MAX-PLANCK-INSTITUT
FÜR
INFORMATIK

An $O(n \log n \log\log n)$ algorithm for the on-line closest pair problem

Christian Schwarz Michiel Smid

MPI–I–91–107 July 1991

Im Stadtwald
W 6600 Saarbrücken
Germany

An $O(n \log n \log \log n)$ algorithm for the on-line closest pair problem*

Christian Schwarz Michiel Smid

Max-Planck-Institut für Informatik

D-6600 Saarbrücken, Germany

July 30, 1991

Abstract

Let V be a set of n points in k-dimensional space. It is shown how the closest pair in V can be maintained under insertions in $O(\log n \log \log n)$ amortized time, using $O(n)$ space. Distances are measured in the L_t-metric, where $1 \leq t \leq \infty$. This gives an $O(n \log n \log \log n)$ time on-line algorithm for computing the closest pair. The algorithm is based on Bentley's logarithmic method for decomposable searching problems. It uses a non-trivial extension of fractional cascading to k-dimensional space. It is also shown how to extend the method to maintain the closest pair during semi-online updates. Then, the update time becomes $O((\log n)^2)$, even in the worst case.

1 Introduction

The closest pair problem is one of the problems in computational geometry that has received much attention. In this problem, we are given a set of n points in k-dimensional space and we want to compute a closest pair or its distance. Distances are measured in the L_t-metric, where $1 \leq t \leq \infty$.

In the L_t-metric, the distance $d_t(p, q)$ between two points $p = (p_1, \ldots, p_k)$ and $q = (q_1, \ldots, q_k)$ in k-dimensional space 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

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

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

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

In the *off-line* version of the closest pair problem, the complete set of points is known at the start of the algorithm. In this case, several optimal $O(n \log n)$ algorithms are known. See Bentley and Shamos [2], Shamos and Hoey [10], Preparata and Shamos [8], Vaidya [17].

In this paper, we consider the *on-line* version: The points become available one after another. A new point arrives as soon as the insertion of the previous point has been completed. At the start of the algorithm, the final size of the point set is not known.

To solve this on-line problem, we need a data structure that maintains the closest pair in the current set V. The only operation that has to be supported is the insertion of a point. During an insertion, we have to update the closest pair.

In Smid [13], a data structure is given that maintains the minimal distance in amortized $O((\log n)^{k-1})$ time, using $O(n)$ space. This leads to an $O(n(\log n)^{k-1})$ time algorithm for the on-line closest pair problem. In the planar case, this result is optimal. In Smid [15], an algorithm is sketched that improves the running time — for $k \geq 3$ — to $O(n(\log n)^2 / \log\log n)$.

In this paper, we give a data structure that uses $O(n)$ space and that maintains the closest pair in $O(\log n \log\log n)$ amortized time. As a result, we can compute the closest pair in a point set on-line in $O(n \log n \log\log n)$ time.

Note that recently several papers have appeared that are concerned with the dynamic closest pair problem. Supowit [16] considers the case where there are only deletions. He obtains an amortized deletion time of $O((\log n)^k)$ using $O(n(\log n)^{k-1})$ space. If both insertions and deletions have to be supported, the best linear size data structure is obtained by combining results of Smid [12], Dickerson and Drysdale [4] and Salowe [9]. The resulting structure has a worst-case update time of $O(\sqrt{n} \log n)$. In Smid [14], an amortized update time of $O((\log n)^k \log\log n)$ is obtained using $O(n(\log n)^k)$ space.

The framework of the algorithm for maintaining the closest pair under insertion of points is as follows: Let V be the current set of points, let (P,Q) be the current closest pair and let $\delta = d(P,Q)$. The algorithm for inserting a point p does the following:

1. It finds out if there is a point $q \in V$, $q \neq p$, such that $d(p,q) < \delta$. If there is no such point, the insertion of p does not change the closest pair. Otherwise, it finds a point q in V that is closest to p, and sets $(P,Q) := (p,q)$ and $\delta := d(p,q)$.

2. It inserts p into the data structure.

In step 1, the closest pair is updated. It is clear that if the closest pair changes, then the new point is part of the new closest pair.

The main problem is how to implement step 1. We have to find a nearest neighbor q of the new point p, but only if $d(p,q) < \delta$. Hence, we need a data structure for this restricted post-office problem. For the static version of this query problem, it is easy to give a data structure solving it. This static structure is presented in Section 2. Then,

in Section 3, we apply the logarithmic method for decomposable searching problems (see Bentley [1]). The result is a data structure of linear size maintaining the closest pair in $O((\log n)^2)$ amortized time per insertion.

In [7], Mehlhorn gives a class of algorithms for which he proves that Bentley's logarithmic method is optimal. In the second part of the paper, however, we show how fractional cascading (see Chazelle and Guibas [3] and Edelsbrunner, Guibas and Stolfi [6]) can be applied in a non-trivial way to improve the time bound. (Clearly, the resulting algorithm does not belong to Mehlhorn's class.)

In our case, the objects stored are hypercubes belonging to different grids. We have to implement the following queries: Given a point, do a point location, i.e., find the hypercube containing it, in all grids. Standard fractional cascading cannot be applied here, because the hypercubes in the various grids have different sizes.

We show how to extend fractional cascading to this higher-dimensional searching problem. It turns out that we have to define an ordering relation for the hypercubes in one grid that depends on the orderings in other grids. This ordering is defined and analyzed in Section 4. Unfortunately, it takes $O(\log \log n)$ time to compare two hypercubes in such an ordering. In Section 5, we present the improved data structure for maintaining the closest pair. In Section 6, we use the new orderings for point location in all grids, employing a variant of fractional cascading similar to [6]. In that section, we also consider step 2 of the insertion algorithm. It is shown that we can use old information to speed up this step (compared to Section 3). At the end of Section 6, we have proved the main result of this paper.

In Section 7, we consider the case where the updates are *semi-online*. A sequence of updates is called semi-online, if the insertions are online, but with each inserted point p, we get an integer d saying that p will be deleted d updates from the moment of insertion. This type of updates was introduced by Dobkin and Suri [5]. They show that in the planar case, the closest pair in a point set can be maintained in $O((\log n)^2)$ amortized time when semi-online updates are performed. (Their algorithm is a generalization of the logarithmic method.) This method was made worst-case in Smid [11].

We show that the method of [5, 11] can be applied to the static data structure of Section 2. The result is a data structure of linear size maintaining the closest pair in $O((\log n)^2)$ time per semi-online update, even in the worst case.
In Section 8, we finish the paper with some concluding remarks.

Throughout this paper, we use the following notation. If V is a set of points in k-space, then $\delta(V)$ denotes the minimal distance in V, i.e.,

$$\delta(V) := \min\{d(p, q) : p, q \in V, p \neq q\}.$$

2 The restricted post-office problem

Definition 1 Let $c \in \mathbb{N}$ be fixed. In the *restricted post-office problem*, we are given a set V of n points in k-space and $\sigma \in \mathbb{R}$ such that $0 < \sigma \leq c\,\delta(V)$. We have to store V in a (static) data structure such that restricted post-office queries can be solved. In such a query, we get a point p in k-space and $\delta \in \mathbb{R}$ such that $0 < \delta \leq \sigma$, and we have

to compute the value of

$$f(p, V, \sigma, \delta) := \min\{d(p, q) : q \in V, d(p, q) \leq \delta\}.$$

(We define $\min \emptyset := \infty$.) In case $f(p, V, \sigma, \delta) < \infty$, we also want a nearest neighbor of p in V. ∎

Hence, in a restricted post-office query we want a nearest neighbor q of p, if $d(p, q) \leq \delta$. We note here that it is not necessary to know the value of $\delta(V)$. In the applications to be presented later, the algorithms will guarantee that the value of σ is at most equal to $c\,\delta(V)$, for $c = 1$ resp. $c = 2$. Before we give the data structure for such queries, we define the notion of a σ-grid.

