Maintaining the minimal distance of a point set in polylogarithmic time *

Michiel Smid

A 13/90

Fachbereich Informatik, Universität des Saarlandes
D-6600 Saarbrücken, West-Germany

June 1990

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

Maintaining the minimal distance of a point set in polylogarithmic time*

Michiel Smid
Fachbereich Informatik
Universität des Saarlandes
D-6600 Saarbrücken
West-Germany

June 28, 1990

Abstract

A dynamic data structure is given that maintains the minimal distance of a set of n points in k-dimensional space in $O((\log n)^{k+2})$ amortized time per update. The size of the data structure is bounded by $O(n(\log n)^k)$. Distances are measured in an arbitrary 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, when arbitrary updates are performed.

1 Introduction

One of the fundamental type of problems in computational geometry are proximity problems, where we are given a set of e.g. points and we want to compute the minimal distance among these points, or we want for each point its nearest neighbor, etc. 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 [7], Vaidya [10].)

The L_t-metric is defined for t such that $1 \leq t \leq \infty$. In this metric, 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

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

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

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

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 [3] considered the case when the updates are *semi-online*. A sequence of updates is called semi-online, 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-online update. This result was made worst-case by the author in [8].

For arbitrary updates on the minimal euclidean distance of a set of planar points, the first non-trivial result was by Overmars [5,6], who gave an $O(n)$ time update algorithm. His method uses $O(n \log\log n)$ space. Aggarwal, Guibas, Saxe and Shor [1] showed that in a 2-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 [9], the author gives an 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.

In this paper, we give a data structure of size $O(n(\log n)^k)$, that maintains the minimal distance in $O((\log n)^{k+2})$ amortized time per update. The data structure is composed of a number of structures that solve similar—but simpler—problems. More precisely, we first define so-called structures of type i that estimate the distance between two sets A and B, the points of which have coordinates of opposite sign in a fixed set of $k - i$ positions. These structures are defined recursively for $i = 0, 1, \ldots, k-1$. Then, these structures are used to define the final data structure.

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

In Section 2, we show how a k-dimensional cube can be found that contains a prescribed number of points. This algorithm is necessary to build the structure of type 0, that is defined in Section 3. The structure of type 0 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. In Section 4, 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. In Section 5, we give the general data structure for maintaining the minimal distance of a point set. The structure is defined recursively, and it uses the structure of type $(k-1)$ as a building block. We finish the paper in Section 6 with some concluding remarks.

2 Some preliminary results

To initialize the data structure of type 0, we need a k-dimensional cube, having its "bottom-left" corner at the origin, that contains a prescribed number of points. In this section, we give an algorithm to find such a k-cube.

Let V be a set of n points in k-dimensional space, and let m be an integer, such that $1 \leq m \leq n$. We assume that all coordinates of the points in V are non-negative.

We give an algorithm that finds the smallest axes-parallel k-dimensional cube $[0:s] \times \ldots \times [0:s]$, that contains at least m points of V. Points that are on the boundary of a k-cube, are assumed to be contained in this cube.

Define the following total ordering on k-dimensional points. For points $p = (p_1, \ldots, p_k)$ and $q = (q_1, \ldots, q_k)$,

$$p \preceq q \quad \text{iff} \quad \max(p_1, \ldots, p_k) \leq \min(q_1, \ldots, q_k).$$

Hence, $p \preceq q$, iff there is an axes-parallel k-cube $[0:s] \times \ldots \times [0:s]$, that contains p on its boundary, and if it contains q, then q is also at the boundary. This ordering is transitive.

Given the set V, apply the linear time algorithm of Blum et al. [2] to find an m-th point $p = (p_1, \ldots, p_k)$ of V, according to this ordering. More precisely, find a point $p \in V$, such that there are at least $m-1$ points $q \in V$ for which $q \preceq p$, and there are at least $n-m$ points $r \in V$ for which $p \preceq r$. Then the k-cube $[0:s] \times \ldots \times [0:s]$, where $s = \max(p_1, \ldots, p_k)$, is the cube we want. Now walk along the points of V, and for each point q, check whether $q \preceq p$. If it does, it is contained in the cube.

Lemma 1 *Let V be a set of n points in k-space, having non-negative coordinates, and let m be an integer, such that $1 \leq m \leq n$. In $O(n)$ time, using $O(n)$ space, we can find the smallest axes-parallel k-cube $[0:s] \times \ldots \times [0:s]$, that contains at least m points of V. In the same amount of time and space, we can find the m points of V that are contained in this cube.*

Proof: The complexity bounds are clear. So we only have to prove that the algorithm finds the smallest k-cube that contains at least m points of V. Let S

be the cube that is found by the algorithm. Suppose there is a smaller cube $[0 : s'] \times \ldots \times [0 : s']$ containing at least m points. Since the point p that is selected by the algorithm is on the boundary of the cube S, this point is neither contained nor on the boundary of this smaller cube. Hence, there are at least m points $q \in V$ for which $q \preceq p$ and $p \not\preceq q$. Therefore, p does not have rank m in the ordering $\preceq$. This is a contradiction. □

The following lemma shows that if we have a k-cube containing m points, then we have an upper bound on the minimal distance, during a sequence of $O(m)$ updates.

Lemma 2 *Let V be a set of n points in k-space, and let m be an integer, such that $1 \leq m \leq n$. Suppose we have a k-dimensional cube with side-lengths s, that contains at least m points of V. Then, during a sequence of $\lceil m/(k+1)^k \rceil - 2$ insertions and deletions of points in V, the minimal L_t-distance between points in V is less than s.*

Proof: Assume w.l.o.g. that the k-cube is equal to $[0 : s] \times \ldots \times [0 : s]$. Partition this cube into $(k+1)^k$ subcubes

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

where the i_j are integers such that $0 \leq i_j \leq k$ and $1 \leq j \leq k$. At least one such subcube contains $\lceil m/(k+1)^k \rceil$ points of V. During a sequence of $\lceil m/(k+1)^k \rceil - 2$ updates, this subcube contains at least two points. Hence, during this sequence, 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. □

3 The type 0 data structure

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.

