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

Michiel Smid

Max-Planck-Institut für Informatik

D-6600 Saarbrücken, Germany

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.

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

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 [7] and Edelsbrunner [3].

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 short. Edelsbrunner, Haring and Hilbert [4] considered the static version 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 [5]) 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 [7, 12].) For the dynamic case, there are data structures by Dobkin and Suri [2] and Smid [9] that can handle semi-online updates in $O((\log n)^2)$ time using linear space, for the planar case. Supowit [11] gives a structure that performs deletions in $O((\log n)^k)$ amortized time using $O(n(\log n)^{k-1})$ space. Finally, Smid [8, 10] 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 $O((\log n)^k \log\log n)$ 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 as a data structure for solving 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. Then, in Section 4, 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 5, 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 6 with some concluding remarks and open problems.

2 Rectangular point location

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 $b_i \in \mathcal{R} \cup \{\infty\}$, $i = 1, \ldots, k$. Hence, a k-box may be infinite.

In the k-dimensional rectangular point location problem, we are given a set of n non-overlapping k-boxes. The boxes do not necessarily partition k-space. We want to store these boxes in a data structure such that the following three operations can be supported.

Point location: Given a query point p in k-space, find the boxes—if any—that contain p. If p lies on the boundary of a box, there may be several boxes 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 $i\text{-}split(s)$ 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 bounded as for unbounded boxes, and for $1 \leq i \leq k$ and $a_i < s < b_i$.

Edelsbrunner, Haring and Hilbert [4] introduced the skewer tree for the static version of this problem.

The k-dimensional skewer tree: Let V be a set of non-overlapping k-boxes. The skewer tree is recursively defined as follows.

Suppose $k = 1$. Then the skewer tree is a balanced binary search tree storing all endpoints of the intervals in V and the values $-\infty$ and ∞. With each value s, we store the interval $below(s)$ resp. $above(s)$, which is the interval in V that has its right resp. left endpoint at s. If $below(s)$ or $above(s)$ does not exist, then the value of this variable is nil.

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 σ 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. The root 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 1-dimensional skewer trees as substructures. In these 1-dimensional structures, the *below*- and *above*-values are k-boxes instead of intervals.

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 $1/2 \leq \alpha < 1$. Let $k = 2$. For each node v of the skeleton tree of the 2-dimensional skewer tree, let n_v denote the total number of rectangles that are stored in the subtree of v (including v itself), and let $d(v)$ denote the depth of v in the skeleton tree. 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 $k > 2$, 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. Again, $d(v)$ is the depth of v in the skeleton tree.

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

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: Let $p = (p_1, \ldots, p_k)$ be a query point. If $k = 1$, we do a simple binary search.

Assume that $k > 1$. We do a query with point $p' = (p_2, \ldots, p_k)$ in the $(k-1)$-dimensional skewer tree that is stored with 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. If p lies in the interior of a reported box, then the search procedure is finished. Otherwise, we proceed recursively: 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, 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.

By Lemma 1, the skewer tree has height $O(\log n)$. Therefore, it follows that the query time is bounded by $O((\log n)^k)$.

Note that in the planar case, we do a logarithmic number of binary searches with the same y-coordinate p_2. Therefore, using dynamic fractional cascading, the query time can be improved to $O((\log n)^{k-1})$. (See Chazelle and Guibas [1] and Mehlhorn and Näher [5] for details about fractional cascading.) Note that since we only allow insertions and splits of boxes, we do not need the full power of dynamic fractional cascading. Therefore, there is no $\log\log n$-factor.

Lemma 2 *An α-balanced k-dimensional skewer tree, equipped with fractional cascading, has size $O(n)$, can be built in $O(n \log n)$ time and has a query time of $O((\log n)^{k-1})$.*

Proof: The query time follows from the above discussion. In [4], the bounds on the size and the building time are proved for a skewer tree that is not equipped with fractional cascading. In [5], it is shown that dynamic fractional cascading, where only insertions are allowed, increases the complexity by at most a constant factor. Note that a split operation can be implemented as an insertion. ■

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

Insertion: Suppose we want to insert the k-box $B = [a_1 : b_1] \times \ldots \times [a_k : b_k]$. If $k = 1$, we insert the two endpoints of the interval B. So assume that $k > 1$. 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.

Note that the fractional cascading information also has to be updated. See [5] for details. 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 $k = 1$, we insert the element s and update all relevant information.

Assume that $k > 1$. 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 $s < \beta_1 \leq b_1$ can be 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 1-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.