Definition 2 Let $\sigma \in \mathbb{R}_{>0}$. The subdivision of k-space induced by the set of $(k-1)$-dimensional hyperplanes $\bigcup_{1 \leq i \leq k} \bigcup_{j \in \mathbb{Z}} h_{i,j} : x_i = j \cdot \sigma$ is called *σ-grid.*

The σ-grid consists of hypercubes of the form $[i_1\sigma : (i_1+1)\sigma] \times \ldots \times [i_k\sigma : (i_k+1)\sigma]$, for integers $i_1, \ldots, i_k$. These hypercubes are called *σ-boxes.*

If $p = (p_1, \ldots, p_k)$ is a point in k-space and $[i_1\sigma : (i_1+1)\sigma] \times \ldots \times [i_k\sigma : (i_k+1)\sigma]$ is a σ-box, then we say that p is *contained* in this σ-box if $i_1 = \lfloor p_1/\sigma \rfloor, \ldots, i_k = \lfloor p_k/\sigma \rfloor$. In this way, even if a point lies on the boundary of a σ-box, there is a unique σ-box containing it. ∎

In the rest of this section, σ-boxes will be sorted in lexicographical order. This ordering is obtained by comparing the "lower-left" corners of the σ-boxes lexicographically.

The data structure $DS(V, \sigma)$**:** Let L be the set of all σ-boxes that contain at least one point of V. We store the elements of L in a balanced binary search tree, sorted in lexicographical order. With each σ-box h, we store a list of all points of V that are contained in h. The elements in this list are stored in arbitrary order.

The query algorithm: Let $p = (p_1, \ldots, p_k)$ be a point in k-space and let $\delta \in \mathbb{R}$ such that $0 < \delta \leq \sigma$. We have to compute $f(p, V, \sigma, \delta)$.

1. We perform 3^k point location queries, with query points

   $$(p_1 + \epsilon_1, \ldots, p_k + \epsilon_k), \text{ where } \epsilon_1, \epsilon_2, \ldots, \epsilon_k \in \{-\delta, 0, \delta\}.$$

   That is, for each of these 3^k points r, we search in the binary search tree for the σ-box containing r.

2. If we do not find any σ-box, we output "$f(p, V, \sigma, \delta) = \infty$".

3. Otherwise, for each σ-box found, we walk through the list of points that is stored with it. Of all the points encountered in this way, we select a point $q \neq p$ that is closest to p. If $d(p, q) \leq \delta$, we output "$f(p, V, \sigma, \delta) = d(p, q)$" together with the point q. Otherwise, we output "$f(p, V, \sigma, \delta) = \infty$".

The running time of this algorithm depends on the following lemma.

Lemma 1 *Let V be a set of points in k-space and let $c \in \mathbb{N}$. Then, any k-dimensional cube C having sides of length $c\,\delta(V)$ contains at most $(ck+c)^k$ points of V.*

Proof: Partition the cube C into $(ck+c)^k$ subcubes with sides of length $\delta(V)/(k+1)$. Assume that C contains more than $(ck+c)^k$ points of V. Then one of the subcubes contains at least two points of V. These two points have a distance at most equal to the L_t-diameter of this subcube. But this diameter is at most $k \cdot \delta(V)/(k+1) < \delta(V)$, contradicting the fact that the minimal distance of V is equal to $\delta(V)$. ∎

Theorem 1 *Let V be a set of n points in k-space and let $\sigma \in \mathbb{R}$ such that $0 < \sigma \leq c\,\delta(V)$, with $c \in \mathbb{N}$ fixed. There exists a data structure such that, for each point p in k-space and for each $\delta \in \mathbb{R}$ satisfying $0 < \delta \leq \sigma$, we can find the value of $f(p, V, \sigma, \delta)$ in $O(\log n)$ time. The data structure has size $O(n)$ and can be built in $O(n \log n)$ time.*

Proof: To answer a query, we have to find a nearest neighbor of p, if this neighbor has distance at most δ to p. That is, we have to find all points of V that are inside the L_t-ball of radius δ centered at p. This ball is contained in the k-dimensional cube $[p_1 - \delta : p_1 + \delta] \times \ldots \times [p_k - \delta : p_k + \delta]$. Therefore, it suffices to compare p with all points of V that are in this cube. 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 belonging to the σ-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 query algorithm is correct.

If $W = \emptyset$, then certainly $W \subseteq W'$. So assume that $W \neq \emptyset$. Let $x = (x_1, \ldots, x_k)$ be a point in W. Assume w.l.o.g. that $x_i \geq p_i$ for $i = 1, \ldots, k$. Then $p_i \leq x_i \leq p_i + \delta$ for $i = 1, \ldots, k$. Let B be the σ-box whose list contains x. Assume that $x \notin W'$. Then B does not contain any of the 2^k points $(p_1 + \beta_1, \ldots, p_k + \beta_k)$, where $\beta_1, \ldots, \beta_k \in \{0, \delta\}$. These 2^k points are the corners of the k-dimensional cube

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

having sides of length δ. Since $x \in B'$, and since B does not contain any of the corner points of B', it follows that the σ-box B must have at least one side of length strictly less than δ. This is a contradiction, because the σ-boxes have sides of length σ and we assumed that $\delta \leq \sigma$. Hence, $x \in W'$ and, therefore, $W \subseteq W'$. This proves the correctness of the query algorithm.

Given a point r, we can compute the σ-box h containing r in constant time. Given this σ-box, we can search for it in the binary search tree in $O(k \log n)$ time. Since $\sigma \leq c\,\delta(V)$, it follows from Lemma 1 that each σ-box contains at most $(ck+c)^k$ points of V. It follows that the total query time is bounded by

$$O(3^k k \log n + 3^k (ck+c)^k) = O(\log n), \tag{1}$$

because k is a constant.

We only store non-empty σ-boxes. Moreover, each point of V is stored only once. Therefore, the data structure has size $O(n)$. It is clear that it can be built in $O(n \log n)$ time. ∎

3 Maintaining the closest pair under insertions

We apply Bentley's logarithmic method for decomposable searching problems [1] to the static data structure of the previous section. This gives a data structure that maintains the closest pair under insertion of points.

The dynamic data structure: Let V be the current set of points in k-space, and let n denote its size. Write n in the binary number system, $n = \sum_{i \geq 0} \alpha_i 2^i$, $\alpha_i \in \{0, 1\}$. Set V is partitioned into subsets: for each i such that $\alpha_i = 1$, there is one subset V_i, of size 2^i. The data structure stores the following information:

1. The current closest pair (P, Q) and its distance δ.
2. For each i such that $\alpha_i = 1$, a static data structure $DS(V_i, \sigma_i)$, where $\sigma_i \in \mathbb{R}$ satisfies $\delta(V) \leq \sigma_i \leq \delta(V_i)$. (That is, $c = 1$ w.r.t. Definition 1.)

The insertion algorithm: Suppose we want to insert point p. The algorithm makes two steps.

1. For each i such that $\alpha_i = 1$, we use the structure $DS(V_i, \sigma_i)$ to compute the value of $f(p, V_i, \sigma_i, \delta)$. If all these values are ∞, we continue with step 2. Otherwise, we compute an index j such that $f(p, V_j, \sigma_j, \delta)$ is minimal. The algorithm has found a point $q \in V_j$ which is a nearest neighbor of p in V_j. We set $(P, Q) := (p, q)$ and $\delta := d(p, q)$.
2. Let l be the integer such that $\alpha_0 = \alpha_1 = \ldots = \alpha_{l-1} = 1$, $\alpha_l = 0$. Let $V_l := \{p\} \cup V_0 \cup V_1 \cup \ldots \cup V_{l-1}$. We set $\sigma_l := \delta$, build a static data structure $DS(V_l, \sigma_l)$, and discard the structures $DS(V_0, \sigma_0), \ldots, DS(V_{l-1}, \sigma_{l-1})$ (thereby implicitly making the sets $V_0, \ldots, V_{l-1}$ empty).

Theorem 2 *The given dynamic data structure maintains the closest pair in a set of n points in k-space at a cost of $O((\log n)^2)$ amortized time per insertion. The data structure has size $O(n)$.*

Proof: Note that we can apply the results of Section 2 to the structures $DS(V_i, \sigma_i)$, because $0 < \sigma_i \leq \delta(V_i)$, and we only perform restricted post-office queries with $\delta = \delta(V) \leq \sigma_i$.

First we prove the correctness of the algorithm. Let V be the set of points at the start of the insertion algorithm. The closest pair is updated in step 1.

If all values $f(p, V_i, \sigma_i, \delta)$ are ∞, then for each i, the set $\{d(p, q) : q \in V_i, d(p, q) \leq \delta\}$ is empty. That is, for each i, all points in V_i have distance more than δ to p. In this case, there is no point in V having distance at most δ to p, and therefore the insertion of p does not cause a change of the closest pair.