Let A and B be two sets of points in k-space. We assume that the points of A and the points of B lie in opposite k-dimensional quadrants. Therefore, we may assume w.l.o.g. that all coordinates of the points in A are non-positive, and all coordinates of the points in B are non-negative. (If the origin 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 having all their coordinates non-positive, or having all their coordinates non-negative, are inserted and deleted. In the first case, 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 between the points 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. All coordinates of the points in A resp. B are non-positive resp. non-negative. The cardinality of A resp. B is denoted by a resp. b. It may be possible that $a = 0$ or $b = 0$. However, $a + b > 0$.

We fix $1 \leq t \leq \infty$, and we measure all distances in the L_t-metric. The minimal distance between points in A and points in B, is denoted by $d(A, B)$, whereas $d(A \cup B, A \cup B)$ denotes the minimal distance between all points in $A \cup B$. (We define $d(A, \emptyset) := d(\emptyset, B) := \infty$.)

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

$$\delta \geq d(A, B), \tag{1}$$

$$\text{if } d(A, B) = d(A \cup B, A \cup B), \text{ then } \delta = d(A, B). \tag{2}$$

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.

Throughout this paper, we fix a constant C that is sufficiently large.

The data structure of type 0: If $a + b \leq C$, then we store the set A resp. B in a search tree T_A resp. T_B. Furthermore, there is a variable δ, that is equal to $d(A, B)$.

Suppose that $a + b > C$. Assume that $a \leq b$. (Otherwise, interchange A and B.) Consider the smallest axes-parallel k-cube $C_+ := [0 : s] \times \ldots \times [0 : s]$ that contains at least $\lceil b/2 \rceil$ points of B. Let C_- be the k-cube $[-s : 0] \times \ldots \times [-s : 0]$. Let A' resp. B' be the set of those points in A resp. B that are in the interior of the cube C_- resp. C_+. (Note that A and/or B may be empty.)

The data structure of type 0 for the pair (A, B) consists of the following. There are search trees T_A and T_B, storing the sets A and B. Furthermore, there is a pointer to a recursively defined structure of type 0 for the pair (A', B'). Let δ' be the variable that is maintained by this structure. Then the value of δ corresponding to the structure for the pair (A, B) is equal to δ'.

Of course, we have to prove that the value of δ satisfies (1) and (2). We will prove this in Lemma 4, after we have given the update algorithm.

Building the structure of type 0: If $a + b \leq C$, then store A and B in search

trees T_A and T_B, and compute $\delta := d(A, B)$.

Suppose that $a + b > C$. Assume w.l.o.g. that $a \leq b$. Use the algorithm of Section 2 to compute the k-cube C_+ and the set B'. Given the side-lengths of C_+, compute the set A'. Then build the structure of type 0 for the pair (A', B'), using the same algorithm recursively.

This gives search trees $T_{A'}$ and $T_{B'}$ for the sets A' and B', and a variable δ'. Copy these search trees, and insert the points of $A \setminus A'$ in the copy of $T_{A'}$, and the points oF $B \setminus B'$ in the copy of $T_{B'}$. This gives two trees T_A and T_B containing the sets A and B. Finally, set $\delta := \delta'$.

In the next lemma, we give an upper bound on the building time for the structure of type 0. In this lemma, and in the following ones, we give complexity bounds of the form $O(f(a + b))$ for some function f. These asymptotic bounds are valid for any pair of integers $a \geq 0$ and $b \geq 0$, such that $a + b$ is sufficiently large.

Lemma 3 *The data structure of type 0 for the pair (A, B) can be built in time $O((a + b) \log(a + b))$.*

Proof: Let $T(a, b)$ denote the building time for sets A and B of sizes a and b. We do not count here the time needed to insert the points of $A \setminus A'$ and $B \setminus B'$ in the copies of the search trees. If $a + b \leq C$, then clearly $T(a, b)$ is bounded by a constant. So, suppose that $a + b > C$. Assume w.l.o.g. that $a \leq b$. By Lemma 1, it takes $O(b)$ time to find the k-cube C_+ and the set B'. Note that we can compute simultaneously the set $B \setminus B'$ in $O(b)$ time. Once the side-length of the cube C_+ is known, the sets A' and $A \setminus A'$ can be computed in $O(a)$ time.

Suppose that $|B'| \geq \lceil b/2 \rceil$. Then—since all points in B' are in the interior of C_+—there exists a smaller cube $[0 : s'] \times \ldots \times [0 : s']$ containing at least $\lceil b/2 \rceil$ points of B. This is a contradiction, because C_+ is the smallest such cube. Therefore, $|B'| \leq \lceil b/2 \rceil$.

Since $|A'| \leq a$ and $|B'| \leq \lceil b/2 \rceil$, it takes at most $T(a, \lceil b/2 \rceil)$ time to build the structure of type 0 for the pair (A', B'). Copying the two search tree takes $O(a + b)$ time.

Hence, $T(a, b) = O(a + b) + T(a, \lceil b/2 \rceil)$ if $a \leq b$. If $a \geq b$, then the same relation holds with a and b interchanged. It follows that $T(a, b) = O((a + b) \log(a + b))$.

Since each point of A and B is inserted exactly once in a search tree, the total time for completing the search trees for all (sub)structures of type 0 is bounded by $O((a + b) \log(a + b))$. (The time for copying the search trees was counted already.)

We have shown that the entire algorithm spends $O((a + b) \log(a + b))$ time. □

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:

1. If $a + b \leq C$, update the search tree T_A or T_B that stores the set A or B in which the update occurs, and recompute the value of $\delta := d(A, B)$.

2. Suppose that $a + b > C$. Assume w.l.o.g. that the the update occurs in the set B. (Otherwise, interchange A and B.)

   (a) If the update is a deletion of point p, and if $|A| + |B \setminus \{p\}| \leq C$, then delete p from the tree T_B, and compute the value of $\delta := d(A, B \setminus \{p\})$. Discard the structure of type 0 for the pair (A', B').

