-space into $(k-1)$-boxes for the set V_i, where we take only the first $k-1$ coordinates into account. (Note that in this recursive call, the value of δ remains equal to the minimal distance in the k-dimensional set V.) This gives a collection of $(k-1)$-boxes of the form

$$(b_1 : c_1] \times (b_2 : c_2] \times \ldots \times (b_{k-1} : c_{k-1}],$$

together with a corresponding partition of V_i. Replace each such box by the k-box

$$(b_1 : c_1] \times (b_2 : c_2] \times \ldots \times (b_{k-1} : c_{k-1}] \times (a_i : a_{i+1}].$$

The resulting boxes—for all i together—form the desired partition of k-space, together with the partition of V.

Lemma 13 *Let $k \geq 1$ and consider the k-boxes that are computed by the above algorithm. These boxes are non-overlapping and form a partition of k-space. Each box has sides of length at least δ. Each box contains at least one and at most $(k+1)^k$ points of V.*

Proof: It is easy to see that the boxes are non-overlapping, that they partition k-space, and that each of them contains at least one point of V. Let $B = (b_1 : c_1] \times \ldots \times (b_k : c_k]$ be a box that is computed by the algorithm. It is clear from the algorithm that $c_i \geq b_i + \delta$, $i = 1, \ldots, k$. Hence, B has sides of length at least δ. It remains to show that B contains at most $(k+1)^k$ points of V. We distinguish two cases.

If $c_i = b_i + \delta$, for all i, then it follows from Lemma 11 that there are at most $(k+1)^k$ points in B.

Otherwise, let I be the set of indices i for which $c_i > b_i + \delta$. If $i \in I$, then all points of V that are contained in B have equal i-th coordinates. Hence, if $|I| = k$, there is only one point in B, which is certainly at most $(k+1)^k$. Otherwise, if $|I| < k$, we consider the points of $V \cap B$ as points in $(k - |I|)$-space, by deleting the coordinates corresponding to the indices in I. These points are contained in a $(k - |I|)$-cube with side lengths δ. The distances between these points is at least equal to δ. Therefore, Lemma 11 implies that there are at most $(k - |I| + 1)^{k-|I|} \leq (k+1)^k$ of these points. Hence, also in this case, B contains at most $(k+1)^k$ points. ■

Remark: Since the partition of k-space is only computed if the number n of points is such that $\log n > 2(1 + 1/k)^{k/(k-1)}$, the number of points in a k-box is at most $(k+1)^k < (1/2)^{k-1} k^k (\log n)^{k-1} \leq (2k)^k (\log n)^{k-1}$.

Lemma 14 *The data structure has size $O(n)$ and can be built in $O(n \log n)$ time.*

Proof: First note that the given algorithm correctly builds the data structure. Since each box in the partition of k-space is non-empty, there are at most n such boxes. Therefore, the skewer tree has size $O(n)$. Since each point is stored in exactly one list—corresponding to one of the boxes that contain the point—all these lists together have size $O(n)$. This proves that the entire data structure has linear size.

The closest pair and the skewer tree can be computed in $O(n \log n)$ time. It remains to show that the given partitioning algorithm runs in $O(n \log n)$ time. Let $T(n, k)$ denote this running time. Then

$$\begin{aligned} T(n,1) &= O(n \log n), \\ T(n,k) &= O(n \log n) + \sum_{i=1}^{l} T(n_i, k-1), \text{ if } k \geq 2, \end{aligned}$$

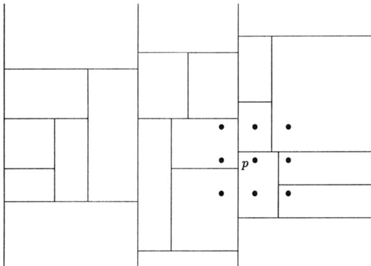


Figure 2: The 9 point location queries in the planar case.