Otherwise, there are finite values of $f(p, V_i, \sigma_i, \delta)$. Consider the integer j and the point q that are chosen in step 1. Then, since q is not only a nearest neighbor of p in V_j, but even in the entire set V, it follows that (p, q) is the closest pair in the new set $V \cup \{p\}$.

This proves that step 1 correctly maintains the closest pair. In step 2, the rest of the data structure is maintained. It is clear that the partition of the new set $V \cup \{p\}$ is according to the binary representation of the new number of points, i.e., $n+1$. Therefore, we only have to prove the bounds on the σ_i's.

Consider the integer l in step 2. Note that afterwards there are only values σ_i for $i \geq l$. Let $i > l$. Then the subset V_i and the value of σ_i did not change. Hence, it is still true that $\sigma_i \leq \delta(V_i)$. Moreover, it is clear that $\delta(V \cup \{p\}) \leq \delta(V)$. Since $\delta(V) \leq \sigma_i$, it follows that $\delta(V \cup \{p\}) \leq \sigma_i$.

It remains to show that $\delta(V \cup \{p\}) \leq \sigma_l \leq \delta(V_l)$. This follows because $\sigma_l = \delta(V \cup \{p\})$ and $V_l \subseteq V \cup \{p\}$.

This proves the correctness of the entire algorithm. The proofs of the complexity bounds are the same as in [1]. We repeat them here, because it indicates where there is room for improvement.

Using Theorem 1, it immediately follows that the dynamic data structure has size

$$O\left(\sum_{i:\alpha_i=1} 2^i\right) = O\left(\sum_{i=0}^{\log n} 2^i\right) = O(n).$$

Let $P(n)$ and $Q(n)$ denote the building time and query time of the static structure DS. Then, $P(n) = O(n \log n)$ and $Q(n) = O(\log n)$. In step 1 of the algorithm, we perform at most $1 + \lfloor \log n \rfloor$ restricted post-office queries, one in each structure $DS(V_i, \sigma_i)$. Therefore, the total time for step 1 is bounded by $O(Q(n) \log n)$.

Consider the integer l in step 2. The time for building $DS(V_l, \sigma_l)$ is $P(2^l)$. Suppose we start with an empty set V and execute a sequence of n insertions. Then a static structure for 2^l points is built at most $n/2^l$ times. Therefore, the total time for step 2 during these n insertions is bounded by

$$O\left(\sum_{l=0}^{\log n} \frac{n}{2^l} P(2^l)\right) = O(P(n) \log n). \tag{2}$$

Hence, the amortized time of step 2 is $O((P(n)/n) \log n)$. We have shown that the overall amortized insertion time is bounded by

$$O(Q(n) \log n + (P(n)/n) \log n) = O((\log n)^2). \tag{3}$$

This completes the proof. ∎

Remark: In Section 2, we mentioned that we do not have to know the value of $\delta(V_l)$ if we build a data structure $DS(V_l, \sigma_l)$. We said that the algorithm will guarantee $\sigma_l \leq c\,\delta(V_l)$.[1] This is indeed true in the above algorithm: If we build the structure $DS(V_l, \sigma_l)$, we set $\sigma_l := \delta = \delta(V \cup \{p\})$. Since $V_l \subseteq V \cup \{p\}$, it is clear that $\sigma_l \leq \delta(V_l)$.

Can we improve the insertion time of Theorem 2? In [7], Mehlhorn shows that for a certain class of algorithms, the logarithmic method of [1] is optimal. That is, if we take our algorithms from his class, we cannot improve the running time.

[1] Note that $c = 1$ in this section, see item 2 of the definition of the dynamic data structure.

Consider again step 1 of the insertion algorithm. Let $Q'(n)$ denote the time for one point location query in the data structure DS, i.e., the time needed to search in DS for the σ-box containing a query point. Then, according to (1),

$$Q(n) = O\left(3^k Q'(n) + 3^k(ck+c)^k\right),$$

where $Q(n)$ is the total query time of the static data structure DS. The total time for step 1 is bounded by

$$O(Q(n)\log n) = O\left(3^k Q'(n)\log n + 3^k(ck+c)^k \log n\right). \tag{4}$$

In step 1, we perform 3^k point location queries for each $i \in [0,\ldots,\lfloor \log n \rfloor]$. The query points, however, are the same for all i. If we denote by $Q''(n)$ the time to locate one query point in all structures $DS(V_i,\sigma_i), 0 \le i \le \lfloor \log n \rfloor$, we can write (4) as

$$O(Q(n)\log n) = O\left(3^k Q''(n) + 3^k(ck+c)^k \log n\right). \tag{5}$$

Note that $Q''(n) = O((\log n)^2)$. In order to improve the overall insertion time, we shall improve the bound on $Q''(n)$. In the next sections, we show that we can do this using fractional cascading. For this purpose, we have to define new orderings for σ_i-boxes. For the new data structure, we shall improve the number of comparisons to locate one point in all static structures from $O((\log n)^2)$ to $O(\log n)$. Unfortunately, because of the new orderings, each comparison takes $O(\log\log n)$ time. In this way, $Q''(n)$ will be improved to $O(\log n \log\log n)$. Hence, according to (5), the total time for step 1, which is the first term in (3), will be improved to $O(\log n \log\log n)$.

To improve step 2, we shall see that we can use the old structures for the sets $V_0,\ldots,V_{l-1}$ to build the structure for V_l in $O(2^l)$ comparisons. (In fact, to achieve this we need to maintain more information.) Again, each comparison takes $O(\log\log n)$ time. Thus, we can replace $P(n)$ in the second term of (3) by $O(n\log\log n)$, improving this term to $O(\log n \log\log n)$, too.

4 Orderings for σ_i-boxes

Throughout this section, $m \in \mathbb{N}$ and $\sigma_0, \sigma_1 \ldots, \sigma_m$ is a sequence of real numbers such that $\sigma_i \le \sigma_{i+1}$ and $\sigma_{i+1}/\sigma_i \in \mathbb{N}$ for $0 \le i < m$. We consider a sequence of σ_i-grids for $0 \le i \le m$. Note that the σ_i-grid is a refinement of the σ_{i+1}-grid, for $0 \le i < m$. That is, a σ_{i+1}-box only overlaps *complete* σ_i-boxes.

Lemma 2 *Let $A_i \ne B_i$ be two σ_i-boxes. Let $A_i \subseteq A_{i+1} \ldots \subseteq A_m$ and $B_i \subseteq B_{i+1} \ldots \subseteq B_m$ be the sequences of σ_l-boxes, $i \le l \le m$, containing A_i and B_i, respectively. Then, there exists an index $s \in [i \ldots m]$, such that $A_l \ne B_l$ for $l \in [i \ldots s]$ and $A_l = B_l$ for $l \in [s+1 \ldots m]$.*

Proof: Follows from the refinement property of sequence $\sigma_0, \sigma_1, \ldots, \sigma_m$. ∎

Our searching strategy in Section 6 uses the results of the search in the σ_i-grid for the search in the σ_{i+1}-grid. Therefore, the ordering in the σ_i-grid depends on the ordering in the σ_{i+1}-grid.

Definition 3 For $0 \leq i \leq m$, let $\leq_{lex}$ be the lexicographical ordering of σ_i-boxes in the σ_i-grid. The ordering $\leq_i$ for σ_i-boxes is inductively defined:

1. $i = m$: $a \leq_m b \iff a \leq_{lex} b$, i.e., σ_m-boxes are ordered lexicographically.
2. $i < m$: in this case, $\leq_{i+1}$ is already defined. Let a and b be σ_i-boxes, and let A and B be the σ_{i+1}-boxes such that $a \subseteq A$ and $b \subseteq B$. Then, if
   (a) $A = B$: $a \leq_i b \iff a \leq_{lex} b$
   (b) $A \neq B$: $a \leq_i b \iff A \leq_{i+1} B$. ∎

Note that if $\sigma_i = \sigma_{i+1}$, the orderings $\leq_i$ and $\leq_{i+1}$ are identical.

Lemma 3 *For each $0 \leq i \leq m$, $\leq_i$ is a total ordering on the set of σ_i-boxes.*

Proof: By backward induction on i, starting with $i = m$. First note that, for any σ, the lexicographical ordering $\leq_{lex}$ is a total ordering on the σ-boxes in the σ-grid.