   (b) Otherwise, first update the search tree T_B. If $p \notin C_+$, set $\delta := \delta'$, where δ' is the variable that is maintained by the structure of type 0 for the pair (A', B'). If $p \in C_+$, update the structure for the pair (A', B') recursively. This structure maintains a variable δ'. Then, set $\delta := \delta'$.

3. After $(a + b)/(4\,(k+1)^k) - 2$ updates are performed in this way, we start over again. That is, we completely rebuild the structure of type 0 for the pair (A, B), and we continue performing updates as described above.

Lemma 4 *At any time, the value of δ satisfies (1) and (2).*

Proof: If $a + b \leq C$, then the value of δ is equal to $d(A, B)$. It is clear that δ satisfies the requirements of (1) and (2).

So assume that $a + b > C$. Furthermore, assume inductively that the value of the variable δ' that is maintained by the structure for the pair (A', B') is correct.

After the structure for the pair (A, B) is built, we have $\delta = \delta'$. It follows from the update algorithm that at any moment the values of δ and δ' are equal. (This is also true if the structure for the pair (A', B') is rebuilt.) By the induction hypothesis, we have

$$\delta' \geq d(A', B'), \tag{3}$$

$$\text{if } d(A', B') = d(A' \cup B', A' \cup B'), \text{ then } \delta' = d(A', B'). \tag{4}$$

Since $A' \subseteq A$ and $B' \subseteq B$, we have $d(A', B') \geq d(A, B)$. Therefore, it follows from (3), and from the fact that $\delta = \delta'$, that $\delta \geq d(A, B)$. Hence, δ satisfies (1).

Now suppose that $d(A, B) = d(A \cup B, A \cup B)$. By Lemma 2, during a sequence of $\lceil m/(k+1)^k \rceil - 2$ updates, the value of $d(A, A)$ or $d(B, B)$ is less than s, where $m = \lceil \max(a_0, b_0)/2 \rceil$, and a_0 and b_0 are the sizes of A and B at the moment the structure for the pair (A, B) is built. (Here, $d(B, B) < s$ if $a_0 \leq b_0$, and $d(A, A) < s$ if $b_0 \leq a_0$. In case $a_0 \leq b_0$, Lemma 2 is applied to the points of B that are in the interior or on the boundary of the cube C_+.) Since

$$\lceil m/(k+1)^k \rceil - 2 \geq \frac{\max(a_0, b_0)}{2\,(k+1)^k} - 2 \geq \frac{a_0 + b_0}{4\,(k+1)^k} - 2,$$

it follows that during a sequence of $(a_0 + b_0)/(4\,(k+1)^k) - 2$ updates, the value of $d(A, A)$ or $d(B, B)$ is less than s. Clearly, at the start of the algorithm, or after a rebuilding operation, also at least one of $d(A, A)$ and $d(B, B)$ is at most s. Hence, at any moment, $d(A \cup B, A \cup B) < s$. But then, by our assumption, we have

$d(A,B) < s$. Since points in A and $B \setminus B'$ are "separated" by the k-cube C_+, we have $d(A \setminus A', B) \geq s$. Similarly, $d(A, B \setminus B') \geq s$. Therefore,

$$d(A,B) = \min\left(d(A',B'), d(A \setminus A', B), d(A, B \setminus B')\right) = d(A',B').$$

Hence, $d(A',B') = d(A \cup B, A \cup B)$. It is clear that $d(A \cup B, A \cup B) \leq d(A' \cup B', A' \cup B') \leq d(A',B')$. Hence, $d(A',B') = d(A' \cup B', A' \cup B')$. Then it follows from (4) that $\delta' = d(A',B')$. Since $\delta = \delta'$, and $d(A,B) = d(A',B')$, we have $\delta = d(A,B)$. This proves that δ satisfies the requirement of (2). □

Lemma 5 *In the data structure of type 0 for the pair (A,B), points can be inserted and deleted, at a cost of $O((\log(a+b))^2)$ amortized time per update.*

Proof: Let $U(a,b)$ denote the amortized update time for sets A and B of sizes a and b. If $a + b \leq C$, then $U(a,b)$ is bounded by a constant. The same holds if $a + b - 1 \leq C$ and the update is a deletion.

Assume that $a + b > C$. Here, a and b are the current sizes of the sets A and B. Let a_0 resp. b_0 be the size of the set A resp. B at the start of the algorithm, i.e., at the moment the structure for the pair (A,B) is built. Assume w.l.o.g. that $a_0 \leq b_0$. During a sequence of $(a_0 + b_0)/(4\,(k+1)^k) - 2$ updates, we have $|A'| \leq |A| = a$, and

$$b = |B| \geq b_0 - \frac{a_0 + b_0}{4\,(k+1)^k} + 2 \geq b_0 - \frac{2b_0}{4\,(k+1)^k} \geq b_0 - \frac{2b_0}{16} = \frac{7}{8} b_0.$$

We saw already in the proof of Lemma 3 that initially $|B'| \leq \lceil b_0/2 \rceil \leq 1 + b_0/2$. Therefore, during the sequence of $(a_0 + b_0)/(4\,(k+1)^k) - 2$ updates:

$$|B'| \leq 1 + \frac{b_0}{2} + \frac{a_0 + b_0}{4\,(k+1)^k} - 2 \leq \frac{b_0}{2} + \frac{2b_0}{4\,(k+1)^k} \leq \frac{b_0}{2} + \frac{2b_0}{16} = \frac{5}{8} b_0.$$

It follows that at any moment

$$|B'| \leq \frac{5}{8} b_0 \leq \frac{5}{8}\frac{8}{7} b = \frac{5}{7} b.$$

If in the update the data structure for the pair (A,B) is not rebuilt, we spend an amount of time that is bounded by

$$O(\log(a+b)) + U(|A'|, |B'|) \leq O(\log(a+b)) + U(a, 5b/7),$$

where the logarithmic factor is the time needed to update the appropriate search tree T_A or T_B. After $(a_0 + b_0)/(4\,(k+1)^k) - 2$ updates, we rebuild the structure. By Lemma 3, this takes $O((a+b)\log(a+b))$ time. It follows that the amortized update time $U(a,b)$ satisfies the following recurrence:

$$U(a,b) = O(\log(a+b)) + U(a, 5b/7) + \frac{O((a+b)\log(a+b))}{(a_0 + b_0)/(4\,(k+1)^k) - 2}.$$

Since $a + b = \Theta(a_0 + b_0)$, this is equivalent to

$$U(a,b) = O(\log(a+b)) + U(a, 5b/7), \tag{5}$$

if initially $a_0 \leq b_0$. If $a_0 \geq b_0$, then the above relation holds with a and b interchanged.