Again, the fractional cascading information also has to be updated. See [5] for details. The problem of rebalancing is treated 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.[6]):

After the update has been carried out, 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. (Note that the root of the skeleton tree never gets out of balance. For the balance condition, we only count "real" boxes, we do not count boxes that are copies caused by fractional

cascading.) Then we rebuild the complete subtree rooted at the *father of* w as a perfectly balanced skewer tree. (The dynamic fractional cascading information has to be updated accordingly. See [5] for details.)

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 boxes in its left resp. right subtree lie completely to the left of σ resp. lie completely to the right of σ or touch σ with their left boundaries, where σ is the hyperplane that is stored in v. Moreover, 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 will be shown in Lemma 5 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 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 $O(n)$.*

3 Analysis of the insert and split operations

In this section, we complete the proof of Theorem 1. The proof of the following lemma is similar to that of Lemma 13 in [5] and, therefore, it is omitted. (The proof can be obtained from the author.) For the notion of *augmented catalogue*, see [5].

Lemma 3 *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 all its subtrees, has size $O(\alpha^{d(u)} n)$. Here, n is the number of rectangles that are stored in the entire skewer tree.*

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 4 *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: Let m be the number of boxes that are stored in the subtree rooted at v. Then, using a similar analysis as in [5], it can be shown that the time for the rebuilding operation is bounded by $O(m \log m + \alpha^{d(v)}\, n + m) = O(\alpha^{d(v)}\, n \log n)$, 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 5 *The rebalancing algorithm correctly rebuilds an α-balanced skewer tree. Moreover, 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 boxes 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 boxes 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 $(1/2)^{d'(u)}\, m$ boxes 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 boxes 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 boxes at this moment. Note that $n' \geq n$. At this moment, the subtree rooted at w stores more than $\alpha^{d(w)} n' \geq \alpha^{d(w)} n$ boxes. We saw above that immediately after the previous rebalancing operation caused by w, its subtree contained at most $(1/(2\alpha))\alpha^{d(w)} n$ boxes. 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 6 *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. Clearly, $I(n, 1) = O(\log n)$, even in the worst case. Let $k \geq 2$. It takes $O(\log n)$ time to search for the node v. If node v exists, we spend at most $I(n, k-1)$ time in the skewer tree that is stored with v.

If node v does not exist, we add a skewer tree for one k-box. This takes time proportional to the size of the subtree of the father of the new subtree. (We have to update the fractional cascading information.) Therefore, the time for this adding can be very large. Below, we analyze the amortized time for this operation.

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 5, 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 4, this rebuilding takes $O(\alpha^{d(v)}\, n \log n)$ time. During this sequence, several subtrees may have been added to the subtree of w. (This happens if the node v of the insertion algorithm does not exists.) The time to add these subtrees 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 insertion, 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 insertion time. This proves that the total amortized insertion time satisfies $I(n,k) = O((\log n)^2) + I(n, k-1)$. Using this relation, it follows that $I(n,k) = O((\log n)^2)$.

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

4 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 delete 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, insert 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. We equip the structure with fully dynamic fractional cascading, as in [5]. 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, insert and *i-split*(s) are performed as before. 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 delete and *i-merge* are inverses of insert and *i-split* and are, therefore, left to the reader. Note that after a delete or merge operation, the data structure still satisfies the balance condition, because n_0 is kept fixed. Therefore, no rebalancing operation is necessary.

Theorem 2 *For the problem of point location in 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 $O(\log n)$.

The proofs of the complexity bounds are basically the same as those in Sections 2 and 3. The $O(\log\log n)$ terms come from the fact that we use fully dynamic fractional cascading.

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

5 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 $d_t(p, q) := (\sum_{i=1}^{k} |p_i - q_i|^t)^{1/t}$, 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 7 *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 8 *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: This follows from the previous lemma. ■

5.1 The closest pair 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. A pair (P, Q) that maintains the closest pair, and a variable $\delta = d(P, Q)$.

2. 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.

3. The k-boxes of the partition are stored in an α-balanced k-dimensional skewer tree of Section 2 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 $(1+1/k)^{k/(k-1)} = \exp(1/k + O((1/k)^2)) = 1 + 1/k + O((1/k)^2)$.