$i = m$: in this case, $\leq_i$ coincides with $\leq_{lex}$ and is therefore a total ordering.

$i < m$: let us assume that $\leq_{i+1}$ is a total ordering on the σ_{i+1}-boxes. We shall show that $\leq_i$ is a total ordering on the σ_i-boxes. Let a, b and c be σ_i-boxes and let A, B and C be the σ_{i+1}-boxes containing a, b and c, respectively. Of course, $\leq_i$ is reflexive: $a \leq_i a$ holds because, in this case, $A = A$ and therefore $a \leq_i a \iff a \leq_{lex} a$ from case 2a of Definition 3, and $a \leq_{lex} a$ since $\leq_{lex}$ is reflexive.

To show that $\leq_i$ is antisymmetric, assume $a \leq_i b$ and $b \leq_i a$. We shall show that $a = b$. If $A = B$, then again case 2a of Definition 3 applies, and we have $a \leq_{lex} b \wedge b \leq_{lex} a$, and therefore $a = b$ holds. Otherwise, $A \neq B$. Then, case 2b of Definition 3 applies, and we have $A \leq_{i+1} B$ as well as $B \leq_{i+1} A$. Since $\leq_{i+1}$ is antisymmetric by the induction hypothesis, it follows that $A = B$, which is a contradiction.

Now we show that $\leq_i$ is transitive, i.e. $a \leq_i b \wedge b \leq_i c \implies a \leq_i c$. First note that, in any case, $a \leq_i b \implies A \leq_{i+1} B$. Therefore, we have $A \leq_{i+1} B$ and $B \leq_{i+1} C$. Since $\leq_{i+1}$ is transitive, $A \leq_{i+1} C$ holds. If $A \neq C$, then $a \leq_i c$. Otherwise, $A = C$. First, let us assume $A \neq B$. But then $B \leq_{i+1} C = A$ and $A \leq_{i+1} B$ hold, and from the antisymmetry of $\leq_{i+1}$, it follows that $A = B$, contradicting our assumption. Thus $A = B$, and then $A = B = C$. So, $a \leq_i b \wedge b \leq_i c$ implies $a \leq_{lex} b \wedge b \leq_{lex} c$, which implies $a \leq_{lex} c$. This, in turn, implies $a \leq_i c$. At this point, we have shown that $\leq_i$ is a partial ordering. To show that it is a total ordering, we have to show that for any two σ_i-boxes a and b, $a \leq_i b$ or $b \leq_i a$ holds. If $A = B$, this follows from $a \leq_i b \iff a \leq_{lex} b$. If $A \neq B$, then $a \leq_i b \iff A \leq_{i+1} B$ and the induction hypothesis implies that $A \leq_{i+1} B$ or $B \leq_{i+1} A$ holds. ∎

Remark: Let $0 \leq i < j \leq m$. Let a and b be σ_i-boxes, and let A (resp. B) be the σ_j-box containing a (resp. b). Then $a \leq_i b \implies A \leq_j B$.

This fact will be applied later in the following way: consider a list of σ_i-boxes that are sorted w.r.t. $\leq_i$. If we walk through this list, we visit certain σ_j-boxes. According to the property just mentioned, we visit these σ_j-boxes in non-decreasing $\leq_j$-order.

Lemma 4 *Let A_i and B_i be two σ_i-boxes, and let $A_i \neq B_i$. Let s be the index of Lemma 2, and let A_s and B_s be the σ_s-boxes containing A_i and B_i, resp. Then*

$$A_i \leq_i B_i \iff A_s \leq_{lex} B_s.$$

Proof: Let $A_i \subseteq A_{i+1} \ldots \subseteq A_m$ and $B_i \subseteq B_{i+1} \ldots \subseteq B_m$ be the sequences of σ_l-boxes, $i \leq l \leq m$, containing A_i and B_i, respectively. From Lemma 2, we know that s exists, and that $A_l \neq B_l$ for $i \leq l \leq s$ and $A_l = B_l$ for $l > s$. Recursively applying case 2b of Definition 3, we get

$$A_i \leq_i B_i \iff A_{i+1} \leq_{i+1} B_{i+1} \iff \ldots \iff A_s \leq_s B_s.$$

If $s = m$, then $A_s \leq_s B_s \iff A_s \leq_{lex} B_s$ from case 1 of Definition 3. Otherwise, $s < m$ and $A_{s+1} = B_{s+1}$. From case 2a of Definition 3, it follows that $A_s \leq_s B_s \iff A_s \leq_{lex} B_s$. In both cases, we obtain $A_i \leq_i B_i \iff A_s \leq_{lex} B_s$. ∎

Lemma 5 *Let A_i and B_i be σ_i-boxes. Comparison of A_i and B_i w.r.t. ordering $\leq_i$, i.e. deciding whether $A_i \leq_i B_i$ holds, can be done in time $O(\log m)$.*

Proof: Deciding whether A_i and B_i are equal can be done by simply comparing the coordinates of the boxes involved. So assume $A_i \neq B_i$ from now on.

Let $A_i \subseteq A_{i+1} \ldots \subseteq A_m$ and $B_i \subseteq B_{i+1} \ldots \subseteq B_m$ be the sequences of σ_l-boxes, $i \leq l \leq m$, containing A_i and B_i, respectively, and let s be the index of Lemma 2. From Lemma 4, $A_i \leq_i B_i \iff A_s \leq_{lex} B_s$. Since lexicographical comparison of two σ_s-boxes takes constant time, comparing A_i and B_i can be done in constant time, once index s is computed. From Lemma 2, we know that $A_l \neq B_l$ for $i \leq l \leq s$ and $A_l = B_l$ for $l > s$. For a fixed σ_l-grid, deciding whether A_i and B_i are in the same σ_l-box is easy: compute the σ_l-boxes containing A_i and B_i, respectively, and check if they are equal. Therefore, index s can be computed by a binary search on the grid indices in the interval $[i \ldots m]$ in time $O(\log m)$. It follows that the total time for deciding $A_i \leq_i B_i$ is $O(\log m)$. ∎

5 The improved data structure

In this section, we present a variant of the data structure of Section 3. The most important difference is that we link the structures representing V_i and V_{i+1}, for $0 \leq i < \lfloor \log n \rfloor$. First, we need two definitions. Let $\sigma_0, \sigma_1 \ldots, \sigma_m$ be a sequence of real numbers, as in the previous section.

Definition 4 For $0 \leq i \leq m$, b_i^∞ denotes the *σ_i-box at infinity*. By definition, this symbolic σ_i-box has the property that $h \leq_i b_i^\infty$ for any σ_i-box h. ∎

In order to link the structures for subsets V_i and V_{i+1}, we use a variant of fractional cascading. The data structure for V_i will essentially consist of a list of σ_i-boxes. From the list of σ_{i+1}-boxes, we shall "copy" a certain fraction of boxes into the list of σ_i-boxes, and these copies will have a pointer to their originals in the list of σ_{i+1}-boxes. To be able to include the σ_{i+1}-boxes that are copied in the list of σ_i-boxes, we must find a way to represent σ_{i+1}-boxes by σ_i-boxes.

Definition 5 Let $0 \leq i < m$. If H is a σ_{i+1}-box, then the *σ_i-copy* of H is the lexicographically smallest σ_i-box that is contained in H. The σ_i-copy of b^∞_{i+1} is b^∞_i. ∎

The improved data structure: Let V be the current set of points in k-space, and let n denote its size. Write n in binary, $n = \sum_{i=0}^{\lfloor \log n \rfloor} \alpha_i 2^i, \alpha_i \in \{0,1\}$. The set V is partitioned into subsets: for each $0 \leq i \leq \lfloor \log n \rfloor$, there is one subset V_i of size $\alpha_i 2^i$. The data structure stores the following information:

1. The current closest pair (P, Q) and its distance δ.

2. A sequence $\sigma_0, \sigma_1, \ldots, \sigma_{\lfloor \log n \rfloor}$ of grid sizes, satisfying $\sigma_i \leq \sigma_{i+1}$ and $\sigma_{i+1}/\sigma_i \in \mathbb{N}$ for $0 \leq i < \lfloor \log n \rfloor$, and $\delta(V) \leq \sigma_i \leq 2\,\delta(V_i)$ for $0 \leq i \leq \lfloor \log n \rfloor$. (Note that $\delta(\emptyset) = \infty$.) These values define the orderings $\leq_i$, $0 \leq i \leq \lfloor \log n \rfloor$.