We prove that $U(a,b) = O((\log(a+b))^2)$. Let α be the constant in the logarithmic factor in (5). Choose a constant β, such that $U(a,b) \leq \beta(\log(a+b))^2$ for all a and b for which $a + b \leq C$, and such that $\beta \log 8/7 \geq \alpha$. Let $a + b > C$, and assume that $U(a',b') \leq \beta(\log(a'+b'))^2$ for all a' and b' for which $a' + b' < a + b$. Assume w.l.o.g. that initially $a_0 \leq b_0$. Then

$$\begin{aligned} U(a,b) &\leq \alpha \log(a+b) + U(a, 5b/7) \\ &\leq \alpha \log(a+b) + \beta(\log(a + 5b/7))^2 \\ &\leq \alpha \log(a+b) + \beta \log(a+b) \log(a + 5b/7) \\ &\leq \beta(\log(a+b))^2, \end{aligned}$$

provided we can prove that $\alpha + \beta \log(a + 5b/7) \leq \beta \log(a+b)$, or, equivalently,

$$\beta \log\left(\frac{a+b}{a+5b/7}\right) \geq \alpha. \tag{6}$$

We have assumed that $a_0 \leq b_0$. During a sequence of $(a_0 + b_0)/(4\,(k+1)^k) - 2$ updates, we have

$$a \leq a_0 + \frac{a_0 + b_0}{4\,(k+1)^k} - 2 \leq b_0 + \frac{2b_0}{4\,(k+1)^k} - 2 \leq b_0 + \frac{2b_0}{16} = \frac{9}{8} b_0.$$

We saw already that $b \geq 7b_0/8$. Hence,

$$a \leq \frac{9}{8} b_0 \leq \frac{9}{8}\frac{8}{7} b = \frac{9}{7} b. \tag{7}$$

It follows that

$$\frac{a+b}{a+5b/7} = 1 + \frac{2b/7}{a+5b/7} \geq 1 + \frac{2b/7}{9b/7 + 5b/7} = \frac{8}{7},$$

and therefore

$$\beta \log\left(\frac{a+b}{a+5b/7}\right) \geq \beta \log \frac{8}{7}.$$

By our choice of β, we have $\beta \log 8/7 \geq \alpha$, which proves (6).

We have shown that $U(a,b) \leq \beta(\log(a+b))^2$ for all a and b. This proves the lemma. □

Lemma 6 *The size of the data structure of type 0 for the pair (A,B) is bounded by $O(a+b)$.*

Proof: Let $S(a,b)$ be the size of the data structure for sets A and B of sizes a and b. If $a+b \leq C$ then $S(a,b)$ is bounded by a constant. Otherwise, if $a+b > C$,

$$S(a,b) \leq O(a+b) + S(a, 5b/7),$$

if initially the size of A is less than that of B. If initially $|A| \geq |B|$, then the same recurrence holds with a and b interchanged. Using the fact that during a sequence of $(a_0+b_0)/(4\,(k+1)^k) - 2$ updates we always have $a \leq 9b/7$, if initially $|A| \leq |B|$ (see (7)), it can be shown by induction that $S(a,b) = O(a+b)$. □

4 The type i data structure

In this section, we recursively define the structure of type i, that maintains the "distance" between two point sets, the points of which have coordinates of opposite sign in a set of $k-i$ positions. The variable that is maintained by this structure satisfies the same requirements as in (1) and (2). (So the structure does not always give the minimal distance. That is why "distance" was between brackets.) The structure uses the structure of type $(i-1)$ as a building block.

Let i be an integer, such that $0 \leq i \leq k-1$. Let A and B be two sets of points in k-dimensional space. 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 w.l.o.g. that the points of A lie in the space $\mathcal{R}_i^- := (-\infty : 0]^{k-i} \times \mathcal{R}^i$, and the points of B lie in $\mathcal{R}_i^+ := [0 : \infty)^{k-i} \times \mathcal{R}^i$. Hence, the regions in which A and B lie are separated by an i-dimensional space.

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 $\mathcal{R}_i^- \cup \mathcal{R}_i^+$ are inserted and deleted. If a point in $\mathcal{R}_i^-$ is inserted or deleted, then the update "occurs" in set A. Otherwise, the update "occurs" in set B.

The cardinality of A resp. B is denoted by a resp. b. It may be possible that $a=0$ or $b=0$, but, $a+b>0$.

Again, we fix $1 \leq t \leq \infty$, and we measure all distances in the L_t-metric. The minimal distance between points in A and points in B, is denoted by $d(A,B)$, whereas $d(A \cup B, A \cup B)$ denotes the minimal distance between all points in $A \cup B$. (Note that $d(A,\emptyset) = d(\emptyset,B) = \infty$.)

We give a dynamic data structure of type i, that maintains a variable $\delta \in \mathcal{R} \cup \{\infty\}$, such that

$$\delta \geq d(A,B), \tag{8}$$

$$\text{if } d(A,B) = d(A \cup B, A \cup B), \text{ then } \delta = d(A,B). \tag{9}$$

As before, we say that the structure stores the pair (A,B).

In the rest of this section, we use the same constant C as in the previous section. This constant is assumed to be sufficiently large.

The data structure of type i is defined inductively. The structure of type 0 was defined in the previous section. So let $0 < i \leq k-1$, and assume that the structure of type $(i-1)$ is defined already.