for integers $n_i \geq 1$ such that $\sum_{i=1}^{l} n_i = n$. Using induction, it follows that $T(n,k) = O(n \log n)$, because k is a constant. ∎

6.2 Inserting a point

We now show how the closest pair is maintained after a point is inserted into the data structure. If the number n of points is such that $\log n < 2(1+1/k)^{k/(k-1)}$, then we just compute the new closest pair from scratch. The first time that $\log n \geq 2(1+1/k)^{k/(k-1)}$, we build the complete data structure. From now on, we assume that $\log n > 2(1+1/k)^{k/(k-1)}$.

The insert algorithm: Let $p = (p_1, \ldots, p_k)$ be the point to be inserted. Then we perform 3^k point location queries in the skewer tree, with query points $(p_1 + \epsilon_1, \ldots, p_k + \epsilon_k)$, for $\epsilon_1, \ldots, \epsilon_k \in \{-\delta, 0, \delta\}$. Each query gives at most 2^k answers. So all queries together give at most 6^k different k-boxes. For each of these boxes, we walk through its list of points. For each point q in these lists, if $d(p,q) < \delta$, we set $(P,Q) := (p,q)$ and $\delta := d(p,q)$. (See Figure 2.)

Next, we insert p into the list of a k-box it belongs to. If afterwards this list contains at least $(2k)^k(\log n)^{k-1}$ points, we perform a split operation on its k-box, as described below.

Split operation: Suppose we want to split a box $B = [a_1 : b_1] \times \ldots \times [a_k : b_k]$ of the partition. Let V' be the set of points that are stored in the list of B.

For $i = 1, \ldots, k$, we compute the values m_i and M_i, which are the minimal resp. maximal i-th coordinate of any point of V'. If $M_i - m_i \leq 2\delta$, for all $i = 1, \ldots, k$, the algorithm stops.

Otherwise, we take an index i for which $M_i - m_i > 2\delta$. We compute the median c_i of the i-th coordinates of the points of V'. There are three possible cases.

1. If $a_i + \delta \leq c_i \leq b_i - \delta$, we perform the operation *i-split*(c_i) on box B in the skewer tree. We also split the list of box B in two lists corresponding to the two new boxes. Then, the algorithm is finished.

2. If $a_i \leq c_i < a_i + \delta$, we perform the operation *i-split*$(a_i + \delta)$ on box B in the skewer tree. This gives two new k-boxes B' and B'', obtained from B by replacing the i-th interval by $[a_i : a_i + \delta]$ resp. $[a_i + \delta : b_i]$. We split the list of box B in two lists corresponding to these two new boxes. Then, we split the box B' using the same algorithm recursively.

3. If $b_i - \delta < c_i \leq b_i$, we perform the operation *i-split*$(b_i - \delta)$ on box B in the skewer tree. This gives two new k-boxes B' and B'', obtained from B by replacing the i-th interval by $[a_i : b_i - \delta]$ resp. $[b_i - \delta : b_i]$. We split the list of box B in two lists corresponding to these two new boxes. Then, we split the box B'' using the same algorithm recursively.

Remark: The split operation is called if a box contains at least $(2k)^k(\log n)^{k-1}$ points. We assumed that $\log n > 2(1 + 1/k)^{k/(k-1)}$. Therefore, $(2k)^k(\log n)^{k-1} > (2k+2)^k$. Then, Lemma 12 guarantees that there is an index i such that $M_i - m_i > 2\delta$ at the start of the split operation.

If $a_i = -\infty$, then $a_i + \delta = -\infty$. Similarly if $b_i = \infty$. Furthermore, if case 2 resp. 3 applies, then a_i resp. b_i is finite. Hence, the *i-split* operations in the above algorithm are well defined.

