Discrete Comput Geom 7:415–431 (1992)

Discrete & Computational Geometry
© 1992 Springer-Verlag New York Inc.

Maintaining the Minimal Distance of a Point Set in Polylogarithmic Time*

Michiel Smid

Max-Planck-Institut für Informatik, W-6600 Saarbrücken, Federal Republic of Germany

Abstract. A dynamic data structure is given that maintains the minimal distance in a set of n points in k-dimensional space in $O((\log n)^k \log\log n)$ amortized time per update. The size of the data structure is bounded by $O(n(\log n)^k)$. Distances are measured in the Minkowski L_t-metric, where $1 \leq t \leq \infty$. This is the first dynamic data structure that maintains the minimal distance in polylogarithmic time for fully on-line updates.

1. Introduction

One of the fundamental type of problems in computational geometry are proximity problems, where we are given a set of points and we want to compute the minimal distance among these points, or we want for each point its nearest neighbor. Such problems have been studied extensively, and many results are known. The earliest results were only concerned with planar point sets. For example, it is well known that the minimal euclidean distance between n points in the plane can be found in $O(n \log n)$ time, and this is optimal. Given a set of n planar points, a euclidean nearest neighbor can be computed for each point in the set, in $O(n \log n)$ time, which is also optimal. These results have been extended to optimal $O(n \log n)$ algorithms for both problems in arbitrary, but fixed, dimension, using an arbitrary L_t-metric. (See Preparata and Shamos [10] and Vaidya [15].)

In the L_t-metric, for $1 \leq t \leq \infty$, the distance $d_t(p, q)$ between two k-dimensional points $p = (p_1, \ldots, p_k)$ and $q = (q_1, \ldots, q_k)$ is defined by

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

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

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

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

Important examples are the L_1-metric also known as the Manhattan-metric, the L_2-metric which is the "usual" euclidean metric, and the L_∞-metric which is also known as the maximum-norm.

In this paper, we consider the problem of maintaining the minimal distance when points are inserted and deleted. Dobkin and Suri [5] considered the case where the updates are *semi-on-line*. A sequence of updates is called semi-on-line when the insertions arrive on-line, but with each inserted point we get an integer l indicating that the inserted point will be deleted l updates after the moment of insertion. They showed that in the plane, such updates can be performed at an amortized cost of $O((\log n)^2)$ time per semi-on-line update. This result was made worst-case by the author in [12].

Supowit [14] gives an algorithm that maintains the minimal distance in a set of k-dimensional points in $O((\log n)^k)$ amortized time per deletion. His method uses $O(n(\log n)^{k-1})$ space. This data structure does not support insertions of points.

For arbitrary updates on the minimal euclidean distance of a set of planar points, the first nontrivial result was by Overmars [8] and [9], who gave an $O(n)$ time update algorithm. His method uses $O(n \log \log n)$ space. Aggarwal *et al.* [1] showed that in a two-dimensional Voronoi diagram, points can be inserted and deleted in $O(n)$ time. This also leads to an update time of $O(n)$ for the minimal distance, using only $O(n)$ space.

In [13], the author gives an $O(n \log n)$ time algorithm that computes the $O(n^{2/3})$ smallest distances defined by a set of n points in k-dimensional space. This result is used to give a dynamic data structure of size $O(n)$, that maintains the minimal distance of a set of n points in k-space at a cost $O(n^{2/3} \log n)$ time per update. Using results of Dickerson and Drysdale [4] and Salowe [11], the update time can be improved to $O(n^{1/2} \log n)$.

In this paper, we give a data structure of size $O(n(\log n)^k)$ that maintains the minimal distance in $O((\log n)^k \log \log n)$ amortized time per update. The data structure is composed of a number of structures that solve similar—but simpler—problems. More precisely, we define so-called structures of type i that estimate the distance between two sets A and B, whose points have coordinates of opposite sign in a fixed set of $k - i$ positions. These structures are defined recursively for $i = 0, 1, \ldots, k$. For $i = k$, we get the final data structure that maintains the minimal distance.

The result gives the first fully dynamic data structure that maintains the minimal distance in polylogarithmic amortized time.

In Section 2, we define two versions of the structure of type 0. This structure stores two sets A and B that lie in two opposite k-dimensional quadrants, and it maintains a variable δ that gives a lower bound on the minimal distance between the sets A and B. If the minimal distance in $A \cup B$ is equal to the distance between A and B, then the value of δ is equal to this distance. The first version of this structure is presented for reasons of clarity. Then we give the second version, which makes it possible to speed up the building algorithm of the structure of type 1. In Section 3, we define similar structures of type i. Now the structure stores sets A and B that lie in spaces that intersect in some i-dimensional space. These structures are defined inductively, taking the structure of type 0 as the basis of the construction. The variable δ that is maintained by the structure of type i satisfies the same constraints as that for the structure of type 0. If $i = k$, the variable δ will be equal to the minimal distance in the entire point set.

In Section 4, we apply dynamic fractional cascading (see Chazelle and Guibas [3] and Mehlhorn and Näher [7]) to speed up the update algorithm of the structure of type 1. As a result, this also improves the update time of the complete structure. We finish the paper in Section 5 with some concluding remarks.

2. The Type 0 Data Structure

Before we define the data structure of type 0, we prove a lemma that will be used in the correctness proof of this structure. If C is a k-dimensional cube and V is a set of points in k-space, then all points of V that are on the boundary or in the interior of C are said to be *contained* in C.

Lemma 1. *Let V be a set of points in k-space. Consider a k-dimensional cube with sides of length s that contains more than $(k + 1)^k$ points of V. Then the minimal L_t-distance between points in V is less than s.*

Proof. Assume without loss of generality that the k-cube is equal to $[0:s] \times \cdots \times [0:s]$. Partition this cube into $(k + 1)^k$ subcubes

$$[i_1 s/(k + 1):(i_1 + 1)s/(k + 1)] \times \cdots \times [i_k s/(k + 1):(i_k + 1)s/(k + 1)],$$

where the i_j are integers such that $0 \le i_j \le k$ and $1 \le j \le k$. Since the cube contains at least $(k + 1)^k + 1$ points of V, there is one subcube that contains at least two points of V. Hence, there are two points in V that have a distance which is at most the L_t-diameter of this subcube. Since the L_t-diameter of the subcube is at most $k \times s/(k + 1) < s$, the minimal L_t-distance among all pairs of distinct points in V is less than s. □

In the final data structure that maintains the minimal distance we need several data structures that solve similar—but simpler—problems. In this section, we give the first of these structures.