The data structure of type i: If $a+b \leq C$, then we store A and B in search trees T_A and T_B. Furthermore, there is a variable δ, that is equal to $d(A,B)$.

Suppose that $a+b > C$. Assume that $a \leq b$. (Otherwise, interchange A and B.) Split the set B in two subsets B_1 and B_2, of size $\lceil b/2 \rceil$ resp. $\lfloor b/2 \rfloor$, such that all $(k-i+1)$-th coordinates of the points in B_1 are at most equal to those of the points in B_2. The sets B_1 and B_2 are separated by some $(k-1)$-dimensional hyperplane $x_{k-i+1} = l$. Split the set A into sets A_1 and A_2, where all points in A_1 have an $(k-i+1)$-th coordinate which is less than l, and all points in A_2 have an $(k-i+1)$-th coordinate which is at least l.

Note that $A_1 \subseteq (-\infty : 0]^{k-i} \times (-\infty : l] \times \mathcal{R}^{i-1}$, $A_2 \subseteq (-\infty : 0]^{k-i} \times [l : \infty) \times \mathcal{R}^{i-1}$, $B_1 \subseteq [0 : \infty)^{k-i} \times (-\infty : l] \times \mathcal{R}^{i-1}$, $B_2 \subseteq [0 : \infty)^{k-i} \times [l : \infty) \times \mathcal{R}^{i-1}$. Therefore, the points in the sets A_1 and B_2 have coordinates of "opposite" sign in the first $k-i+1$ positions. (In the $(k-i+1)$-th position, the coordinates have opposite sign w.r.t. their difference to l.) Similarly for the sets A_2 and B_1. Furthermore, for $u = 1, 2$, $A_u \subseteq \mathcal{R}_i^-$ and $B_u \subseteq \mathcal{R}_i^+$.

The data structure of type i for the pair (A,B) consists of the following. There are search trees T_A and T_B, storing the sets A and B. There are two pointers to structures of type $(i-1)$, one structure for the pair (A_1, B_2) and the other structure for the pair (A_2, B_1). These structures maintain variables δ_{12} resp. δ_{21}. Furthermore, there are two pointers to recursively defined structures of type i, one structure for the pair (A_1, B_1) and one structure for the pair (A_2, B_2). Let δ_{11} resp. δ_{22} be the variables that are maintained by these two structures. The value of δ corresponding to the structure of type i for the pair (A,B) is equal to the minimum of the variables δ_{11}, δ_{22}, δ_{12} and δ_{21}.

Building the structure of type i: If $a+b \leq C$, then store A and B in search trees T_A and T_B, and compute $\delta := d(A,B)$.

Suppose that $a+b > C$. Assume w.l.o.g. that $a \leq b$. First store the sets A and B in balanced search trees T_A and T_B. Next, use a linear time algorithm to find the median of the $(k-i+1)$-th coordinates of the points in B. Then partition the sets A and B according to this median into sets A_1, A_2, B_1 and B_2. Build the structures of type $(i-1)$ for the pairs (A_1, B_2) and (A_2, B_1). Let δ_{12} resp. δ_{21} be the variables that are stored with these two structures. Next, build the two structures of type i for the pairs (A_1, B_1) and (A_2, B_2), using the same algorithm recursively. Let δ_{11} and δ_{22} be the variables that are stored with these structures.

The value of δ for the structure for the pair (A,B) is set to the minimum of the variables δ_{11}, δ_{22}, δ_{12} and δ_{21}.

In the rest of this section, we shall prove the following theorem:

Theorem 1 *Let i be an integer such that $0 \leq i \leq k-1$. 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 $a \geq 0$, $b \geq 0$, and $a + b > 0$.*

The data structure of type i for the pair (A, B) maintains a variable δ that satisfies requirements (8) an (9). The structure has size $O((a+b)(\log(a+b))^i)$, and can be built in $O((a+b)(\log(a+b))^{i+1})$ time. In this structure, points can be inserted and deleted at a cost of $O((\log(a+b))^{i+2})$ amortized time per update.

This theorem will be proved by induction on i. It follows from the results in Section 3, that the theorem holds for $i = 0$. So let $1 \leq i \leq k-1$, and assume that the theorem holds for $i-1$. We shall prove that then the theorem also holds for i.

Lemma 7 *The data structure of type i for the pair (A, B) can be built in time $O((a+b)(\log(a+b))^{i+1})$.*

Proof: Let $T(a, b)$ denote the building time of the structure of type i for sets A and B of sizes a and b. If $a + b \leq C$, then $T(a, b)$ is bounded by a constant. Suppose that $a + b > C$. Assume w.l.o.g. that $a \leq b$. It takes $O(a \log a + b \log b) = O((a+b)\log(a+b))$ time to store the sets A and B in two balanced search trees. The time to partition the sets A and B into sets A_1, A_2, B_1 and B_2 is bounded by $O(a+b)$. By the induction hypothesis, the two data structures of type $(i-1)$ for the pairs (A_1, B_2) and (A_2, B_1) can be build in $O((a+b)(\log(a+b))^i)$ time.

It takes $T(|A_1|, |B_1|) + T(|A_2|, |B_2|)$ time to build the two structures of type i for the pairs (A_1, B_1) and (A_2, B_2). This expression is equal to $T(|A_1|, \lceil b/2 \rceil) + T(a - |A_1|, \lfloor b/2 \rfloor)$.

It follows that there is an a_1, $0 \leq a_1 \leq a$, such that

$$T(a, b) = O((a+b)(\log(a+b))^i) + T(a_1, \lceil b/2 \rceil) + T(a - a_1, \lfloor b/2 \rfloor),$$

if $a \leq b$. If $a \geq b$, then the same recurrence holds with a and b interchanged. It can easily be shown that $T(a, b) = O((a+b)(\log(a+b))^{i+1})$. This proves the lemma. □

The update algorithm: To insert or delete a point p in a structure of type i for the pair (A, B), we do the following:

1. If $a + b \leq C$, update the search tree T_A or T_B that stores the set A or B in which the update occurs, and recompute the value of $\delta := d(A, B)$.

