Maintaining the minimal distance of a point set in less than linear time *

Michiel Smid

A 06/90

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

* 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 less than linear time*

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

April 12, 1990

Abstract

Consider a set of n points in d-dimensional space. It is shown that the ordered sequence of $O(n^{2/3})$ smallest distances defined by these points can be computed in optimal $O(n \log n)$ time and $O(n)$ space. Here, distances are measured in an arbitrary L_t-metric, where $1 \leq t \leq \infty$. This result is used to give a dynamic data structure of linear size, that maintains the minimal distance of the n points in $O(n^{2/3} \log n)$ time per update.

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. Also, a euclidean nearest neighbor can be computed for each point of a set of n planar points, in optimal $O(n \log n)$ time. 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 [9].)

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 d-dimensional points $p = (p_1, \ldots, p_d)$ and $q = (q_1, \ldots, q_d)$ is

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

defined by

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

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

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

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 [2] 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, however, the best result known today is by Overmars [5,6], who gives 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 this paper, we give an algorithm that computes the $O(n^{2/3})$ smallest distances defined by a set of n points in d-dimensional space. Using this result, we give a dynamic data structure of $O(n)$ size, that maintains the minimal distance of a set of n points in d-space at the cost $O(n^{2/3} \log n)$ time per update. The algorithm works for an arbitrary L_t-metric, where $1 \leq t \leq \infty$.

This is the first data structure that can handle arbitrary updates in sublinear time. In fact, for dimensions $d \geq 4$, the update time is even better than the previously best result for semi-online updates. This best result was an update time of $O(n^{1-\beta(d)}(\log n)^2)$, where $\beta(d) = 1/(d(d+3)+4)$. See [2].

The rest of this paper is organized as follows. In Section 2, we give an algorithm that computes the k smallest distances defined by a set of n points in d-space. For $k \leq n$, the algorithm is improved, resulting in an algorithm that computes the ordered sequence of $O(n^{2/3})$ distances in $O(n \log n)$ time. This result is used in Section 3 to give the dynamic data structure that maintains the minimal distance. We finish the paper in Section 4 with some concluding remarks.

2 Computing the k smallest distances

Let V be a set of n points in d-dimensional space, where $d \geq 2$. We consider an L_t-metric, where t is fixed and satisfies $1 \leq t \leq \infty$. The points of V define $\binom{n}{2}$ distances,

one distance for each pair of points. Given an integer k, $1 \leq k \leq \binom{n}{2}$, we want to compute the k smallest distances, sorted in increasing order. More precisely, we want a sequence $\delta_1 \leq \delta_2 \leq \ldots \leq \delta_k$ of distances, such that the other $\binom{n}{2} - k$ distances all are at least δ_k. (If distances occur more than once, this is a more precise description of the problem.) If we speak about the k-th distance, we mean a distance δ such that there are $k-1$ distances that are at most equal to δ and $\binom{n}{2} - k$ distances that are at least equal to δ.

For $k = 1$, this problem can be solved in $O(n \log n)$ time, and this is optimal. See e.g. [7,9]. For $k > 1$, nothing seems to be known, except for the trivial $O(n^2 + k \log k)$ time algorithm.

Before we give an algorithm for this problem, we prove a lemma. A *d-cube having side-lengths δ* is the d-dimensional hyper-cube that is defined by the product of intervals $[x_1 : x_1 + \delta] \times \ldots \times [x_d : x_d + \delta]$, for some real numbers $x_1, \ldots, x_d$.

Lemma 1 *Let V be a set of n points in d-dimensional space, and let δ_k be the k-th smallest distance in the L_t-metric. Then any d-cube having side-lengths δ_k contains at most $2(d+1)^d \sqrt{k}$ points of V.*

Proof: Let $l = 1/(d+1)$. Consider a d-cube C having side-lengths $l\,\delta_k$. If we put $d+1$ copies of C next to each other, we get a d-dimensional hyper-rectangle having one of its side-lengths equal to $(d+1)\,l\,\delta_k = \delta_k$, and the other side-lengths equal to $l\,\delta_k$. Repeating this, we see that we can cover a d-cube having side-lengths δ_k by $(d+1)^d$ copies of C.

Note that the d-cube C has an L_1-diameter equal to $d\,l\,\delta_k < \delta_k$. That is, each pair of points p and q that are contained in C, or are on the boundary of C, have an L_1-distance less than δ_k. Since the L_t-distance is at most the L_1-distance, i.e., $d_t(p,q) \leq d_1(p,q)$, cube C also has L_t-diameter less than δ_k.