Throughout this paper we fix $1 \le t \le \infty$, and we measure all distances in the L_t-metric. For point sets P and Q we define

$$d(P, Q) := \min\{d(p, q) \mid p \in P, q \in Q, p \neq q\}.$$

(We define $d(P, \varnothing) := d(\varnothing, Q) := \infty$.) We also fix the constant $c_k := 1 + (k + 1)^k$.

Next we fix a constant α, such that $0 < \alpha < \frac{1}{3}$. Recall that a binary search tree is called a BB[α]-tree if $\alpha \le n_w/n_v \le 1 - \alpha$ for all nodes v and w such that v is the father of w. Here n_u is the number of leaves in the subtree of u. (See [2] and [6] for details about BB[α]-trees.) We use BB[α]-trees as leaf-search trees. That is, elements are stored in sorted order in the leaves. These leaves are linked by pointers, from left to right and from right to left. Furthermore, there are pointers to the left and rightmost elements. Hence we can access the c smallest or largest elements in $O(c)$ time.

Let A and B be two sets of points in k-space. We assume that the points of A and B lie in opposite k-dimensional quadrants. Therefore, we may assume without loss of generality that $A \subseteq \mathscr{R}_\rho^- := (-\infty:0]^{k-1} \times (-\infty:\rho]$ and $B \subseteq \mathscr{R}_\rho^+ := [0:\infty)^{k-1} \times [\rho:\infty)$ for some real number ρ. (If the point $(0, \ldots, 0, \rho)$ belongs to $A \cup B$, then it belongs to A.) We want to maintain the minimal distance between points in A and points in B, when updates of the following type are performed. Only points that lie in $\mathscr{R}_\rho^- \cup \mathscr{R}_\rho^+$ are inserted and deleted. If a point of $\mathscr{R}_\rho^-$ is inserted or deleted, we say that the update "occurs" in set A. Otherwise, the update "occurs" in set B.

The data structure does not always have to give the minimal distance between A and B. It only has to if the minimal distance in the set $A \cup B$ is equal to the distance between A and B. The reason for this is the following: Later this data structure will be part of the data structure that maintains the minimal distance between all points. The part of the structure of the present section takes care of the distance between two subsets A and B. Another part of the data structure will take care of the distances between points in A, and yet another part considers distances in the set B. If the distance between A and B is "large," then the structure of this section is not relevant; other parts of the final structure will give the minimal distance in the complete set. If, however, the distance between A and B is equal to the minimal distance in the complete set, then the structure of this section delivers this minimal distance.

So A and B are sets of points in k-space such that $A \subseteq \mathscr{R}_\rho^-$ and $B \subseteq \mathscr{R}_\rho^+$. Let $a = |A|$, $b = |B|$, and $n = |A| + |B|$. It may be possible that $a = 0$ or $b = 0$ but $n = a + b > 0$. Let r be the point $(0, \ldots, 0, \rho)$ in k-space.

We give a dynamic data structure that maintains a variable $\delta \in \mathscr{R} \cup \{\infty\}$ such that:

(1) if A or B is empty, then $\delta = \infty$;
(2) if A and B are both nonempty, then $\delta = d(p, q)$ for some $p \in A$ and $q \in B$, $p \neq q$;
(3) if $d(A, B) = d(A \cup B, A \cup B)$, then $\delta = d(A, B)$.

We say that the structure stores the pair (A, B) and we call the structure of type 0, because the regions in which the sets A and B lie intersect in one point.

The Data Structure of Type 0 with Reference Point r

1. We maintain two BB[α]-trees T_A and T_B that store the points of A and B, sorted by their L_∞-distances from the reference point r. Points with equal L_∞-distances are stored in increasing lexicographic order.

2. If A or B is empty, the value of the variable δ is equal to ∞.
3. If A and B are both nonempty, let A_0 (resp. B_0) be the set of $\min(c_k, |A|)$ (resp. $\min(c_k, |B|)$) smallest—i.e., leftmost—points of T_A (resp. T_B). In this case, the value of δ is equal to $d(A_0, B_0)$.

First, we prove that the value of δ that is maintained by this data structure is correct.

Lemma 2. *The value of δ satisfies* (1), (2), *and* (3).

Proof. If A or B is empty, then $\delta = \infty$. Hence, (1) holds in this case. If A is empty and B consists of exactly one element, then $d(A, B) = d(A \cup B, A \cup B) = \infty$ and, hence, (3) holds. If A is empty and B contains more than one element, then $d(A, B) \neq d(A \cup B, A \cup B)$. Therefore, (3) also holds.

Now assume that A and B are both nonempty. It is clear that (2) holds. Let $a_0 = |A_0|$ and $b_0 = |B_0|$.

If $a_0 \leq c_k$ and $b_0 \leq c_k$, then $A_0 = A$ and $B_0 = B$ and, hence, $\delta = d(A, B)$. Thus, (3) holds in this case.

Assume that a_0 and b_0 are both greater than c_k and assume that $d(A, B) = d(A \cup B, A \cup B)$. We have to show that $\delta = d(A, B)$.

Let s_A (resp. s_B) be the maximum of the L_∞-distances of the points in A_0 (resp. B_0) to the reference point r. Let s be the minimum of s_A and s_B. Assume without loss of generality that $s = s_A$.

Consider the k-cube $[-s:0]^{k-1} \times [\rho - s:\rho]$. All points of A_0 are contained in this cube. Since $|A_0| \geq c_k > (k+1)^k$, it follows from Lemma 1 that $d(A, A) < s$. Therefore, $d(A \cup B, A \cup B) < s$. But then, by our assumption, we have $d(A, B) < s$. Since points in A and $B \backslash B_0$ are "separated" by a k-cube of side lengths $s_B \geq s$ we have $d(A, B \backslash B_0) \geq s$. Similarly, $d(A \backslash A_0, B) \geq s$. Therefore,

$$d(A, B) = \min(d(A_0, B_0), d(A \backslash A_0, B), d(A, B \backslash B_0)) = d(A_0, B_0).$$

But, by the definition of the data structure, $\delta = d(A_0, B_0)$. It follows that $\delta = d(A, B)$ and, hence, δ satisfies (3).