2. Suppose that $a + b > C$. Assume w.l.o.g. that the update occurs in set B. (Otherwise, interchange A and B.)

   (a) If the update is a deletion of point p, and if $|A| + |B \setminus \{p\}| \leq C$, then delete p from the tree T_B, and compute the value of $\delta := d(A, B \setminus \{p\})$. Discard the structures for the pairs (A_1, B_1), (A_2, B_2), (A_1, B_2) and (A_2, B_1).

(b) Otherwise, first update the search tree T_B. If p has a $(k-i+1)$-th coordinate that is at most equal to the initial median of the $(k-i+1)$-th coordinates, then update the structure of type $(i-1)$ for the pair (A_2, B_1), and recursively update the structure of type i for the pair (A_1, B_1). Otherwise, if p has a $(k-i+1)$-th coordinate that is larger than the initial median, update the structure of type $(i-1)$ for the pair (A_1, B_2), and recursively update the structure of type i for the pair (A_2, B_2).

Afterwards, set the value of δ—corresponding to the structure for the pair (A, B)—to the minimum of the variables δ_{11}, δ_{22}, δ_{12} and δ_{21}.

3. After $(a+b)/(4\,(k+1)^k) - 2$ updates are performed in this way, we start over again. That is, we completely rebuild the structure of type i for the pair (A, B), and we continue performing updates as described above.

Lemma 8 *At any time, the value of δ satisfies (8) and (9).*

Proof: If $a + b \leq C$, then the value of δ is equal to $d(A, B)$. Clearly, δ satisfies the requirements of (8) and (9).

So assume that $a + b > C$. Furthermore, assume inductively that the values of the variables δ_{11} and δ_{22} that are maintained by the structures of type i for the pairs (A_1, B_1) and (A_2, B_2) are correct. By the induction hypothesis, the values of the variables δ_{12} and δ_{21} corresponding to the structures of type $(i-1)$ for the pairs (A_1, B_2) and (A_2, B_1) satisfy requirements (8) and (9).

After the structure for the pair (A, B) is built, the value of δ is equal to the minimum of the values of δ_{11}, δ_{22}, δ_{12} and δ_{21}. It follows from the update algorithm that at any moment the value of δ is equal to this minimum. (This is also true if one of the structures for the pairs (A_u, B_v) is rebuilt.)

Let u and v be such that $\delta = \delta_{uv}$. Then, by requirement (8), we have $\delta_{uv} \geq d(A_u, B_v)$. Since $d(A, B) \leq d(A_u, B_v)$, it follows that $d(A, B) \leq \delta_{uv} = \delta$. Hence, δ satisfies (8).

Now suppose that $d(A, B) = d(A \cup B, A \cup B)$. In order to prove requirement (9), we have to show that $\delta = d(A, B)$. We have shown already that $\delta \geq d(A, B)$. So it remains to be shown that $\delta \leq d(A, B)$. Clearly,

$$d(A, B) = \min\left(d(A_1, B_1), d(A_2, B_2), d(A_1, B_2), d(A_2, B_1)\right).$$

Let u and v be such that $d(A, B) = d(A_u, B_v)$. We have

$$d(A \cup B, A \cup B) \leq d(A_u \cup B_v, A_u \cup B_v) \leq d(A_u, B_v) = d(A, B) = d(A \cup B, A \cup B).$$

Hence, all inequalities above are in fact equalities. Therefore, $d(A_u, B_v) = d(A_u \cup B_v, A_u \cup B_v)$. It follows from (9) that $\delta_{uv} = d(A_u, B_v)$. Therefore, $\delta \leq \delta_{uv} = d(A_u, B_v) = d(A, B)$. This finishes the proof. □

Lemma 9 *In the data structure of type i for the pair (A, B), points can be inserted and deleted, at a cost of $O((\log(a+b))^{i+2})$ amortized time per update.*

Proof: Let $U(a,b)$ denote the amortized update time for a data structure of type i for the pair (A,B), where the sets A and B have sizes a and b. If $a+b \leq C$, then $U(a,b)$ is bounded by a constant. The same holds if $a+b-1 \leq C$ and the update is a deletion.

Assume that $a+b > C$. Here, a and b are the current sizes of the sets A and B. Let a_0 resp. b_0 be the size of the set A resp. B at the moment the structure of type i for the pair (A,B) is built. Assume w.l.o.g. that $a_0 \leq b_0$. In the same way as in the proof of Lemma 5, it can be shown that during a sequence of $(a_0+b_0)/(4\,(k+1)^k)-2$ updates, we have $|A_u| \leq |A| = a$, and $|B_u| \leq 5b/7$, for $u = 1,2$.

If in the update the data structure for the pair (A,B) is not rebuilt, we spend $O(\log(a+b))$ time to update the search tree that stores the set in which the update occurs. By the induction hypothesis, we spend $O((\log(a+b))^{i+1})$ time to update the appropriate structure of type $(i-1)$. Finally, we spend at most $U(a,5b/7)$ time to update the structure of type i for the pair (A_u,B_u). Here, $u=1$ if the update occurs in A_1 or B_1, and $u=2$ otherwise. It follows that in case the data structure is not rebuilt, we spend an amount of time that is bounded by

$$O\left((\log(a+b))^{i+1}\right) + U(a,5b/7).$$

After $(a_0+b_0)/(4\,(k+1)^k)-2$ updates, we rebuild the structure. By Lemma 7, this takes $O((a+b)(\log(a+b))^{i+1})$ time. Hence, the amortized update time $U(a,b)$ satisfies the following recurrence:

$$U(a,b) = O\left((\log(a+b))^{i+1}\right) + U(a,5b/7) + \frac{O((a+b)(\log(a+b))^{i+1})}{(a_0+b_0)/(4\,(k+1)^k)-2}.$$