Now assume that a d-cube having side-lengths δ_k contains more than $2(d+1)^d\sqrt{k}$ points of V. Cover this d-cube by $(d+1)^d$ copies of C. Then one of these copies contains more than $2\sqrt{k}$ points of V. These points define more than

$$\binom{\lceil 2\sqrt{k} \rceil}{2} \geq \frac{1}{2} 2\sqrt{k}\,(2\sqrt{k} - 1) \geq k$$

distances. Since these points are contained in a d-cube having an L_t-diameter which is less than δ_k, these distances are all smaller than δ_k. So we have at least k distances that are smaller than δ_k. This is a contradiction, because δ_k is the k-th smallest distance. □

The upper bound in this lemma is tight up to a constant factor: Take $\lceil 2\sqrt{k} \rceil$ points "close" to each other, and the remaining $n - \lceil 2\sqrt{k} \rceil$ points "far" away, all at "large" distances from each other.

We need a data structure for the orthogonal range searching problem. In this problem, we are given a set V of points in d-space. A query consists of an axis-parallel hyper-rectangle $[x_1 : y_1] \times \ldots \times [x_d : y_d]$, and we have to report all points

of V that lie in this rectangle, i.e., all points $p = (p_1, \ldots, p_d)$ in V that satisfy $x_1 \leq p_1 \leq y_1, \ldots, x_d \leq p_d \leq y_d$.

In Mehlhorn [4], range trees with a slack parameter are introduced, that solve the orthogonal range searching problem. We recall his result:

Theorem 1 *Let V be a set of n points in d-space, and let $0 < \epsilon < 1$ be a real number. A range tree with slack parameter $\lceil \epsilon \log n \rceil$, that stores the set V*

1. *has size $O(n)$,*
2. *can be built in $O(n \log n)$ time,*
3. *has a query time of $O(n^{\epsilon(d-1)} \log n + A)$, if A is the number of reported answers,*
4. *has an amortized update time of $O((\log n)^2)$.*

We use a range tree with slack parameter to compute smallest distances in a point set. We take $\epsilon = 1/(3(d-1))$, giving a data structure for the orthogonal range searching problem having a query time of $O(n^{1/3} \log n + A)$.

We fix $1 \leq t \leq \infty$, and we denote by $\delta(p, q)$ the distance between p and q in the L_t-metric. Let V be a set of n points in d-space, and let k be an integer, $1 \leq k \leq \binom{n}{2}$. The algorithm for computing the k smallest distances uses the following data structures:

1. There is a d-dimensional range tree with slack parameter—called the R-tree—that will contain all points of V, that we have considered so far.
2. There is a balanced binary search tree—called the D-tree—that will contain the k smallest distances found so far, in increasing order.

During the algorithm, we maintain the following invariant:

Invariant: Let $V = \{X_1, \ldots, X_n\}$. There is an integer i, such that $\lceil 2\sqrt{k} \rceil \leq i \leq n$. The D-tree contains the k smallest distances that are defined by the points $X_1, \ldots, X_i$. δ_k is the maximal value stored in the D-tree. All points $X_1, \ldots, X_i$ are stored in the R-tree.

Initialization: We set the value of i to $\lceil 2\sqrt{k} \rceil$, and we build an R-tree for the points $X_1, \ldots, X_i$. We compute all distances between these i points. This gives us at least k distances. The k smallest ones are stored in the D-tree, in increasing order. If distances occur more than once, we store some arbitrary k smallest ones. (Note that in a binary search tree, values can be stored more than once, without causing problems in the search and update algorithms, or in the logarithmic time bounds of these algorithms.) Finally, we set $\delta_k :=$ the maximal value stored in the D-tree.

The algorithm: For $i = \lceil 2\sqrt{k} \rceil, \ldots, n-1$, we do the following:

1. Let $p := X_{i+1}$, $p = (p_1, \ldots, p_d)$. Do a range query in the R-tree, with query-cube $[p_1 - \delta_k : p_1 + \delta_k] \times \ldots \times [p_d - \delta_k : p_d + \delta_k]$. For each reported point q for which $\delta(p,q) < \delta_k$, do the following: Insert $\delta(p,q)$ in the D-tree; delete δ_k from the D-tree; set δ_k to the new maximal value that is stored in the D-tree.