We are left with the case that $a_0 > c_k$ and $b_0 \leq c_k$. (The case $a_0 \leq c_k$ and $b_0 > c_k$ can be handled symmetrically.) Note that then $B_0 = B$. Assume that $d(A, B) = d(A \cup B, A \cup B)$. We have to show that $\delta = d(A, B)$.

Define s_A as above. Then we get in the same way as before that $d(A, B) < s_A$. We also get

$$d(A, B) = \min(d(A_0, B), d(A \backslash A_0, B)) = d(A_0, B).$$

Therefore, $\delta = d(A_0, B_0) = d(A_0, B) = d(A, B)$. This proves that δ satisfies (3). □

The Update Algorithm

To insert or delete a point p in a structure of type 0 for the pair (A, B) we do the following: Insert or delete the point in the tree T_A or T_B, depending on whether the update occurs in A or B. Then compare each of the $\min(c_k, |A|)$ smallest points in T_A with each of the $\min(c_k, |B|)$ smallest points in T_B and set δ to the minimal distance between these pairs of points.

Theorem 1. *The data structure of type* 0 *for the pair* (A, B) *has size* $O(n)$ *and can be built in* $O(n \log n)$ *time. In this data structure, points can be inserted and deleted at a cost of* $O(\log n)$ *time per update.*

Proof. The bounds on the size and the building time are clear. It is also clear that the update algorithm correctly maintains the data structure of type 0 and that it uses $O(\log n)$ time per update. □

We now change the data structure of type 0. The reason for doing this is twofold. Using the alternative structure of type 0 we are able to improve the building time of the structure of type 1. It will also allow us to use dynamic fractional cascading in a relatively straightforward manner.

The Alternative Data Structure of Type 0

Let $A \subseteq (-\infty:0]^{k-1} \times (-\infty:\rho]$ and $B \subseteq [0:\infty)^{k-1} \times [\rho:\infty)$ be two sets of points of sizes $a \geq 0$ and $b \geq 0$ such that $n = a + b > 0$. Let r be the point $(0, \ldots, 0, \rho)$ in k-space. The alternative data structure of type 0 for the pair (A, B) with reference point r consists of the following:

1. Two BB[α]-trees T_A^k and T_B^k that store the points of A and B sorted by their kth coordinates. Points with equal kth coordinates are stored in increasing lexicographic order.
2. Two BB[α]-trees $T_{A,k-1}^{r;\infty}$ and $T_{B,k-1}^{r;\infty}$ that store the points of A and B sorted by their L_∞-distances from the reference point r, where we take in the distance computations only the $k-1$ first coordinates into account. Points with equal L_∞-distances to r—for the $k-1$ first coordinates—are stored in increasing lexicographical order.
3. Let A' be the set of points $p = (p_1, \ldots, p_k)$ in A such that $d_\infty(p, r) = |p_k - \rho|$. Then there are BB[$\alpha$]-trees $T_{A'}$ and $T_{A \backslash A'}$ that store the points of A' and $A \backslash A'$ sorted by their L_∞-distances from the reference point r. Points with equal L_∞-distances are stored in increasing lexicographic order.
4. There are BB[α]-trees $T_{B'}$ and $T_{B \backslash B'}$ that are defined similarly.
5. If A or B is empty, then $\delta = \infty$.
6. If A and B are both nonempty, let A_0 (resp. B_0) be the subset of A (resp. B), consisting of the $\min(c_k, |A|)$ (resp. $\min(c_k, |B|)$) smallest points with respect to the L_∞-distance to the reference point r. Then $\delta = d(A_0, B_0)$.

Note that the point set A_0 can be obtained from the trees $T_{A'}$ and $T_{A \backslash A'}$: Take the $\min(c_k, |A'|)$ smallest points in $T_{A'}$ and the $\min(c_k, |A \backslash A'|)$ smallest points in $T_{A \backslash A'}$. Then A_0 consists of the $\min(c_k, |A|)$ smallest of these points.

Therefore Lemma 2 and Theorem 1 remain valid for the alternative data structure of type 0.

In the rest of this section, we prove a lemma that will be used later. An ordered sequence S is called a *subsequence* of an ordered sequence T if the following holds for all p and q: If p and q are elements in S such that p is to the left of q, then p is also to the left of q in the sequence T.

We introduce the following notations. Let X be a set of points in k-space and let r be a point in k-space. Then S_X^k denotes the sequence consisting of the points of X sorted by their kth coordinates, and $S_{X,k}^{r,\infty}$ denotes the sequence consisting of the points of X sorted by their L_∞-distances to point r. By $S_{X,k-1}^{r,\infty}$ we denote the sequence consisting of the points of X sorted by their L_∞-distances to point r, where we take in the distance computations only the $k-1$ first coordinates into account.

Lemma 3. *Let B be a set of points in k-space, σ a real number, and s the point $(0, \ldots, 0, \sigma)$ in k-space. Partition B into sets C and D such that $C \subseteq \{p \in B: p_k \leq \sigma\}$ and $D \subseteq \{p \in B: p_k \geq \sigma\}$ where p_k is the kth coordinate of p. Let $C' := \{p \in C: d_\infty(p, s) = |p_k - \sigma|\}$ and $D' := \{p \in D: d_\infty(p, s) = |p_k - \sigma|\}$. Then:*

1. *$S_{C',k}^{s,\infty}$ is a subsequence of the inverted sequence S_C^k;*
2. *$S_{D',k}^{s,\infty}$ is a subsequence of S_D^k;*
3. *$S_{C\setminus C',k}^{s,\infty}$ is a subsequence of $S_{C,k-1}^{0,\infty}$ where 0 stands for the zero vector in k-space;*
4. *$S_{D\setminus D',k}^{s,\infty}$ is a subsequence of $S_{D,k-1}^{0,\infty}$.*

Proof. We only give the proof for the C-sequences. To prove the first claim, let p and q be elements of $S_{C',k}^{s,\infty}$ such that p is to the left of q in this sequence. Then $d_\infty(p, s) \leq d_\infty(q, s)$. Since p and q are both in C', it follows that $|p_k - \sigma| \leq |q_k - \sigma|$. Then, using that $p_k, q_k \leq \sigma$, it follows that $q_k \leq p_k$. Hence, p is to the right of q in the sequence S_C^k.