3. For each $i \in [0 \ldots \lfloor \log n \rfloor]$, a list $V(i)$ storing the points of V_i in arbitrary order.

4. For each $i \in [0 \ldots \lfloor \log n \rfloor]$, an *augmented list* $A(i)$, storing σ_i-boxes sorted w.r.t. ordering $\leq_i$. These lists are defined as follows:

   Let $B(i)$ be the list consisting of all σ_i-boxes that contain at least one point of V_i. (Note that V_i and, hence, $B(i)$ may be empty.)

   (a) For $i = \lfloor \log n \rfloor$, let $C(i) := \{b^\infty_i\}$. Then $A(i)$ is the union of the two (disjoint) lists $B(i)$ and $C(i)$.

   (b) Let $0 \leq i < \lfloor \log n \rfloor$, and suppose that the augmented list $A(i+1)$ is defined already.

   Let $C(i)$ be the list consisting of σ_i-copies of every fourth element of list $A(i+1)$, plus $\{b^\infty_i\}$.

   The augmented list $A(i)$ is the union[2] of the lists $B(i)$ and $C(i)$. The elements in $A(i)$ are sorted w.r.t. $\leq_i$.

   For each $i \in [0, \ldots, \lfloor \log n \rfloor]$, each σ_i-box h in the list $A(i)$ contains

   - a list L^h_i of all points in V_i that are contained in h, if $h \in B(i)$. (A box $h \notin B(i)$ does not contain any point of $V(i)$.) Each point p in L^h_i contains a pointer to its copy in the list $V(i)$.
   - If h belongs to $C(i)$, it contains a *bridge*-pointer, which is a pointer to the σ_{i+1}-box $H \in A(i+1)$ such that h is the σ_i-copy of H.
   - If h belongs to $B(i)$, it contains a *next_bridge*-pointer, which is a pointer to the smallest (w.r.t. $\leq_i$) element in $A(i)$ that is not smaller than h and that contains a *bridge*-pointer. (Note that, if $h \in C(i)$, the *next_bridge*-pointer of h points to h itself.)

   For each $i \in [0, \ldots, \lfloor \log n \rfloor]$, each point p in the list $V(i)$ contains a pointer to the σ_i-box in $A(i)$ containing p.

[2]The union is set-theoretic, i.e., each element is stored only once.

5. For each $i \in [0, \ldots, \lfloor \log n \rfloor]$, lists $V_i^1, V_i^2, \ldots V_i^k$. List V_i^j stores the points of V_i sorted by their j-th coordinates. Each point in the list V_i^j has a pointer to its copy in the list $V(i)$.

This induces a *layered structure*, with *levels* $0 \leq i \leq \lfloor \log n \rfloor$, where each level corresponds to an augmented list $A(i)$ containing σ_i-boxes, together with the other parts of the structure (lists V_i^j, list $V(i)$) which are indexed by i. There are links, called *bridges*, between level i and level $i+1$, for $0 \leq i < \lfloor \log n \rfloor$.

Lemma 6 *The size of the improved data structure is $O(n)$.*

Proof: First, we show that $|A(i) \setminus \{b_i^\infty\}| \leq 2^{i+1}$ for each $0 \leq i \leq \lfloor \log n \rfloor$. Let $i = \lfloor \log n \rfloor$. Then, $|A(i) \setminus \{b_i^\infty\}| = |B(i)| \leq |V_i| = 2^i = 2^{\lfloor \log n \rfloor} \leq 2^{\lfloor \log n \rfloor + 1}$. If, for $0 \leq i < \lfloor \log n \rfloor$, $|A(i+1) \setminus \{b_{i+1}^\infty\}| \leq 2^{i+2}$, then $|A(i) \setminus \{b_i^\infty\}| \leq |B(i)| + |C(i) \setminus \{b_i^\infty\}| \leq |V_i| + |A(i+1) \setminus \{b_{i+1}^\infty\}|/4 \leq 2^i + 2^{i+2}/4 = 2^{i+1}$.

The closest pair and its distance, the sequence of grid sizes, the lists $V(i), 0 \leq i \leq \lfloor \log n \rfloor$, and the lists $V_i^j, 0 \leq i \leq \lfloor \log n \rfloor, 1 \leq j \leq k$, use space $O(n)$ altogether. It remains to show the space bound for the augmented lists $A(i)$. Each $A(i)$ uses space $O(2^i)$, since each record h of $A(i)$ consists of a constant number of fields. Therefore, the total space used by the lists $A(i)$ is $O(\sum_{i=0}^{\lfloor \log n \rfloor} 2^i) = O(n)$. ∎

6 The improved insertion algorithm

Suppose we want to insert point $p = (p_1, p_2, \ldots, p_k)$. Like the insertion algorithm of Section 3, the improved insertion algorithm makes two steps.

1. Update the closest pair.
2. Update the rest of the data structure.

We treat both steps separately.

6.1 Update the closest pair

We have to compute the values of $f(p, V_i, \sigma_i, \delta)$ for all i such that V_i is non-empty. These values are computed by issuing 3^k point location queries with query points

$$(p_1 + \epsilon_1, \ldots, p_k + \epsilon_k), \text{ where } \epsilon_1, \ldots, \epsilon_k \in \{-\delta, 0, \delta\},$$

at all levels of the layered structure. Each query is solved by a fractional cascading-like search using the bridges that link the levels of the layered structure. More precisely, each of the 3^k query points is located iteratively in levels $0, 1, \ldots, \lfloor \log n \rfloor$. Once the 3^k query points are located at all levels, the values $f(p, V_i, \sigma_i, \delta)$ can be computed as in the query algorithm of Section 2. Then we proceed as in step 1 of Section 3.

Point location in all lists $A(i)$: We give the algorithm to locate one point $r = (r_1, \ldots, r_k)$ in all lists $A(i), 0 \leq i \leq \lfloor \log n \rfloor$. More precisely, for each $0 \leq i \leq \lfloor \log n \rfloor$,

let a_i be the σ_i-box that contains r. Then we find the smallest (w.r.t. $\leq_i$) σ_i-box b_i in the list $A(i)$, such that $a_i \leq_i b_i$. Given b_i, we can easily check if r is contained in it. If it does, $a_i = b_i$ and we have located a_i. Otherwise, a_i is not stored in $A(i)$.

We start with $i = 0$. We do a linear search in the list $A(0)$ and find the smallest (w.r.t. $\leq_0$) σ_0-box b_0 such that $a_0 \leq_0 b_0$.
Now let $0 < i \leq \lfloor \log n \rfloor$, and assume that we have located b_{i-1} in $A(i-1)$. Then:

- follow the *next_bridge*-pointer from b_{i-1}, giving element c_{i-1};
- follow the *bridge*-pointer from c_{i-1} to element $\tilde{c}_i$ in list $A(i)$;
- Start in $\tilde{c}_i$ and walk back along the list $A(i)$ until we have reached the smallest (w.r.t. $\leq_i$) element b_i such that $a_i \leq_i b_i$.

The following lemma implies that we indeed have to walk back in the list $A(i)$, once we are in $\tilde{c}_i$.

Lemma 7 *For each* $1 \leq i \leq \lfloor \log n \rfloor$, $a_i \leq_i \tilde{c}_i$.

Proof: Let $1 \leq i \leq \lfloor \log n \rfloor$. First note that $a_{i-1} \leq_{i-1} c_{i-1}$. If $a_i = \tilde{c}_i$, then $a_i \leq_i \tilde{c}_i$. Assume that $a_i \neq \tilde{c}_i$. Since the query point r is contained in a_{i-1} and a_i, we have $a_{i-1} \subseteq a_i$. Moreover, $c_{i-1} \subseteq \tilde{c}_i$, because c_{i-1} is the σ_i-copy of $\tilde{c}_i$. Then $a_{i-1} \leq_{i-1} c_{i-1}$ implies $a_i \leq_i \tilde{c}_i$. ∎

The next lemma implies that if we start in $\tilde{c}_i$, we have to make at most four steps back to find b_i.