2. Insert point p in the R-tree, and increase i by one.

Theorem 2 *The algorithm, as described above, computes the ordered sequence of k smallest L_t-distances, defined by a set of n points in d-dimensional space, in*

$$O(n^{4/3} \log n + n\sqrt{k} \log k)$$

time and uses $O(n + k)$ space.

Proof: It is clear that after the initialization, the D-tree contains the k smallest distances that are defined by the first i points of V. In each iteration of the algorithm, we have to update the D-tree. All new distances that have to be inserted in the D-tree, are caused by point $p = X_{i+1}$ and by points that lie in an L_t-ball around p with radius δ_k. These points surely lie in a d-cube centered at p, having side-lengths $2\delta_k$. (Note that this d-cube is an L_∞-ball with radius $2\delta_k$.) It follows that all new L_t-distances that are less than the current value of δ_k, are correctly inserted in the D-tree. For each inserted distance, another distance is deleted. Hence, the number of distances stored in the D-tree remains equal to k. At the end of the algorithm, the k smallest distances are stored in the D-tree, in sorted order. This proves the correctness of the algorithm.

The R-tree contains at most n points. Hence, by Theorem 1, this structure has size $O(n)$. Since the D-tree stores k distances, its size is bounded by $O(k)$. Therefore, the algorithm uses $O(n + k)$ space.

The time needed to compute all distances between the first $\lceil 2\sqrt{k} \rceil$ points of V is bounded by $O(k)$. Selecting the k smallest distances, and storing them in the D-tree takes $O(k \log k)$ time. Finally, it takes $O(\sqrt{k} \log k)$ time to build the R-tree. Hence, the initialization of the algorithm takes $O(k \log k)$ time.

Now consider the rest of the algorithm. With each iteration, we do a range query in the R-tree, which stores at most n points. The query-rectangle is a d-cube having side-lengths $2\delta_k$, where δ_k is the k-th smallest distance in the set of points that are stored in the R-tree. By Lemma 1, at most $O(\sqrt{k})$ points of the R-tree lie in this rectangle. Hence, the query gives $O(\sqrt{k})$ answers, and—by Theorem 1—these answers are computed in $O(n^{1/3} \log n + \sqrt{k})$ time. For each answer, we spend an amount of $O(\log k)$ time to update the D-tree. Finally, it takes $O((\log n)^2)$ time to update the R-tree. Therefore, in each iteration, we spend $O(n^{1/3} \log n + \sqrt{k} \log k + (\log n)^2)$ time. For all iterations together, this takes $O(n^{4/3} \log n + n\sqrt{k} \log k)$ time.

It follows that the entire algorithm spends an amount of time that is bounded by

$$O(k \log k + n^{4/3} \log n + n\sqrt{k} \log k) = O(n^{4/3} \log n + n\sqrt{k} \log k),$$

because $k < n^2$. This proves the theorem. □

Remark: In the 2-dimensional case, the time bound can be improved to $O(n \log n + n\sqrt{k} \log k)$, by generalizing the sweep-line algorithm of Hinrichs, Nievergelt and Schorn [3]. This gives a much better result if k is small. We do not give this sweep-line algorithm here, because we now give an improved algorithm that runs even faster for small k.

Let V be a set of n points in d-space, and let k be an integer such that $1 \leq k \leq n$. In the k smallest distances, at most $2k$ points are involved. In the next algorithm, we first discard at least $n - 2k$ points, from which we know that they do not contribute to the k smallest distances.

An improved algorithm: First, we compute for each point in V its nearest neighbor, using the algorithm of Vaidya [9]. (If $d = 2$, we can use the algorithm given in [7].) This gives n pairs of points and n distances. We select the k smallest of these n distances. This gives a set of k pairs of points, and hence a set V' of at most $2k$ points. Then we compute the k smallest distances in this set V', using our first algorithm.

Theorem 3 *Let $1 \leq k \leq n$. The improved algorithm correctly computes the ordered sequence of k smallest L_t-distances in the set V, in*

$$O(n \log n + k\sqrt{k} \log k)$$

time and $O(n)$ space.

Proof: To prove the correctness of the algorithm, we only have to prove that the k smallest distances in the set V' are equal to those in the set V. Let δ'_k be the k-th smallest distance in the n distances that we get from Vaidya's algorithm. Then the k smallest distances in the set V are at most equal to δ'_k. Let q be a point in $V \setminus V'$. Since the distance of q to all other points of V is at least δ'_k, point q does not contribute to the k smallest distances in the set V. (This argument is also correct if distances occur more than once.) This proves that the algorithm is correct.