If $x = (x_1, \ldots, x_k)$ is a point in k-space, then x' will denote the point $(x_1, \ldots, x_{k-1})$ in $(k-1)$-space. To prove the third claim, let p and q be elements of $S_{C\setminus C',k}^{s,\infty}$ such that p is to the left of q in this sequence. Then $d_\infty(p, s) \leq d_\infty(q, s)$. Since $p, q \notin C'$, we have $d_\infty(p, s) = d_\infty(p', s')$ and $d_\infty(q, s) = d_\infty(q', s')$. Since $0' = s'$, it follows that $d_\infty(p', 0') = d_\infty(p, s) \leq d_\infty(q, s) = d_\infty(q', 0')$. Thus, p is to the left of q in the sequence $S_{C,k-1}^{0,\infty}$. □

3. The Type *i* Data Structure

In this section we recursively define the structure of type i that maintains the "distance" between two point sets, whose points have coordinates of opposite sign in $k - i$ positions. The variable that is maintained by this structure satisfies the same requirements as in (1), (2), and (3). The structure of type i uses the structure of type $(i - 1)$ as a building block.

Let i be an integer such that $0 \leq i \leq k$. Let A and B be two sets of points in k-dimensional space such that $a = |A| \geq 0$, $b = |B| \geq 0$, and $n = a + b > 0$. We assume that the points of A and B have coordinates of opposite sign in a fixed set of $k - i$ positions. Therefore, we may assume without loss of generality that the points of A lie in the space $\mathscr{R}_i^- := (-\infty : 0]^{k-i} \times \mathscr{R}^i$, and the points of B lie in $\mathscr{R}_i^+ := [0 : \infty)^{k-i} \times \mathscr{R}^i$.

If $0 \leq i < k$, the sets A and B are supposed to be disjoint, whereas for $i = k$ these sets must be equal.

We want to maintain the "minimal distance" between points in A and points in B when updates of the following type are performed. Only points that lie in $\mathscr{R}_i^- \cup \mathscr{R}_i^+$ are inserted and deleted. If a point in $\mathscr{R}_i^-$ is inserted or deleted, then the update "occurs" in set A. Otherwise, the update "occurs" in set B.

We give a dynamic data structure of type i that maintains a variable $\delta = \delta_i(A, B) \in \mathscr{R} \cup \{\infty\}$ such that:

(4) if A or B is empty or if $|A \cup B| \leq 1$, then $\delta = \infty$;
(5) if A and B are both non empty and $|A \cup B| \geq 2$, then $\delta = d(p, q)$ for some $p \in A$ and $q \in B$, $p \neq q$;
(6) if $d(A, B) = d(A \cup B, A \cup B)$, then $\delta = d(A, B)$.

We say that the structure stores the pair (A, B). In the rest of this section we use the same constant $c_k = (k + 1)^k + 1$ as before.

The Data Structure of Type i

If $i = 0$, we take the alternative structure of type 0 that was defined in the previous section. So let $1 \leq i \leq k$ and assume that the structure of type $(i - 1)$ is defined already. The structure of type i consists of the following.

1. An augmented BB[α]-tree T that contains the points of $A \cup B$ in its leaves, sorted by their $(k - i + 1)$th coordinates. (Points with equal $(k - i + 1)$th coordinates are stored in increasing lexicographic order.) In each node u we store a hyperplane H_u: $x_{k-i+1} = \sigma_u$, where σ_u is the maximal $(k - i + 1)$th coordinate of all points in the left subtree of u. Let A_u (resp. B_u) denote the subset of A (resp. B) that is stored in the subtree of u.

   Any node u in T contains the following additional information.
2. If u is a leaf, then it contains the value $\delta_i(A_u, B_u) = \infty$.
3. If u is not a leaf let v (resp. w) be the left (resp. right) son of u. Note that

$$\begin{aligned} A_v &\subseteq (-\infty : 0]^{k-i} \times (-\infty : \sigma_u] \times \mathscr{R}^{i-1}, \\ A_w &\subseteq (-\infty : 0]^{k-i} \times [\sigma_u : \infty) \times \mathscr{R}^{i-1}, \\ B_v &\subseteq [0 : \infty)^{k-i} \times (-\infty : \sigma_u] \times \mathscr{R}^{i-1}, \\ B_w &\subseteq [0 : \infty)^{k-i} \times [\sigma_u : \infty) \times \mathscr{R}^{i-1}. \end{aligned}$$

   Therefore, the points in the sets A_v and B_w have coordinates of "opposite" sign in the first $k - i + 1$ positions. (In the $(k - i + 1)$th position, the coordinates have opposite sign with respect to their difference to σ_u.) Similarly for the sets A_w and B_v. Moreover, $A_v, A_w \subseteq \mathscr{R}_i^-$ and $B_v, B_w \subseteq \mathscr{R}_i^+$.

(a) Node u contains two (pointers to) structures of type $(i - 1)$, one structure for the pair (A_v, B_w) and the other structure for the pair (A_w, B_v). These structures maintain variables $\delta_{i-1}(A_v, B_w)$ and $\delta_{i-1}(A_w, B_v)$.
(b) The sons of u contain variables $\delta_i(A_v, B_v)$ and $\delta_i(A_w, B_w)$. The value of $\delta_i(A_u, B_u)$ corresponding to node u satisfies

$$\delta_i(A_u, B_u) = \min(\delta_i(A_v, B_v), \delta_i(A_w, B_w), \delta_{i-1}(A_v, B_w), \delta_{i-1}(A_w, B_v)).$$

Remark. If $i = k$, the sets A and B are equal. In this case, each point in $A \cup B$ is stored exactly once in the BB[α]-tree T. Each node u needs only one (pointer to a) structure of type $(k - 1)$, because the pairs (A_v, B_w) and (A_w, B_v) are equal in this case.

Lemma 4. *For any node u of T, the value of $\delta_i(A_u, B_u)$ satisfies* (4), (5), *and* (6).