Lemma 8 *For each* $1 \leq i \leq \lfloor \log n \rfloor$, *there are at most three elements in the list* $A(i)$ *strictly between* b_i *and* $\tilde{c}_i$.

Proof: If we reach the front of list $A(i)$ within three or less steps backwards from $\tilde{c}_i$, the lemma is clear. So assume z_i is the σ_i-box that we reach by going four steps back in the list $A(i)$, starting in $\tilde{c}_i$. Since $\tilde{c}_i$ was copied, z_i was also copied. Let z_{i-1} be the σ_{i-1}-copy of z_i. Then, because $z_i \leq_i \tilde{c}_i$ and $z_i \neq \tilde{c}_i$, we have $z_{i-1} \leq_{i-1} c_{i-1}$. Since c_{i-1} is the first element after b_{i-1} that contains a bridge, we have $z_{i-1} \leq_{i-1} b_{i-1}$. As there are no elements in $A(i)$ between a_{i-1} and b_{i-1}, $z_{i-1} \leq_{i-1} a_{i-1}$ holds, which implies $z_i \leq_i a_i$. Now $a_i \leq_i b_i$ by definition, and we conclude $z_i \leq_i b_i$, i.e., b_i does not come earlier than z_i in the list $A(i)$. ∎

Lemma 9 *The algorithm locates a query point* r *in all lists* $A(i)$ *in* $O(\log n \log\log n)$ *time.*

Proof: First note that $|A(0)| \leq 3$. At level i, we perform $O(1)$ steps, each consisting of a comparison of two σ_i-boxes. From Lemma 5, each comparison takes time $O(\log\log n)$. ∎

Corollary 1 *Step 1 of the improved insertion algorithm takes* $O(\log n \log\log n)$ *time.*

Proof: Since k is a constant, it follows from (5) that the total time for step 1 is $O(Q''(n) + \log n)$, if $Q''(n)$ is the time to locate one query point in all lists $A(i)$. By Lemma 9, we have $Q''(n) = O(\log n \log\log n)$. ∎

6.2 Update the rest of the data structure

In this section, δ (resp. δ_{new}) denotes the minimal distance at the start (resp. end) of the insertion algorithm. That is, $\delta = \delta(V)$ and $\delta_{new} = \delta(V \cup \{p\})$. Let l be such that $\alpha_0 = \alpha_1 = \ldots = \alpha_{l-1} = 1, \alpha_l = 0$. We have to build new structures for the levels $0, 1, \ldots, l$. Note that afterwards, there will be new values $\sigma_0, \ldots, \sigma_l$, and hence, the orderings $\leq_0, \ldots, \leq_l$ will change. For the choice of the new values $\sigma_0, \ldots, \sigma_l$, we use the following lemma.

Lemma 10 *Let $\sigma_0, \ldots, \sigma_{\lfloor \log n \rfloor}$ be the grid sizes before the insertion of p, and let $\sigma := \sigma_0 / \lfloor \sigma_0 / \delta_{new} \rfloor$. Then $\sigma \leq \sigma_i$ as well as $\sigma_i / \sigma \in \mathbb{N}$ for $0 \leq i \leq \lfloor \log n \rfloor$, and $\sigma \leq 2\,\delta_{new}$.*

Proof: First we note that $\delta_{new} \leq \delta$. From the definition of the improved data structure, $\delta \leq \sigma_0$. Thus, $\lfloor \sigma_0 / \delta_{new} \rfloor \geq 1$, which implies $\sigma \leq \sigma_0$. Since $\sigma_0 \leq \sigma_i$ for any $0 \leq i \leq \lfloor \log n \rfloor$, we have $\sigma \leq \sigma_i$. Similarly, as $\sigma_0 / \sigma \in \mathbb{N}$ by definition and $\sigma_i / \sigma_{i-1} \in \mathbb{N}, 0 < i \leq \lfloor \log n \rfloor$, it follows that $\sigma_i / \sigma \in \mathbb{N}$.

Let $q := \lfloor \sigma_0 / \delta_{new} \rfloor$ and $r := \sigma_0 - q\,\delta_{new}$. Then $\sigma_0 = q\,\delta_{new} + r, 0 \leq r < \delta_{new}$, and $q \geq 1$. Now, $\sigma = (q\,\delta_{new} + r)/q = \delta_{new} + r/q \leq \delta_{new} + \delta_{new}/q \leq 2\,\delta_{new}$. ∎

Define $\sigma := \sigma_0 / \lfloor \sigma_0 / \delta_{new} \rfloor$. For $0 \leq i \leq \lfloor \log n \rfloor$, we define the ordering $\leq_i'$ for σ-boxes, as follows.

Let a and b be σ-boxes and let A and B be the σ_i-boxes containing a and b, respectively.

1. If $A = B$, then $a \leq_i' b \iff a \leq_{lex} b$.
2. If $A \neq B$, then $a \leq_i' b \iff A \leq_i B$.

Roughly speaking, this defines the "ordering of σ-boxes in the σ_i-grid". In Lemma 3, it was shown that $\leq_i$ is a total ordering. Using a proof that is completely analogous to that of Lemma 3, it can be shown that $\leq_i'$ is a total ordering on the σ-boxes.

Now we discuss the algorithm updating the levels $0, 1, \ldots, l$. We distinguish two cases. If $l = \lfloor \log n \rfloor + 1$, we set $V_l := \{p\} \cup V_0 \cup V_1 \cup \ldots \cup V_{l-1}$, $V_0 := V_1 := \ldots := V_{l-1} := \emptyset$ and define $\sigma_0 := \delta_{new}, \sigma_1 := \delta_{new}, \ldots, \sigma_l := \delta_{new}$. Then, we build the lists $V_l^j, 1 \leq j \leq k$ and the list $V(l)$. Finally, we build the augmented list $A(l)$, where σ-boxes are ordered lexicographically. By iteratively copying every fourth element, setting up *bridge*- and *next_bridge*-pointers etc., we build $A(l-1), \ldots, A(0)$. This is the same procedure as step 3 of the algorithm for the case $l \leq \lfloor \log n \rfloor$, except that we already have $A(l)$ here. Since the boxes are ordered lexicographically in all lists, comparison of two boxes takes $O(1)$ time, and therefore we can surely implement this rebuilding in $O(n \log n)$ time.

Otherwise, $l \leq \lfloor \log n \rfloor$. Since $a_{\lfloor \log n \rfloor} = 1$, we have $l \leq \lfloor \log n \rfloor - 1$. In this case, the update algorithm makes three steps:

I. For each $0 \leq i < l$:

   (a) Construct the list $B'(i)$ consisting of all σ-boxes that contain at least one point of V_i. This list is sorted w.r.t. $\leq_i'$.

(b) Transform $B'(i)$ into the list $B''(i)$ storing the same elements, but sorted w.r.t. $\leq'_{l+1}$.

Note: After step I, the lists $B''(0), B''(1), \ldots, B''(l-1)$ store σ-boxes, all sorted w.r.t. the same ordering $\leq'_{l+1}$. This will become the new ordering $\leq_l$.

II. Let $V_l := \{p\} \cup V_0 \cup V_1 \cup \ldots \cup V_{l-1}$, and $V_0 := V_1 := \ldots := V_{l-1} := \emptyset$.

For each $j = 1, 2, \ldots k$:

(a) Merge the lists $\{p\}, V_0^j, V_1^j, \ldots, V_{l-1}^j$ into V_l^j.

(b) Merge the lists $\{p\}, V(0), V(1), \ldots, V(l-1)$ into $V(l)$.

(c) Merge the lists $B''(0), B''(1), \ldots, B''(l-1)$ and the σ-box containing p into $B(l)$.

(d) Define $\sigma_0 := \sigma, \sigma_1 := \sigma, \ldots, \sigma_l := \sigma$.

Note: From Lemma 10, $\sigma_0 \leq \sigma_1 \leq \ldots \leq \sigma_{\lfloor \log n \rfloor}$ and $\delta_{new} \leq \sigma_i \leq 2\,\delta_{new} \leq 2\,\delta(V_i)$. Note that, according to the changes of the σ_i's, the orderings $\leq_0, \ldots, \leq_l$ change, too.

The list $B(l)$ consists of all σ_l-boxes that contain at least one point of V_l, sorted w.r.t. $\leq'_{l+1} \equiv \leq_l$.