It takes $O(n \log n)$ time to compute for each point in the set V its nearest neighbor. (See [9], or [7] for the case $d = 2$.) Next, the time needed to select all points that will be put in the set V', and to remove duplicates, is bounded by $O(k \log k)$. (Note that a point can be put several times in V'. Therefore, we really have to remove duplicates.) We are left with a set of at most $2k$ points, for which we compute the k smallest distances. By Theorem 2, this takes $O(k\sqrt{k} \log k)$ time. Hence, the runtime of the algorithm is bounded by

$$O(n \log n + k \log k + k\sqrt{k} \log k) = O(n \log n + k\sqrt{k} \log k).$$

The space bound follows from Theorem 2, and from the fact that Vaidya's algorithm uses $O(n)$ space. □

It is well-known that it takes $\Omega(n \log n)$ time to compute the minimal distance in a set of n points. (See e.g. [7].) Clearly, this lower bound extends to an $\Omega(k + n \log n)$

lower bound for computing the ordered sequence of k smallest distances. Therefore, Theorem 3 implies:

Corollary 1 *Given a set of n points in d-space, the ordered sequence of $O(n^{2/3})$ smallest distances can be computed in $O(n \log n)$ time and $O(n)$ space, and this is optimal.*

3 Maintaining the minimal distance

The algorithm to be presented now is an extension of the algorithm of the previous section. The idea is as follows. We start with the ordered sequence of k smallest distances, where $k = n^{2/3}$. In each update, we have to update this sequence. By Lemma 1, each point in our point set "occurs" at most $O(\sqrt{k})$ times in this sequence. Therefore, in a deletion, at most $O(\sqrt{k})$ distances have to be removed in this sequence. It follows that in this way we can perform $\Omega(\sqrt{k})$ updates until all candidates for being the minimal distance have been removed. If this happens, we start over the algorithm.

Let V be a set of N points in d-space. We fix $1 \leq t \leq \infty$. All distances are measured in the L_t-metric. The distance between p and q is denoted by $\delta(p, q)$. Let $k = \lfloor N^{2/3} \rfloor$. The data structure consists of the following:

1. There is a balanced binary search tree—called the D-tree—in which we store the l smallest distances defined by the current set V, in sorted order. Here, l is an integer, such that $1 \leq l \leq k$. (As before, some distances may occur more than once in the D-tree.)

2. There is a value δ resp. D, that stores the current minimal resp. maximal distance that is stored in the D-tree.

3. All points that are currently present are stored in a d-dimensional range tree with slack parameter, called the R-tree. We take the slack parameter equal to $(\log n)/(3(d-1))$.

The integer l in 1. is only needed for reference in the proofs of the correctness and the runtime of the algorithm.

Initialization: The D-tree is built using the improved algorithm of Section 2. The value of l is set to k. The value δ resp. D is set to the minimal resp. maximal value that is stored in the D-tree. The R-tree is built using the algorithm given in [4].

The insert algorithm: To insert a point $p = (p_1, \ldots, p_d)$, we do the following:

1. In the R-tree, we do a range query with query-cube $[p_1 - D : p_1 + D] \times \ldots \times [p_d - D : p_d + D]$. For each reported answer q, such that $\delta(p, q) < D$, we insert

$\delta(p,q)$ in the D-tree; we delete D from the D-tree; and we set D to the new maximal value that is stored in the D-tree.

2. We set δ to the minimal value that is stored in the D-tree.

3. Finally, we insert p in the R-tree.

Note that no problems arise when distances occur more than once in the D-tree.

The delete algorithm: To delete a point $p = (p_1, \ldots, p_d)$, we do the following:

1. In the R-tree, we do a range query with query-cube $[p_1 - D : p_1 + D] \times \ldots \times [p_d - D : p_d + D]$. For each reported answer q, such that $\delta(p,q) \leq D$, we delete $\delta(p,q)$ from the D-tree; we decrease l by one; and we set D to the new maximal value that is stored in the D-tree.

2. We set δ to the minimal value that is stored in the D-tree.

3. Finally, we delete p from the R-tree.

If a distance $\delta(p,q) = D$ is deleted, it might be possible that this distance "does not occur" in the D-tree, i.e., although the distance is stored in the D-tree, it is stored there because another pair of points are already at distance D, and D is stored because of this pair of points, not because of the pair p and q. This does not, however, give any problems.