Proof. First note that the sets A and B are disjoint if $0 \le i < k$. Therefore, $|A \cup B| \ge 2$ if A and B are both nonempty. That is, the conditions about $|A \cup B|$ in (4) and (5) do not play a role if $0 \le i < k$.

We saw in Lemma 2 that the claim holds for $i = 0$. So let $1 \le i \le k$, and assume that the claim is true for the structure of type $(i - 1)$.

Consider the BB[α]-tree T. Let u be a leaf of T. Note that $|A_u \cup B_u| = 1$ and $\delta_i(A_u, B_u) = \infty$. If one of A_u and B_u is empty, then (4) and (6) hold. Otherwise, if A_u and B_u are both nonempty, these two sets must be equal and have size one. In this case, (4) holds and (5) does not apply. Also (6) holds because $d(A_u, B_u) = \infty$. It follows that the lemma holds for all leaves in T.

Let u be a nonleaf node in T, and let v and w be the left and right sons of u. Assume inductively that the values of $\delta_i(A_v, B_v)$ and $\delta_i(A_w, B_w)$ are correct. By the induction hypothesis, the values of $\delta_{i-1}(A_v, B_w)$ and $\delta_{i-1}(A_w, B_v)$ are correct. By definition,

$$(7) \quad \delta_i(A_u, B_u) = \min(\delta_i(A_v, B_v), \delta_i(A_w, B_w), \delta_{i-1}(A_v, B_w), \delta_{i-1}(A_w, B_v)).$$

If A_u is empty, then A_v and A_w are also empty. Therefore, all terms on the righthand side of (7) are infinite. If follows that in that case $\delta_i(A_u, B_u) = \infty$. Hence, in this case, (4) holds for node u. Since u is not a leaf and since each point in $A_u \cup B_u$ is stored only once we have $|A_u \cup B_u| \ge 2$ and, hence,

$$d(A_u \cup B_u, A_u \cup B_u) < \infty = d(A_u, B_u).$$

Therefore, (6) does not apply in this case. The same holds if B_u is empty.

So assume that A_u and B_u are both nonempty. Note that (4) does not apply, because $|A_u \cup B_u| \ge 2$. At least one of the terms on the right-hand side of (7) is finite. Using the induction hypothesis, it follows easily that (5) holds.

Now suppose that $d(A_u, B_u) = d(A_u \cup B_u, A_u \cup B_u)$. In order to prove (6), we have to show that $\delta_i := \delta_i(A_u, B_u) = d(A_u, B_u)$. It follows from (5) that $\delta_i \ge d(A_u, B_u)$.

So it remains to be shown that $\delta_i \leq d(A_u, B_u)$. Clearly,

$$d(A_u, B_u) = \min(\mathrm{d}(\mathrm{A}_v, B_v), d(A_w, B_w), d(A_v, B_w), d(A_w, B_v)).$$

Take r and s from $\{v, w\}$ such that $d(A_u, B_u) = d(A_r, B_s)$. We have

$$\begin{aligned} d(A_u \cup B_u, A_u \cup B_u) &\leq d(A_r \cup B_s, A_r \cup B_s) \\ &\leq d(A_r, B_s) \\ &= d(A_u, B_u) \\ &= d(A_u \cup B_u, A_u \cup B_u). \end{aligned}$$

Hence, all inequalities above are in fact equalities. Therefore, $d(A_r, B_s) = d(A_r \cup B_s, A_r \cup B_s)$. Using the induction hypothesis, it follows from (6) that $\delta_j(A_r, B_s) = d(A_r, B_s)$ where $j = i$ if $r = s$ and $j = i - 1$ otherwise. Thus, $\delta_i \leq \delta_j(A_r, B_s) = d(A_r, B_s) = d(A_u, B_u)$. This finishes the proof. □

The definition of the structure of type i immediately leads to a recursive building algorithm. Let $T_i(n)$ denote the building time for sets A and B of total size n. Then by Theorem 1, $T_0(n) = O(n \log n)$ and for $i > 0$,

$$T_i(n) = 2T_i(n/2) + 2T_{i-1}(n) + O(n). \tag{8}$$

The $O(n)$ term is the time needed to find the median of the $(k - i +)$th coordinates and for splitting the set according to this median. This recurrence solves to $T_i(n) = O(n(\log n)^{i+1})$.

We now show how Lemma 3 can be used to improve the building time for the structure of type 1 to $O(n \log n)$.

Building the Structure of Type 1

Let $A \subseteq (-\infty:0]^{k-1} \times \mathscr{R}$ and $B \subseteq [0:\infty)^{k-1} \times \mathscr{R}$.

Before we start building the data structure, we do a kind of presorting: First, we sort the points of A by their kth coordinates and store them in the doubly-linked list S_A^k. Similarly, we construct the doubly-linked list S_B^k for the set B. Next, we sort the points of A by their L_∞-distances from the origin 0, where we take in the distance computations only the $k - 1$ first coordinates into account. We store the resulting sorted set in the linked list $S_{A,k-1}^{0,\infty}$. Similarly, we construct the linked list $S_{B,k-1}^{0,\infty}$ for B.

Now the actual building of the data structure of type 1 begins. We find the median σ of the kth coordinates of the points in $A \cup B$. Let s denote the point $(0, \ldots, 0, \sigma)$ in k-space. We partition A into equal-sized sets $A_1 := \{p \in A : p_k \leq \sigma\}$ and $A_2 := \{p \in A : p_k \geq \sigma\}$. Similarly, we partition B into B_1 and B_2.

We walk through the list S_A^k and make two sorted doubly-linked lists $S_{A_1}^k$ and $S_{A_2}^k$ out of them. In the same way, we make two doubly-linked lists $S_{B_1}^k$ and $S_{B_2}^k$.

Similarly, we walk through $S_{A,k-1}^{0,\infty}$ and make two sorted linked lists $S_{A_1,k-1}^{0,\infty}$

and $S^{0,\infty}_{A_2,k-1}$ out of them. In the same way, we make two linked lists $S^{0,\infty}_{B_1,k-1}$ and $S^{0,\infty}_{B_2,k-1}$ out of $S^{0,\infty}_{B,k-1}$.

Let r be the root of the final data structure of type 1. We store in r the hyperplane H_r: $x_k = \sigma$.