At this point, we have updated the items 2, 3 and 5 of the definition in Section 5. As item 1 (the closest pair itself) was already updated in 6.1, only item 4 is left. Concerning that issue, we have already computed the new list $B(l)$, and, implicitly, also the lists $B(j)$ for $0 \leq j < l$ because we know that they are empty.

III. Construct the augmented lists $A(l), A(l-1), \ldots, A(0)$.

After this short description, we shall discuss the steps in more detail. Step I iteratively executes steps Ia and Ib, for $0 \leq i < l$.

Step Ia:

- Walk along the list $A(i)$, stripping off all elements that do not belong to $B(i)$. The list $B(i)$ is ordered w.r.t. $\leq_i$.
- Walk along the list $B(i)$. For each σ_i-box h in this list: walk along the points in L_i^h and determine all σ-boxes that contain at least one point of L_i^h. (Note that, according to Lemma 1, each list L_i^h contains $O(1)$ points.) Sort these (constant number of) σ-boxes lexicographically. The elements of the various L_i^h retain the pointers to their copies in the list $V(i)$ and, moreover, each point $p \in V(i)$ does not point to the σ_i-box h any more, but to the *σ-box containing p*.

This gives the list $B'(i)$ sorted w.r.t. $\leq'_i$.

Step Ib:

- Initialize the list R to the empty list. R will be a list of σ_{l+1}-boxes.

- Walk along the elements of $B'(i)$. For each σ-box h in this list: If h is not contained in the σ_{l+1}-box at the end of R, we add the σ_{l+1}-box containing h at the end of R. Note that this can only happen if we "enter" a new σ_{l+1}-box, i.e. h cannot be contained in one of the previous σ_{l+1}-boxes in R, see the remark after Lemma 3. By now, h is contained in the σ_{l+1}-box at the end of R. We give h a pointer to this box.

  Note: In this way, the list R contains σ_{l+1}-boxes, sorted w.r.t. $\leq_{l+1}$.

- For $j = 1, 2, \ldots, k$:
  Walk along the list V_i^j. For each point q in this list:

  - follow the pointer to the copy of q in the list $V(i)$;
  - follow the pointer to the σ-box in $B'(i)$ that contains q;
  - follow the pointer to the σ_{l+1}-box in R that contains q;
  - store q at the end of the list R_j^h, and give this copy of q a pointer to the copy of q in the list $V(i)$.

  Result: In step Ia, we obtained a list R of σ_{l+1}-boxes, sorted w.r.t. $\leq_{l+1}$. Now we have, for each σ_{l+1}-box $h \in R$, the lists $R_j^h, j = 1, 2, \ldots k$, which contain the points of $V_i \cap h$, sorted by their j-th coordinates. (By $V_i \cap h$, we mean the set of points in V_i that are *contained* in h in the sense of Definition 2.) Note that the set $V(i) \cap h$ and the lists R_j^h are connected by pointers in the same way as the set $V(i)$ and the lists V_i^j, for $j = 1, 2, \ldots k$.

- For each σ_{l+1}-box h in R: construct a list of σ-boxes that contain at least one point of $V_i \cap h$, sorted lexicographically. We call this list $L(h)$. We construct lists $L_1(h), L_2(h), \ldots, L_k(h)$ iteratively, where $L_j(h)$ is the list of j-dimensional σ-boxes that contain at least one point of $V(i) \cap h$, sorted lexicographically w.r.t. dimensions $1, 2, \ldots j$. Each point $p \in V(i) \cap h$ points to the j-dimensional σ-box h containing it. We say that a point $p = (p_1, \ldots, p_k)$ is *contained in a j-dimensional σ-box*, if $(p_1, \ldots, p_j)$ is contained in the σ-box belonging to the σ-grid of j-dimensional space. Clearly, $L_k(h) = L(h)$.

  By walking through the list R_1^h, grouping the points of $V_i \cap h$ in slabs of length σ according to their first coordinates, we get $L_1(h)$.

  Assume we have constructed $L_{j-1}(h)$. We show how to construct $L_j(h)$.

  For each of the elements $t \in L_{j-1}(h)$, initialize a list $L(t)$ to the empty list. $L(t)$ will be a list of j-dimensional σ-boxes, ordered w.r.t. to dimension j. Walk along the list R_j^h. For each point x in this list:

  - follow the pointer to the copy of x in $V(i)$;
  - follow the pointer to the $(j-1)$-dimensional σ-box containing x;
  - if x is not contained in the j-dimensional σ-box at the end of $L(t)$ — note that it cannot be contained in one of the previous boxes in $L(t)$ —, we add the j-dimensional σ-box containing x at the end of $L(t)$;

– give x a pointer to the last element of $L(t)$.

Then, list $L_j(h)$ is obtained by concatenating the lists $L(t)$ for each $t \in L_{j-1}(h)$, walking through $L_{j-1}(h)$ from head to tail. Finally, we concatenate the lists $L(h)$ for each σ_{l+1}-box h in list R by walking through R from head to tail. In this way, we obtain a list of σ-boxes covering the points in V_i which is sorted w.r.t. $\leq_{l+1}$. This list is called $B''(i)$ and is the final result of step Ib.

Step II: We consider the process of merging the lists $\{p\}, V(0), V(1), \ldots, V(l-1)$ into the new list $V(l)$. The other merges are similar.

1. $V' := \text{merge}(\{p\}, V(0))$; $i := 1$;
2. **while** $i < l$ **do** $V' := \text{merge}(V', V(i))$; $i := i + 1$ **od**;
3. $V(l) := V'$;

It follows that the total number of comparisons used by these merging processes is $O(\sum_{i=1}^{l-1} 2^i) = O(2^l)$, which is linear in the size of V_l.

Step III: We show how the list $A(i)$ can be constructed, given $A(i+1)$. Walk along the list $A(i+1)$, and construct $C(i)$, the list of σ_i-copies, according to the definition of the improved data structure in Section 5. For each element in $C(i)$, we set the *bridge*-pointer to the resp. original in $A(i+1)$.

Then we merge $B(i)$ and $C(i)$. During this merge, we keep in mind the elements of $B(i)$ between two consecutive elements of $C(i)$. Having reached such an element $x \in C(i)$, we let the *next_bridge*-pointer of the remembered elements point to x. This completes the construction of $A(i)$.

Lemma 11 *Step 2 of the improved insertion algorithm correctly maintains the data structure and runs in amortized time* $O(\log n \log\log n)$.

Proof: Let $P'(2^l)$ be the time needed by step 2 of the improved insertion algorithm when the set V_l is constructed. From the description above, it follows that the algorithm is correct and makes a linear number of steps, where each step takes time $O(\log\log n)$. Therefore, $P'(2^l) = O(2^l \log\log n)$ for $l < \lfloor \log n \rfloor$ and $P'(2^l) = O(n \log n)$ for $l = \lfloor \log n \rfloor + 1$. Note that we never have to build a set $V_{\lfloor \log n \rfloor}$. Similarly to (2), it follows that the amortized time for step 2 is bounded by

$$O\left(\frac{1}{n}\left(\sum_{l=0}^{\lfloor \log n \rfloor + 1} \frac{n}{2^l} P'(2^l)\right)\right) = O\left(\log n + \sum_{l=0}^{\lfloor \log n \rfloor - 1} \log\log n\right) = O(\log n \log\log n).$$

∎

We combine Lemma 6, Corollary 1 and Lemma 11. This gives the final result:

Theorem 3 *There exists a data structure that maintains the closest pair in a set of* n *points in* k*-space in* $O(\log n \log\log n)$ *amortized time per insertion. The data structure uses* $O(n)$ *space.*

Corollary 2 *The closest pair in a set of* n *points in* k*-space can be computed on-line in* $O(n \log n \log\log n)$ *time using* $O(n)$ *space.*

7 Maintaining the closest pair under semi-online updates

In this section, we extend the results of Section 3 to semi-online updates. Recall that a sequence of updates is called semi-online, if the insertions are on-line, but with each inserted point, we get an integer d indicating that the point will be deleted d updates from the moment of its insertion.

In [5, 11], the following general result is proved. Let $g : T \times T \to \mathbb{R}$ be a function and let V be a subset of T of size n. Suppose we have a static data structure D that stores V, such that for any $p \in T$, the value $\min\{g(p,q) : q \in V, p \neq q\}$ can be computed in $Q(n)$ time. Suppose this static structure has size $S(n)$ and that it can be built in $P(n)$ time. Let $m = \lfloor \log n \rfloor$.