First, we show how this data structure can be built. In [7, 12], it is shown how the closest pair can be computed in $O(n \log n)$ time, using $O(n)$ space. In Section 2, we have shown how a perfectly balanced skewer tree can be built in $O(n \log n)$ time. So it remains to show 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 k-th coordinates. Let p be a smallest point in this ordered set. Let a_1 be the k-th 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 k-th 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 k-th 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 k-th 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 9 *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: The proof follows immediately from the algorithm and Lemma 7. ■

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

Proof: It is clear that the 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 $T(n,1) = O(n \log n)$, and if $k \geq 2$, $T(n,k) = O(n \log n) + \sum_{i=1}^{l} T(n_i, k-1)$, 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. ■

5.2 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 1.)

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.

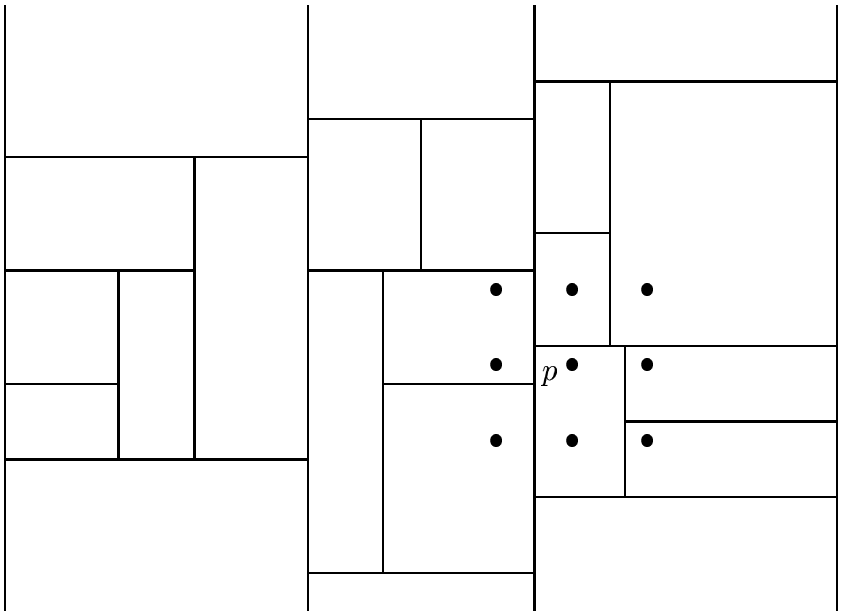

Figure 1: The 9 point location queries in the planar case.

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 8 guarantees that there is an index i such that $M_i - m_i > 2\delta$ at the start of the split operation.

Lemma 11 *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 the definition of the data structure, $b_j - a_j \geq \delta$, for all j. Let I_B be the number of indices j for which $M_j - m_j > 2\delta$. The lemma follows by induction on the value of I_B.

We saw already that $I_B > 0$ at the start of the algorithm. Starting the induction at $I_B = 0$, however, simplifies the proof. If $I_B = 0$, then it follows from Lemma 7 that B contains at most $(2k+2)^k$ points of V. In this case, the lemma follows because $(2k+2)^k \leq \lceil m/2 \rceil$.

The induction step follows immediately from the given split algorithm and from the induction hypothesis. ■

Lemma 12 *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 1, 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 13 *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. Clearly, $W \subseteq W'$ holds, because each k-box in the partition of k-space has sides of length at least δ.

The rest of the data structure is also correctly maintained. This follows since δ can only decrease, n only increases, and using Lemma 11. ■

Lemma 14 *The amortized time to insert a point into the closest pair data structure is bounded by* $O((\log n)^{k-1})$.

Proof: By Theorem 1, 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 12, this operation takes $O((\log n)^{k-1} + (\log n)^2)$ amortized time. It follows from Lemma 11 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. Hence, a split operation adds $O(1 + (\log n)^{3-k})$ to the overall amortized insertion time. 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 3 *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.

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

6 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 $O(n)$.

Maybe 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 save a factor of $O(\log n)$ is to improve Lemma 4. 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 amortized update times in Theorems 1 resp. Theorem 2 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 [5]—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 technique 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] B. Chazelle and L.J. Guibas. *Fractional cascading I: A data structuring technique.* Algorithmica **1** (1986), pp. 133-162.

[2] 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.

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

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

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

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

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

[8] M. Smid. *Maintaining the minimal distance of a point set in less than linear time.* Algorithms Review **2** (1991), pp. 33-44.

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

[10] M. Smid. *Maintaining the minimal distance of a point set in polylogarithmic time.* To appear in Discrete Comput. Geom.

[11] 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.

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