In the root, we have to store data structures of type 0 for the pairs (A_1, B_2) and (A_2, B_1). These two structures have reference point $s = (0, \ldots, 0, \sigma)$. We only describe how the first structure is built.

We store the points of $S^k_{A_1}$ in a perfectly balanced binary search tree $T^k_{A_1}$. In the same way, we build the perfectly balanced binary search tree $T^k_{B_2}$ using the list $S^k_{B_2}$.

Next we store the points of $S^{0,\infty}_{A_1,k-1}$ and $S^{0,\infty}_{B_2,k-1}$ in the perfectly balanced binary search trees $T^{s,\infty}_{A_1,k-1}$ and $T^{s,\infty}_{B_2,k-1}$. (Note that the ordering only depends on the $k-1$ first coordinates. Therefore, the points in these trees are indeed correctly sorted.)

Let $A'_1 := \{p \in A_1 : d_\infty(p, s) = |p_k - \sigma|\}$. We make a linear scan through the reversed sequence $S^k_{A_1}$ to obtain the sequence $S^k_{A'_1}$. (See Lemma 3.) Then we store the points of this sorted list $S^k_{A'_1}$ in the perfectly balanced binary search tree $T_{A'_1}$. Similarly, we make a scan through the list $S^{0,\infty}_{A_1,k-1}$, select the points of A_1, and store them in the perfectly balanced binary search tree $T_{A_1 \setminus A'_1}$. (See Lemma 3.)

In the same way, we build perfectly balanced binary search trees $T_{B'_2}$ and $T_{B_2 \setminus B'_2}$ where $B'_2 := \{p \in B_2 : d_\infty(p, s) = |p_k - \sigma|\}$.

To complete the data structure of type 0 for the pair (A_1, B_2) we compute the value of $\delta_0(A_1, B_2)$ according to its definition in Section 2. It is clear how to do this in constant time, since we can find the $\min(|A_1|, c_k)$ smallest elements in the appropriate trees in constant time.

We are almost finished in the root r. We only need the value of $\delta_1(A, B)$. This value will be computed in the final stage of the algorithm.

We build data structures of type 1—without the δ_1-values—for the pairs (A_1, B_1) and (A_2, B_2) using the same algorithm recursively. Note that at the start of these recursive calls we have the sorted sequences $S^k_{A_j}$, $S^k_{B_j}$, $S^{0,\infty}_{A_j,k-1}$, and $S^{0,\infty}_{B_j,k-1}$ for $j = 1, 2$. These structures form the left and right subtree of the root r.

At this moment we have constructed the skeleton tree of the data structure of type 1 and all structures of type 0 that are stored with the nodes of this skeleton tree. We walk through this tree in postorder and compute the values of δ_1 for the nodes by setting $\delta_i(A_u, B_u) := \infty$ for each leaf u, and

$$\delta_i(A_u, B_u) := \min(\delta_i(A_v, B_v), \delta_i(A_w, B_w), \delta_{i-1}(A_v, B_w), \delta_{i-1}(A_w, B_v))$$

for each node u with sons v and w.

This completes the building algorithm for the data structure of type 1. We analyze its running time. It takes $O(n \log n)$ time to do the presorting. Let $P(n)$ denote the building time after the presorting step. Then it follows from the above algorithm that $P(n) = O(n) + 2P(n/2)$ and, hence, $P(n) = O(n \log n)$. The final walk through the tree in postorder takes an additional amount of $O(n)$ time.

This proves that the complete building time $T_1(n)$ is bounded by $O(n \log n)$. Using (8), it follows that $T_i(n) = O(n(\log n)^i)$ for $1 \leq i \leq k$.

Lemma 5. *Let $1 \leq i \leq k$. The data structure of type i for the pair (A, B) can be built in $O(n(\log n)^i)$ time and has size $O(n(\log n)^i)$.*

Proof. The proof of the building time follows from the above discussion. The size of the data structure satisfies the same recurrence relation as the building time. □

The Update Algorithm

Suppose we want to insert or delete a point p in a structure of type i for the pair (A, B). Assume that the update occurs in B. If $i = 0$, we use the algorithm of Section 2.

Otherwise, $1 \leq i \leq k$. We search with the kth coordinate of p in the BB[α]-tree T to find the leaf x where the update takes place. For each node u that we encounter during this walk we do the following: Let v and w be its two sons and assume that the walk proceeds to v. Then we insert or delete point p in the structure of type $(i-1)$ for the pair (A_w, B_v) using the same algorithm recursively. Note that this will lead to a new value of $\delta_{i-1}(A_w, B_v)$.

Assume that we have to insert p. Then we give the leaf x two new sons y and z which are data structures of type i for the appropriate pairs. In node x itself we build two data structures of type $(i-1)$ for the appropriate pairs. The case where p has to be deleted can be handled similarly.

Next we walk back to the root, starting at the leaf where the update has taken place. At each node u we encounter we set

$$\delta_i(A_u, B_u) := \min(\delta_i(A_v, B_v), \delta_i(A_w, B_w), \delta_{i-1}(A_v, B_w), \delta_{i-1}(A_w, B_v)),$$

where v and w are the sons of u.

Finally, we again walk along this path back to the root and rebalance the BB[α]-tree by means of rotations, as described in [2] and [6]. Note that if a rotation is done at a node, we have to rebuild a constant number of type $(i-1)$ data structures. The algorithm for rebuilding structures of type 0 makes use of Lemma 3.

Remark. If $i = k$, then the sets A and B are equal. Although the update occurs in both sets, we insert or delete the point only once.

Lemma 6. *In the data structure of type i for the pair (A, B), points can be inserted and deleted at a cost of $O((\log n)^{i+1})$ amortized time per update.*

Proof. Let $U_i(n)$ denote the amortized update time for a data structure of type i for the pair (A, B) where $n = |A| + |B|$. It follows from Theorem 1 that $U_0(n) = O(\log n)$, even in the worst case.

Let $1 \leq i \leq k$. If we do not count the rebuilding costs that are caused by the rotations at the nodes on the search path, then we spend an amount of time that is bounded by $O(U_{i-1}(n) \log n)$.