Then there exists a data structure of size $O(S(n))$ that maintains a partition $V_0, V_1, \ldots, V_m$ of V, values $\min\{g(p,q) : q \in V_j, p \neq q\}$ for $0 \leq i \leq m$, $p \in V_i$, $i \leq j \leq m$, and the value

$$\min_{0 \leq i \leq m} \min_{p \in V_i} \min_{i \leq j \leq m} \min_{q \in V_j} g(p,q), \tag{6}$$

in $O((P(n)/n) \log n + Q(n) \log n + (\log n)^2)$ worst-case time per semi-online update.

This data structure is a generalization of the one obtained by applying the logarithmic method of [1]. The main difference is that values $\min\{g(p,q) : q \in V_j, p \neq q\}$ are maintained.

We want to apply this result to the closest pair problem. In [5, 11], the authors take $g(p,q) = d(p,q)$ and for D they take a data structure for the static post-office problem. Then, the value of (6) is equal to $\delta(V)$, the minimal distance in V. The complexity of the resulting update algorithm, however, is very high in the higher-dimensional case.

In this section, we show that instead of D, we can use the data structure DS for the restricted post-office problem of Section 2. This will improve the update time considerably. Let the function g be defined by

$$g(p,q,\delta) := \begin{cases} d(p,q) & \text{if } d(p,q) \leq \delta, \\ \infty & \text{otherwise.} \end{cases}$$

Note that in our case, the function g depends on three variables. Recall the definition of the function f, see Section 2.

Lemma 12 *Let V be a set of points in k-space and let $V_0, V_1, \ldots, V_m$ be a partition of V. Then*

$$\min_{0 \leq i \leq m} \min_{p \in V_i} \min_{i \leq j \leq m} f(p, V_j, \delta(V_j), \delta(V_j)) = \delta(V). \tag{7}$$

Proof: Let $0 \leq i \leq m$, $p \in V_i$ and $i \leq j \leq m$. Using the definition of the function f, it immediately follows that $\delta(V) \leq f(p, V_j, \delta(V_j), \delta(V_j))$. Since i, p and j are arbitrary, it follows that $\delta(V)$ is at most equal to the left-hand side in (7).

Let p and q be a closest pair in V. Let i and j be indices such that $p \in V_i$ and $q \in V_j$. Assume w.l.o.g. that $i \leq j$. Then, $d(p,q) = \delta(V) \leq \delta(V_j)$, and $f(p, V_j, \delta(V_j), \delta(V_j)) = d(p,q) = \delta(V)$. It follows that there is one term on the left-hand side of (7) that is at most equal to $\delta(V)$. Therefore, the minimum of all these terms is surely at most equal to $\delta(V)$. ∎

It immediately follows from Lemma 12 that

$$\min_{0 \leq i \leq m} \min_{p \in V_i} \min_{i \leq j \leq m} \min_{q \in V_j} g(p, q, \delta(V_j)) = \delta(V). \tag{8}$$

Theorem 4 *There exists a data structure that maintains the closest pair in a set of n points in k-space in $O((\log n)^2)$ worst-case time per semi-online update. The size of the data structure is $O(n)$.*

Proof: First note that the closest pair of a set V of size n can be computed (off-line) in $O(n \log n)$ time, using linear space, see [2, 8, 17]. We saw in Theorem 1 that the data structure $DS(V, \delta(V))$ has size $S(n) = O(n)$ and can be built in $O(n \log n)$ time, if $\delta(V)$ is given. Hence, $DS(V, \delta(V))$ can be built in $P(n) = O(n \log n)$ time. A query "given point p, compute $\min\{g(p, q, \delta(V)) : q \in V, p \neq q\}$" can be solved in $Q(n) = O(\log n)$ time, because

$$\min\{g(p, q, \delta(V)) : q \in V, p \neq q\} = f(p, V, \delta(V), \delta(V)).$$

Apply the result of [5, 11] mentioned above. Then we obtain a data structure of size $O(S(n)) = O(n)$ that maintains — among other things — the minimal value of the function g in

$$O((P(n)/n) \log n + Q(n) \log n + (\log n)^2) = O((\log n)^2)$$

time per semi-online update, in the worst case. By (8), the minimal value of g is equal to the minimal distance of the entire set V. The algorithm can easily be extended such that it not only maintains the minimal distance, but also a pair of points having this minimal distance. ∎

Remark: If we apply the results of the previous sections to the data structure of Theorem 4, then the update time still remains $O((\log n)^2)$. There are two reasons for this. First, if we build a structure $DS(V_i, \delta(V_i))$, we have to compute the value of $\delta(V_i)$. (In the previous sections, we took a value σ_i approximating $\delta(V_i)$.) It takes $O(|V_i| \log |V_i|)$ time to compute $\delta(V_i)$. Second, the data structure of Theorem 4 maintains more information than that of the previous sections. More precisely, values $\min\{g(p,q) : q \in V_j, p \neq q\}$ for $0 \leq i \leq m, p \in V_i, i \leq j \leq m$, are maintained. Maintaining these values takes $\Theta((\log n)^2)$ amortized time.

8 Concluding remarks

We have shown how the closest pair in a set of n points in k-space can be computed on-line, in $O(n \log n \log \log n)$ time. We started with a static data structure for the

restricted post-office problem. Then, we applied the logarithmic method to it. Next, we extended fractional cascading in a non-trivial way to a simple form of higher-dimensional fractional cascading. It would be interesting to know if this extended method can be applied to other higher-dimensional query problems.

We have shown that fractional cascading can be applied to improve the query time of a data structure that is obtained from the logarithmic method for decomposable searching problems. The resulting algorithm is faster than the lower bound proved for Mehlhorn's class of algorithms [7]. A natural question is whether this technique can be applied to other problems as well.

It is an open problem whether the $\log \log n$ term in our time bounds can be removed. This term is the time needed to compare two σ_i-boxes in the ordering $\leq_i$. Note that we really need the orderings to prove the correctness and the running time for the point location algorithm. However, it might be possible to define other orderings that are computable in constant time, while retaining the crucial properties of the orderings given in this paper.

Finally, it would be interesting to improve the update time for semi-online updates. Maybe the time bound can be improved for off-line sequences of updates. (Note that a sequence of off-line updates is a special case of a sequence of semi-online updates.)

References

[1] J.L. Bentley. *Decomposable searching problems.* Inform. Proc. Lett. **8** (1979), pp. 244-251.

[2] J.L. Bentley and M.I. Shamos. *Divide-and-conquer in multidimensional space.* Proc. 8th Annual ACM Symp. on Theory of Computing, 1976, pp. 220-230.

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

[4] M.T. Dickerson and R.S. Drysdale. *Enumerating k distances for n points in the plane.* Proc. 7th ACM Symp. on Computational Geometry, 1991, pp. 234-238.

[5] 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.

[6] H. Edelsbrunner, L.J. Guibas and J. Stolfi. *Optimal point location in a monotone subdivision.* SIAM J. Comput. **15** (1986), pp. 317-340.

[7] K. Mehlhorn. *Lower bounds on the efficiency of transforming static data structures into dynamic structures.* Math. Systems Theory **15** (1981), pp. 1-16.

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

[9] J.S. Salowe. *Shallow interdistance selection and interdistance enumeration.* To appear in Proceedings WADS, 1991.

[10] M.I. Shamos and D. Hoey. *Closest-pair problems.* Proc. 16th Annual IEEE Symp. on Foundations of Computer Science, 1975, pp. 151-162.

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

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

[13] M. Smid. *Dynamic rectangular point location, with an application to the closest pair problem.* Report MPI-I-91-101, Max-Planck-Institut für Informatik, Saarbrücken, 1991.

[14] M. Smid. *Maintaining the minimal distance of a point set in polylogarithmic time (revised version).* Report MPI-I-91-103, Max-Planck-Institut für Informatik, Saarbrücken, 1991. See also: Proc. 2nd Annual ACM-SIAM Symp. on Discrete Algorithms, 1991, pp. 1-6.

[15] M. Smid. *Rectangular point location and the dynamic closest pair problem.* Submitted to Second Annual International Symposium on Algorithms, Taiwan.

[16] 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.

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