Since $a+b = \Theta(a_0+b_0)$, this is equivalent to

$$U(a,b) = O\left((\log(a+b))^{i+1}\right) + U(a,5b/7),$$

if initially $a_0 \leq b_0$. If $a_0 \geq b_0$, then the same relation holds with a and b interchanged. From this recurrence, it can be shown in the same way as in the proof of Lemma 5, that $U(a,b) = O((\log(a+b))^{i+2})$. □

Lemma 10 *The size of the data structure of type i for the pair (A,B) is bounded by $O((a+b)(\log(a+b))^i)$.*

Proof: Let $S(a,b)$ be the size of the data structure of type i for sets A and B of sizes a and b. Assume w.l.o.g. that $|A| \leq |B|$, at the moment the data structure is built. If $a+b \leq C$, then $S(a,b)$ is bounded by a constant. Otherwise, if $a+b > C$, then it follows—using the induction hypothesis—that

$$S(a,b) = O\left((a+b)(\log(a+b))^{i-1}\right) + S(a_1,b_1) + S(a-a_1,b-b_1),$$

for some $0 \leq a_1 \leq a$ and $2b/7 \leq b_1 \leq 5b/7$. (In the same way as in the proof of Lemma 5, it follows that both B_1 and B_2 have size at most $5b/7$. Hence, $2b/7 \leq$

$|B_u| \leq 5b/7$, $u = 1, 2$, and therefore, $2b/7 \leq b_1 \leq 5b/7$.) If initially $|A| \geq |B|$, then the same relation holds with a and b interchanged. From this, it can be shown that $S(a, b) = O((a + b)(\log(a + b))^i)$. □

Lemmas 7, 8, 9 and 10 together prove that Theorem 1 holds for i. Then, by induction, the theorem holds for all i such that $0 \leq i \leq k - 1$.

5 The final data structure

Now we are ready to give the dynamic data structure for maintaining the minimal distance in a set of k-dimensional points.

Let V be a set of n points in k-space. We fix $1 \leq t \leq \infty$, and we measure all distances in the L_t-metric.

The data structure: If $n \leq C$, then we store the set V in a search tree T_V. Furthermore, there is a variable δ, that is equal to $d(V, V)$.

Suppose that $n > C$. Split the set V in two subsets V_1 and V_2, of sizes $\lceil n/2 \rceil$ resp. $\lfloor n/2 \rfloor$, such that the first coordinates of all points in V_1 are at most equal to those in V_2.

Note that $V_1 \subseteq (-\infty : l] \times \mathcal{R}^{k-1}$ and $V_2 \subseteq [l : \infty) \times \mathcal{R}^{k-1}$, for some l. Hence, the points of V_1 and V_2 have coordinates of "opposite" sign in the first position.

The data structure for the set V consists of the following. There is a balanced search tree T_V that stores the set V. There is a pointer to a structure of type $(k - 1)$ for the pair (V_1, V_2). Let δ_{12} be the variable that is maintained by this structure. Furthermore, there are two pointers to recursively defined structures, one structure for the set V_1, and the other for the set V_2. Let δ_{11} resp. δ_{22} be the variables that are maintained by these two structures. The value of δ corresponding to the structure for the set V is equal to the minimum of the variables δ_{11}, δ_{22} and δ_{12}.

We will prove later, that δ is indeed equal to the minimal distance in the set V.

Building the structure: If $n \leq C$, then store V in a search tree T_V, and compute $\delta := d(V, V)$.

Suppose that $n > C$. Use a linear time algorithm to find the median of the first coordinates of the points in V. Then partition the set V according to this median into sets V_1 and V_2. Build the structure of type $(k - 1)$ for the pair (V_1, V_2). Let δ_{12} be the variable that is stored with this structure. Next, build the two structures for the sets V_1 and V_2, using the same algorithm recursively. Let δ_{11} and δ_{22} be the variables that are stored with these structures. Then the value of δ for the structure for the set V is set to the minimum of the variables δ_{11}, δ_{22} and δ_{12}.

Lemma 11 *The data structure for the set V can be built in $O(n(\log n)^{k+1})$ time.*

Proof: Let $T(n)$ denote the building time for a set V of size n. If $n \leq C$, then $T(n)$ is bounded by a constant. So, assume that $n > C$. It takes $O(n \log n)$ time to store

the set V in a balanced search tree. The time to partition V into sets V_1 and V_2 is bounded by $O(n)$. By Theorem 1, the data structure of type $(k-1)$ for the pair (V_1, V_2) can be build in $O(n(\log n)^k)$ time. Finally, it takes $T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$ time to build the structures for the sets V_1 and V_2. It follows that

$$T(n) = O\left(n(\log n)^k\right) + T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor).$$

From this, it can easily be shown that $T(n) = O(n(\log n)^{k+1})$. □

The update algorithm: To insert or delete a point p in the data structure, we do the following:

1. If $n \leq C$, update the search tree T_V, and recompute the value of $\delta := d(V, V)$.
2. Suppose that $n > C$.
   (a) If the update is a deletion of point p, and if $|V \setminus \{p\}| \leq C$, then delete p from the tree T_V, and compute the value of $\delta := d(V \setminus \{p\}, V \setminus \{p\})$. Discard the structures for the sets V_1 and V_2, and for the pair (V_1, V_2).
   (b) Otherwise, first update the search tree T_V. Then, update the structure of type $(k-1)$ for the pair (V_1, V_2). If point p has a first coordinate that is at most equal to the initial median of the first coordinates in V, then recursively update the structure for the set V_1. Otherwise, if p has a first coordinate that is larger than this initial median, recursively update the structure for the set V_2.

   Afterwards, set the value of δ—corresponding to the structure for the set V—to the minimum of the variables δ_{11}, δ_{22}, and δ_{12}.
3. After $n/(4\,(k+1)^k) - 2$ updates are performed in this way, we start over again. That is, we completely rebuild the structure for the set V, and we continue performing updates as described above.