Let v be a node that causes a rotation and let m be the number of points that are stored in the subtree rooted at v. During the rotation we rebuild a constant number of structures of type $(i-1)$. For $i=1$, a structure of type 0 can be rebuilt in $O(m)$ time by using Lemma 3. By this observation, and by Lemma 5, we conclude that the constant number of rebuildings—due to the rotation at v—can be done in $O(m(\log m)^{i-1})$ time.

Note that during one update there may be many nodes that cause rotations. In Mehlhorn [6, page 198], however, it is shown that if a single rotation takes $O(m(\log m)^{i-1})$ time, then the amortized cost for all rebalancing operations in one single update is bounded by $O((\log n)^i)$.

Thus, the update time satisfies the following recurrence relation:

$$U_i(n) = O(U_{i-1}(n) \log n) + O((\log n)^i).$$

This recurrence solves to $U_i(n) = O((\log n)^{i+1})$ which proves the lemma. □

This concludes the description of the data structures and algorithms. We summarize the results of this section in the following theorem.

Theorem 2. *Let i be an integer such that $1 \le i \le k$. Let A and B be sets of points in k-space. Assume that the points of A and B have coordinates of opposite sign in a fixed set of $k-i$ positions. Let $a = |A|$ and $b = |B|$ where $a \ge 0$, $b \ge 0$, and $n = a + b > 0$.*

The data structure of type i for the pair (A, B) maintains a variable $\delta_i(A, B)$ that satisfies requirements (4), (5), and (6). The structure has size $O(n(\log n)^i)$ and can be built in $O(n(\log n)^i)$ time. In this structure points can be inserted and deleted at a cost of $O((\log n)^{i+1})$ amortized time per update.

Now let V be a set of n points in k-dimensional space and consider the data structure of type k for the pair (A, B) where $A = B = V$. This data structure maintains a variable $\delta := \delta_k(V, V)$ such that (6) holds. Note that $d(A, B) = d(A \cup B, A \cup B)$. Hence, $\delta = d(A, B) = d(V, V)$. It follows that this data structure maintains the minimal distance in the set V. We have proved the following theorem.

Theorem 3. *There exists a data structure that maintains the minimal L_t-distance of a set of n points in k-dimensional space, at a cost of $O((\log n)^{k+1})$ amortized time per update. The data structure has size $O(n(\log n)^k)$ and can be built in $O(n(\log n)^k)$ time.*

4. Applying Dynamic Fractional Cascading

Now we show how the update algorithm can be improved using dynamic fractional cascading. We assume that the reader is familiar with this data structuring technique. See Chazelle and Guibas [3] and Mehlhorn and Näher [7].

Consider a data structure of type 1. It consists of a BB[α]-tree T in which the nodes contain structures of type 0. Assume that an update occurs in set A. During this update we walk down the tree, and in each node u we visit we update one data structure of type 0. In each such update we do basically the same: Suppose that the walk proceeds to the son v of u. Then we insert or delete the same point in the BB[α]-trees $T^k_{A_v}$, $T^{r_u,\infty}_{A_v,k-1}$, $T_{A'_v}$, and $T_{A_v\backslash A'_v}$. Here $r_u = (0, \ldots, 0, \sigma_u)$ is the reference point of the type 0 structure stored at node u. Note that by Lemma 3 the third tree is a "subsequence" of the first one (or its inverse), and that the fourth is a "subsequence" of the second. Furthermore, the trees $T^k_{A_t}$, if we vary u, are all sorted with respect to the same ordering. The same holds for all trees $T^{r_u,\infty}_{A_t,k-1}$.

These observations make it possible to use fractional cascading in the data structure of type 1. More precisely, we change the data structure of type 1 as follows:

1. Let u range over the nodes of T and let v be any son of u. We apply fractional cascading to the trees $T^k_{A_t}$. That is, instead of these trees, we store augmented catalogues in the nodes u of the BB[α]-tree T, and we store bridges between the augmented catalogues of adjacent nodes, as described in [7]. Similarly, we apply fractional cascading to the trees $T^k_{B_t}$.
2. In the same way, we apply fractional cascading to the trees $T^{r_u,\infty}_{A_t,k-1}$ where u ranges over the nodes of T. We do the same for the trees $T^{r_u,\infty}_{B_t,k-1}$.
3. Let u be a node in T and let v be any of its sons. We link the trees $T_{A'_v}$ and $T^k_{A_v}$ as follows: For each $p \in A'_v$ we put a pointer from p in $T^k_{A_v}$ to p in $T_{A'_t}$. We do the same for B.
4. Let u be a node in T and let v be any of its sons. We link the trees $T_{A_v\backslash A'_v}$ and $T^{r_u,\infty}_{A_v,k-1}$: For each $p \in A_v\backslash A'_v$ we put a pointer from p in $T^{r_u,\infty}_{A_v,k-1}$ to p in $T_{A_v\backslash A'_v}$. We do the same for B.

The update algorithm for the data structure of type i does not change if $i \geq 2$. For $i = 1$, however, it is replaced by the following one.

The Update Algorithm for the Data Structure of Type 1

Suppose we want to insert or delete point p in a data structure of type 1 for the pair (A, B). Assume without loss of generality that the update occurs in set A.

First, we search with the kth coordinate of p in the BB[α]-tree T to find the leaf x where the update takes place. This gives a path in T.

Let u be the root of T and let v be the son of u that is on the path. Then we do binary searches in the trees $T^k_{A_v}$ and $T^{r_u,\infty}_{A_v,k-1}$ and we insert or delete p in these trees. If $p \in A'_v$, we follow the pointer from $T^k_{A_v}$ to $T_{A'_v}$ and insert or delete p in this tree. Otherwise, if $p \in A_v\backslash A'_v$ we follow the pointer from $T^{r_u,\infty}_{A_v,k-1}$ to $T_{A_v\backslash A'_v}$ and perform the update there. Then we recompute the value of $\delta_0(A_v, B_w)$ where w is the other son of u.

In the other nodes u on the search path we update the binary search trees $T^k_{A_v}$ and $T^{r_u,\infty}_{A_v,k-1}$, by following bridges between the corresponding trees that are stored with the father of u. The trees $T_{A'_v}$ and $T_{A_v\backslash A'_v}$ are updated as above. Afterwards, we recompute the value of $\delta_0(A_v, B_w)$ where w is the other son of u.