Lemma 15 *Let $m \geq 2(2k+2)^k$ be an integer. Let B be a k-box in the partition of k-space whose list contains at most m points. Let δ be the minimal distance of the entire set V at the moment the split algorithm is carried out on B. After this algorithm, the sides of all boxes that have been created have length at least δ, and each such box contains at least one and at most $\lceil m/2 \rceil$ points of V.*

Proof: Consider the box $B = [a_1 : b_1] \times \ldots \times [a_k : b_k]$. Note that by Lemma 13, $b_j - a_j \geq \delta$, for all j. Let I_B be the number of indices j for which $M_j - m_j > 2\delta$. The proof is by induction on the value of I_B.

If $I_B = 0$, then it follows from Lemma 11 that B contains at most $(2k+2)^k$ points of V. In this case, the lemma follows because $(2k+2)^k \leq \lceil m/2 \rceil$. (We saw already that $I_B > 0$ at the start of the algorithm. We start the induction at $I_B = 0$, however, to simplify the proof.)

Let $I_B > 0$, and suppose that the lemma is proved for smaller values of I_B. The algorithm takes an index i for which $M_i - m_i > 2\delta$.

If case 1 of the algorithm applies, then it is clear that the two new boxes have sides of length at least δ, that they each contain at least one and at most $\lceil m/2 \rceil$ points. Hence, in this case, the proof is completed.

Otherwise, assume that case 2 applies. It is clear that box B'' has sides of length at least δ and that it contains at most $\lceil m/2 \rceil$ points. Since $M_i - m_i > 2\delta$, there is a point in B's list whose i-th coordinate is greater than $a_i + \delta$. Therefore, box B'' contains at least one point.

Consider box B'. This box is non-empty and has sides of length at least δ. Let $I_{B'}$ be the number of indices j for which $M'_j - m'_j > 2\delta$, where m'_j resp. M'_j is the minimal resp. maximal j-th coordinate of any point in the list of B'. Note that $I_{B'} = I_B - 1$. The algorithm recursively splits box B'. Note that box B' contains at most m points. Therefore, by the induction hypothesis, the boxes into which B' is split all have sides of length at least δ, contain at least one and at most $\lceil m/2 \rceil$ points. Hence, if case 2 applies, the proof is completed.

If case 3 applies, the proof is similar as for case 2. ∎

Lemma 16 *For a box B whose list contains m points, the split operation takes $O(m + (\log n)^2)$ amortized time.*

Proof: First note that the number of *i-split* operations that are performed on the box is at most $I_B \leq k$. By Theorem 2, each *i-split* operation needs $O((\log n)^2)$ amortized time to update the skewer tree. With each *i-split* operation, we compute a median and split a list of size at most m in two sublists. This can be done in $O(m)$ time. This proves the lemma, because k is a constant. ∎

Lemma 17 *The insert algorithm correctly maintains the closest pair data structure.*

Proof: Let δ be the minimal distance just before the insertion of point p. If this minimal distance changes, there must be a point inside the L_t-ball of radius δ centered at p. This ball is contained in the box $[p_1 - \delta : p_1 + \delta] \times \ldots \times [p_k - \delta : p_k + \delta]$. Therefore, it suffices to compare p with all points of the current set V that are in this box. Let

$$W := V \cap ([p_1 - \delta : p_1 + \delta] \times \ldots \times [p_k - \delta : p_k + \delta])$$

be the set of these points, and let W' be the set of points that are contained in the lists corresponding to the at most 6^k boxes that result from the 3^k point location queries. The algorithm compares p with all points in W'. Hence, if we show that $W \subseteq W'$, then it is clear that the algorithm correctly maintains the closest pair.