Lemma 12 *At any time, the value of δ is equal to the minimal distance in the set V.*

Proof: If $n \leq C$, then the value of δ is equal to $d(V, V)$. So assume that $n > C$. Furthermore, assume inductively that $\delta_{11} = d(V_1, V_1)$ and $\delta_{22} = d(V_2, V_2)$. By Theorem 1, the value of δ_{12} corresponding to the structure of type $(k-1)$ for the pair (V_1, V_2) satisfies requirements (8) and (9) given in Section 4.

After the structure for the set V is built, the value of δ is equal to the minimum of the values of δ_{11}, δ_{22} and δ_{12}. It follows from the update algorithm that at any moment the value of δ is equal to this minimum.

Clearly, $\delta_{11} = d(V_1, V_1) \geq d(V, V)$ and $\delta_{22} = d(V_2, V_2) \geq d(V, V)$. By requirement (8) of Section 4, we have $\delta_{12} \geq d(V_1, V_2) \geq d(V, V)$. It follows that $\delta = \min(\delta_{11}, \delta_{22}, \delta_{12}) \geq d(V, V)$. So it remains to be shown that $\delta \leq d(V, V)$.

Let $p, q \in V$, such that $d(p, q) = d(V, V)$. First suppose that p and q are in the same subset V_u, where $u \in \{1, 2\}$. It is clear that $d(V, V) \leq d(V_u, V_u)$. Also,

$d(V_u, V_u) \leq d(p,q)$. Therefore, we have $d(p,q) = d(V_u, V_u) = \delta_{uu}$. Since $\delta \leq \delta_{uu}$, it follows that $\delta \leq d(p,q) = d(V,V)$.

If p and q are not in the same subset, then we may assume w.l.o.g. that $p \in V_1$ and $q \in V_2$. We have $d(p,q) = d(V,V) \leq d(V_1, V_2) \leq d(p,q)$, and, hence, $d(V,V) = d(V_1, V_2)$. It follows from requirement (9) of Section 4, and from the fact that $V = V_1 \cup V_2$, that $\delta_{12} = d(V_1, V_2)$, and hence $\delta_{12} = d(V,V)$. Since $\delta \leq \delta_{12}$, it follows that $\delta \leq d(V,V)$.

We have shown that at any moment $\delta = d(V,V)$, which completes the proof. □

Lemma 13 *In the data structure, points can be inserted and deleted, at a cost of $O((\log n)^{k+2})$ amortized time per update.*

Proof: Let $U(n)$ denote the amortized update time for a data structure storing a set V of n points. If $n \leq C$, then $U(n)$ is bounded by a constant. The same holds if $n - 1 \leq C$ and the update is a deletion.

Assume that $n > C$. Here, n denotes the current size of the set V. Let n_0 be the size of the set V at the moment the structure is built. In the same way as in the proof of Lemma 5, it can be shown that during a sequence of $n_0/(4\,(k+1)^k) - 2$ updates, we have $|V_u| \leq 5n/7$, for $u = 1, 2$.

If in the update the data structure for the set V is not rebuilt, we spend $O(\log n)$ time to update the search tree T_V. By Theorem 1, we spend $O((\log n)^{k+1})$ time to update the structure of type $(k-1)$ for the pair (V_1, V_2). Finally, we spend at most $U(5n/7)$ time to update the structure for the appropriate set V_u. Here, $u = 1$ if the update occurs in V_1, and $u = 2$ otherwise. It follows that in case the data structure is not rebuilt, we spend an amount of time that is bounded by

$$O\left((\log n)^{k+1}\right) + U(5n/7).$$

After $n_0/(4\,(k+1)^k) - 2$ updates, the data structure is rebuilt. By Lemma 11, this takes $O\left(n(\log n)^{k+1}\right)$ time. It follows that the amortized update time $U(n)$ satisfies the following recurrence:

$$U(n) = O\left((\log n)^{k+1}\right) + U(5n/7) + \frac{O(n(\log n)^{k+1})}{n_0/(4\,(k+1)^k) - 2},$$

or, because $n = \Theta(n_0)$,

$$U(n) = O\left((\log n)^{k+1}\right) + U(5n/7).$$

From this, it follows that $U(n) = O((\log n)^{k+2})$. □

Lemma 14 *The size of the data structure is bounded by $O(n(\log n)^k)$.*

Proof: Let $S(n)$ be the size of the data structure for a set of size n. If $n \leq C$, then $S(n)$ is bounded by a constant. Otherwise, if $n > C$, then it follows—using Theorem 1—that

$$S(n) = O\left(n(\log n)^{k-1}\right) + S(n_1) + S(n - n_1),$$

for some $2n/7 \leq n_1 \leq 5n/7$. (Since both V_1 and V_2 have size at most $5n/7$, we have $2n/7 \leq n_1 \leq 5n/7$.) From this relation, it follows that $S(n) = O(n(\log n)^k)$. □

This completes the description of the data structure and its update algorithm. We summarize the results we have obtained in the following theorem.

Theorem 2 *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 $O((\log n)^{k+2})$ amortized time per update. The data structure has size $O(n(\log n)^k)$, and can be built in $O(n(\log n)^{k+1})$ time.*

6 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 $O((\log n)^{k+2})$ amortized time.

The best linear size data structure known at present, is given in [9]. This structure maintains the minimal L_t-distance in a k-dimensional point set in $O(n^{2/3} \log n)$ 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.

The solution given in this paper does not use the notion of sparseness, whereas almost all algorithms that compute minimal distances do use this notion. (See e.g. [4,7,9].) Therefore, it would be interesting to know whether the technique introduced here 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 [3,8] for a general approach to such problems for a special type of updates.)

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), pp. 591-604.

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

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

[4] K. Mehlhorn. *Data Structures and Algorithms, Volume 3: Multi-Dimensional Searching and Computational Geometry.* Springer-Verlag, Berlin, 1984.

[5] M.H. Overmars. *Dynamization of order decomposable set problems.* J. of Algorithms **2** (1981), pp. 245-260.

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

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

[8] M. Smid. *A worst-case algorithm for semi-online updates on decomposable problems.* Report A 03/90, Fachbereich Informatik, Universität des Saarlandes, 1990.

[9] 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.

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