If we have reached the leaf x, then we insert or delete p as usual. Then we walk back to the root and we recompute the δ_1-values by setting

$$\delta_1(A_u, B_u) := \min(\delta_1(A_v, B_v), \delta_1(A_w, B_w), \delta_0(A_v, B_w), \delta_0(A_w, B_v))$$

at each node u.

The rebalancing algorithm is similar as before. That is, we walk along the path to the root and rebalance the BB[α]-tree by means of rotations. If a rotation is done at a node, we rebuild a constant number of type 0 data structures. The algorithm for rebuilding such structures uses Lemma 3. Note that this rebalancing algorithm is basically the same as that for segment trees in [7]. Therefore, the reader is referred to that paper for the details. The reader can also find there the details how the fractional cascading information is maintained.

In the data structure of type i, where $i \geq 2$, we replace the substructures of type 1 by structures that use dynamic fractional cascading. The complexity of the resulting data structure—and the main result of this paper—is given in the following theorem.

Theorem 4. *There exists a data structure that maintains the minimal L_t-distance of a set of n points in k-dimensional space, at a cost of $O((\log n)^k \log\log n)$ amortized time per update. The data structure has size $O(n(\log n)^k)$ and can be built in $O(n(\log n)^k \log\log n)$ time.*

Proof. In [7] it is shown that dynamic fractional cascading does not increase the space complexity and that it increases the building–ime by a $O(\log\log n)$ factor.

To prove the bound on the update time we analyze the update algorithm for the data structure of type 1. In the root of T we spend $O(\log n)$ time to update the appropriate binary search trees. Once we have located p in the binary trees that are stored with a node u, we need $O(\log\log n)$ time to locate p in the binary search trees that are stored with its son v. (Using the bridges, we can locate p in an augmented catalogue in constant time. Then we use a UNION–SPLIT–FIND structure to locate p in the actual catalogue. This takes $O(\log\log n)$ time. See [7].)

It follows that the time for an update—if we do not count the rebalancing costs—is bounded by $O(\log n \log\log n)$.

If a node v causes a rotation, then we rebuild a constant number of structures of types 0. Let m be the number of points that are stored in the subtree of v. In the algorithm without fractional cascading we needed $O(m)$ time to rebuild these structures. In [7] it is shown that dynamic fractional cascading adds only a log log n factor to this rebuilding time. Hence, a rotation takes $O(m \log\log n)$ time. Then it follows—using the result in Mehlhorn [6, page 198]—that the amortized cost for all rebalancing operations in one single update is bounded by $O(\log n \log\log n)$.

Hence, the amortized update time for the data structure of type 1 is bounded by $O(\log n \log\log n)$.

Let $U_i(n)$ denote the amortized update time for the data structure of type i. Then, in the same way as in the proof of Lemma 6, we obtain the following recurrence relation: $U_1(n) = O(\log n \log\log n)$ and

$$U_i(n) = O(U_{i-1}(n) \log n) + O((\log n)^i \log\log n)$$

for $i \geq 2$. This recurrence solves to $U_i(n) = O((\log n)^i \log\log n)$. This completes the proof, because $U_k(n)$ is the amortized update time for the complete data structure. □

5. Concluding Remarks

We have given a data structure that maintains the minimal L_t-distance of a set of points in polylogarithmic time when arbitrary updates are performed. This is the first structure that achieves a polylogarithmic update time. In the k-dimensional case, the structure has size $O(n(\log n)^k)$ and an update takes $O((\log n)^k \log\log n)$ amortized time.

The best linear size data structure that is known at present is based on results in [4], [11] and [13]. This structure maintains the minimal L_t-distance in a k-dimensional point set in $O(n^{1/2} \log n)$ time, even in the worst case.

The basic open problem is, of course, to improve the above results. In particular, it would be interesting to have a data structure of linear size that maintains the minimal distance in polylogarithmic time.

Another open problem is to investigate whether the technique of this paper can be applied to related problems where the maximum or minimum of a two-variable function has to be maintained when objects are inserted and deleted. (See [5] and [12] for a general approach to such problems for a special type of updates.)

Acknowledgment

The author would like to thank an anonymous referee for the many suggestions that made it possible to improve the update time in the first version of this paper.

References

1. A. Aggarwal, L. J. Guibas, J. Saxe, and P. W. Shor, A linear-time algorithm for computing the Voronoi diagram of a convex polygon, *Discrete Comput. Geom.* **4** (1989), 591–604.
2. N. Blum and K. Mehlhorn, On the average number of rebalancing operations in weight-balanced trees, *Theoret. Comput. Sci.* **11** (1980), 303–320.
3. B. Chazelle and L. J. Guibas, Fractional cascading I: A data structuring technique, *Algorithmica* **1** (1986), 133–162.
4. M. T. Dickerson and R. S. Drysdale, Enumerating k distances for n points in the plane, *Proc. 7th ACM Symp. on Comp. Geom.*, 1991 (to appear).
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. K. Mehlhorn, *Data Structures and Algorithms*, Volume 1: *Sorting and Searching*, Springer-Verlag, Berlin, 1984.
7. K. Mehlhorn and S. Näher, Dynamic fractional cascading, *Algorithmica* **5** (1990), 215–241.
8. M. H. Overmars. Dynamization of order decomposable set problems. *J. Algorithms* **2** (1981), 245–260.
9. M. H. Overmars, *The Design of Dynamic Data Structures*, Lecture Notes in Computer Science, Vol. 156, Springer-Verlag, Berlin, 1983.
10. F. P. Preparata and M. I. Shamos, *Computational Geometry, an Introduction*, Springer-Verlag, New York, 1985.
11. J. S. Salowe, Shallow interdistance selection and interdistance enumeration, Manuscript, 1991.
12. M. Smid, A worst-case algorithm for semi-online updates on decomposable problems, Report A 03/90, Fachbereich Informatik, Universität des Saarlandes, 1990.
13. M. Smid, Maintaining the minimal distance of a point set in less than linear time, Report A 06/90, Fachbereich Informatik, Universität des Saarlandes, 1990.
14. 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.
15. P. M. Vaidya, An $O(n \log n)$ algorithm for the all-nearest-neighbors problem, *Discrete Comput. Geom.* **4** (1989), 101–115.

Received July 10, 1990, and in revised form April 24, 1991.