If $W = \emptyset$, then certainly $W \subseteq W'$. So assume that $W \neq \emptyset$. Let $q = (q_1, \ldots, q_k)$ be a point in W. Assume w.l.o.g. that $q_i \geq p_i$ for $i = 1, \ldots, k$. Then $p_i \leq q_i \leq p_i + \delta$ for $i = 1, \ldots, k$. Let B be the k-box in the partition of k-space whose list contains q. Assume that $q \notin W'$. Then B does not contain any of the 2^k points $(p_1 + \alpha_1, \ldots, p_k + \alpha_k)$, where $\alpha_1, \ldots, \alpha_k \in \{0, \delta\}$. These 2^k points are the corners of the k-box

$$B' := [p_1 : p_1 + \delta] \times \ldots \times [p_k : p_k + \delta],$$

having sides of length δ. (Note that in general B' is not part of the partition of k-space.) Since $q \in B'$, and since B does not contain any of the corner points of B', it follows that the box B must have at least one side of length strictly less than δ.

This contradicts the definition of our data structure. Hence, $q \in W'$ and, therefore, $W \subseteq W'$. This proves that the insert algorithm correctly maintains the closest pair.

Consider a box of the partition that is not split during the insertion. Since the value of δ can only decrease, the side lengths of this box remain at least equal to δ. Clearly, if the box contains at least one point before the insertion, so it does afterwards. Also, the box still contains at most $(2k)^k(\log n)^{k-1}$ points, because the value of n only increases.

If a box is split, then Lemma 15 guarantees that the new boxes have sides of length at least δ, that they contain at least one and at most

$$\left\lceil (1/2)(2k)^k(\log n)^{k-1}\right\rceil \leq (2k)^k(\log n)^{k-1})$$

points. (Note that $\lceil (2k)^k(\log n)^{k-1}\rceil \geq 2(2k+2)^k$. Therefore, Lemma 15 can be applied.) Finally, it is clear that the boxes that are not split together with the new boxes are non-overlapping and partition k-space. ∎

Lemma 18 *The amortized time to insert a point into the closest pair data structure is bounded by $O((\log n)^{k-1})$.*

Proof: By Theorem 2, it takes $O((\log n)^{k-1})$ time to perform the 3^k point location queries in the skewer tree. For each of the at most 6^k found k-boxes, we walk through its list of points and compare these with the new point. Since each such list contains $O((\log n)^{k-1})$ points, this step of the insert algorithm takes $O((\log n)^{k-1})$ time. The new point can be inserted in $O(1)$ time into the appropriate list.

If a k-box is split, it contains $\lceil (2k)^k(\log n)^{k-1}\rceil$ points. By Lemma 16, this operation takes $O((\log n)^{k-1} + (\log n)^2)$ amortized time. It follows from Lemma 15 that each of the boxes that are created during a split operation contains at most $\lceil (1/2)(2k)^k(\log n)^{k-1}\rceil$ points. Therefore, at least $\lfloor (1/2)(2k)^k(\log n)^{k-1}\rfloor$ points must be inserted into such a box before it is split again. Charging the $O((\log n)^{k-1} + (\log n)^2)$ time for the split operation to these $\lfloor (1/2)(2k)^k(\log n)^{k-1}\rfloor$ insertions, shows that a split operation adds an amount of time to the overall amortized insertion time that is bounded by $O(1 + (\log n)^{3-k})$. Since $k \geq 2$, this amount is bounded by $O((\log n)^{k-1})$. ∎

This concludes the description of the data structure, the insert algorithm and its analysis. We summarize the results of this section in the following theorem.

Theorem 4 *There exists a data structure that maintains the closest pair in a set of n points in k-dimensional space at a cost of $O((\log n)^{k-1})$ amortized time per insertion. The data structure has size $O(n)$ and can be built in $O(n \log n)$ time.*

This data structure can be used to give an on-line algorithm for computing the closest pair in a point set. Then, the points arrive one after another, and the total number of points is not known in advance. The next point becomes available as soon as the current point has been inserted.

Corollary 1 *The closest pair in a set of n points in k-dimensional space can be be computed on-line in $O(n(\log n)^{k-1})$ time.*

It is well-known that it takes $\Omega(n \log n)$ time to compute the closest pair. (See e.g. [8].) Therefore, in the planar case, Theorem 4 and Corollary 1 give optimal results.

7 Concluding remarks

We have given a dynamic data structure for the k-dimensional rectangular point location problem. If the only dynamic operations are insertions and splits, the data structure has a query time of $O((\log n)^{k-1})$ and an amortized update time of $O((\log n)^2)$. If also deletions and merges have to be supported, these two time bounds increase by a factor of $O(\log \log n)$. The size of the data structure is $O(n)$.

It is an open problem whether a logarithmic factor can be saved in the update times. If during an update no rebalancing is necessary, the update times are $O(\log n)$—in case only insertions and splits have to be supported. Rebalancing is responsible for the amortized $O((\log n)^2)$ time bound on the update times.

One possibility to remove a factor of $O(\log n)$ is to investigate whether a logarithmic factor can be saved in Lemma 5. We rebuild a subtree as a perfectly balanced skewer tree. Since we have the old subtree available—although it is out of balance—it might be possible to rebuild it in $O(\alpha^{d(v)} n)$ time. If this is possible, then the update times in Theorems 1 and 2 resp. Theorem 3 become $O(\log n)$ resp. $O(\log n \log \log n)$.

Another possibility to save a logarithmic factor is to define another balance condition. For example, if it is possible to maintain the skewer tree by rotations—as for segment trees that are based on BB[α]-trees, see [6]—then maybe the update times can be improved.

In the second part of the paper, we have shown how the skewer tree can be used to maintain the closest pair in a point set in $O((\log n)^{k-1})$ amortized time per insertion. In the planar case, this result is optimal. It is an open problem whether this amortized time bound can be made worst-case. Note that we use a variation of the partial rebuilding to rebalance the skewer tree. It is not known at present whether the general partial rebuilding technique can be made worst-case.

Finally, it would be interesting to improve the insertion time for the closest pair problem in dimensions greater than two.

References

[1] M. Blum, R.W. Floyd, V. Pratt, R.L. Rivest and R.E. Tarjan. *Time bounds for selection.* J. Comput. System Sci. **7** (1973), pp. 448-461.

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

[3] D. Dobkin and S. Suri. *Dynamically computing the maxima of decomposable functions, with applications.* Proc. 30th Annual IEEE Symp. on Foundations of Computer Science, 1989, pp. 488-493.

[4] H. Edelsbrunner. *Algorithms in Combinatorial Geometry.* Springer-Verlag, Berlin, 1987.

[5] H. Edelsbrunner, G. Haring and D. Hilbert. *Rectangular point location in d dimensions with applications.* The Computer Journal **29** (1986), pp. 76-82.

[6] K. Mehlhorn and S. Näher. *Dynamic fractional cascading.* Algorithmica **5** (1990), pp. 215-241.

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

[8] F.P. Preparata and M.I. Shamos. *Computational Geometry, an Introduction.* Springer-Verlag, New York, 1985.

[9] M. Smid. *Maintaining the minimal distance of a point set in less than linear time.* Proc. 2nd Canadian Conf. on Computational Geometry, 1990, pp. 1-4.

[10] M. Smid. *Algorithms for semi-online updates on decomposable problems.* Proc. 2nd Canadian Conf. on Computational Geometry, 1990, pp. 347-350.

[11] M. Smid. *Maintaining the minimal distance of a point set in polylogarithmic time.* Proc. 2nd Annual ACM-SIAM Symp. on Discrete Algorithms, 1991, pp. 1-6.

[12] K.J. Supowit. *New techniques for some dynamic closest-point and farthest-point problems.* Proc. 1st Annual ACM-SIAM Symp. on Discrete Algorithms, 1990, pp. 84-90.

[13] P.M. Vaidya. *An $O(n \log n)$ algorithm for the all-nearest-neighbors problem.* Discrete Comput. Geom. **4** (1989), pp. 101-115.