Rebuilding: If after an operation, the D-tree gets empty, or after $\lfloor N^{1/3} \rfloor$ updates, we start over again. That is, we set $k = \lfloor M^{2/3} \rfloor$, where M is the number of points that are present at that moment, and we build the structures anew. Then we proceed performing updates as above.

Lemma 2 *At any moment, the D-tree stores the l minimal distances of the current set of points. Here, l satisfies $1 \leq l \leq k = \lfloor N^{2/3} \rfloor$. Also, at any moment, the variable δ contains the minimal distance of the current set of points.*

Proof: It is clear that after the initialization, the D-tree contains the $l = k = \lfloor N^{2/3} \rfloor$ smallest distances, and that δ is equal to the minimal distance of the points.

If a point p is inserted, new distances are introduced. All distances that have to be inserted in the D-tree are caused by p and by points that lie in an L_t-ball centered at p with radius D. These points lie in a d-cube centered at p, having side-lengths $2D$. It follows that all new distances that are less than the current value of D, are correctly inserted in the D-tree. For each inserted distance, another distance is deleted. Hence, the number of distances stored in the D-tree—i.e., the value of l—does not change with an insertion.

Similarly, when a point p is deleted, we delete all distances that are caused by p and that are at most equal to the current value of D. For each deleted distance, we decrease l by one. In this case, the D-tree will store less distances than before the deletion. All distances that are stored, however, are the l smallest ones in the current set of points. □

Lemma 3 *If the data structure is rebuilt, $\Theta(N^{1/3})$ updates have been performed.*

Proof: If after $\lfloor N^{1/3} \rfloor$ updates, the D-tree is still not empty, we rebuild the data structure. Hence, after $O(N^{1/3})$ updates, a rebuilding is done.

At any moment, the D-tree contains the smallest l distances, for some l satisfying $1 \leq l \leq k = \lfloor N^{2/3} \rfloor$. By Lemma 1, each d-cube having side-lengths $2D$—where D is the l-th distance in the current set of points—contains at most $O(\sqrt{l}) = O(\sqrt{k}) = O(N^{1/3})$ points. When a point is inserted, the number of distances that are stored in the D-tree does not change. When a point is deleted, $O(N^{1/3})$ distances are deleted. Since initially, there are $\lfloor N^{2/3} \rfloor$ distances stored in the D-tree, it takes $\Omega(N^{1/3})$ updates before this tree becomes empty, i.e., before the data structure is rebuilt. □

Theorem 4 *There exists a data structure that maintains the minimal L_t-distance of a set of n points in d-space, at the cost of $O(n^{2/3} \log n)$ amortized time per update. The data structure has size $O(n)$ and can be built in $O(n \log n)$ time.*

Proof: The bounds on the building time and the size of the data structure follow from Corollary 1 and Theorem 1. Consider an update such that the data structure is not rebuilt. Since the number of answers to the range query is bounded by $O(N^{1/3})$, such a query takes $O(n^{1/3} \log n + N^{1/3})$ time, if n is the current number of points. (See Theorem 1.) For each answer, we spend an amount of $O(\log k) = O(\log N)$ time in the D-tree. Finally, it takes $O((\log n)^2)$ amortized time to update the R-tree. Hence, if the structure is not rebuilt we spend amortized

$$O(n^{1/3} \log n + N^{1/3} \log N)$$

time in an update. By Lemma 3, it takes $\Theta(N^{1/3})$ updates, before we rebuild the structure. Therefore, the current number of points—n—is always $\Theta(N)$. Hence, in case no rebuilding is done, an update takes amortized $O(n^{1/3} \log n)$ time.

The structure is rebuilt once every $\Theta(n^{1/3})$ updates, and this takes $O(n \log n)$ time. It follows that the amortized update time is bounded by

$$O(n^{1/3} \log n) + O\left(\frac{n \log n}{n^{1/3}}\right) = O(n^{2/3} \log n).$$

This proves the theorem. □

The amortized complexity of Theorem 4 can be made worst-case. Then we need a worst-case version of Theorem 1. More precisely, we need a structure having the complexity of Theorem 1, except that now a sequence of $O(n^{1/3})$ updates can be performed at the cost of $O(n^{2/3} \log n)$ time per update in the worst case. (Note that the amortized update time in Theorem 1—which is $O((\log n)^2)$—may increase to $O(n^{2/3} \log n)$, without increasing the overall complexity of the algorithm for maintaining the minimal distance.) This structure can easily be obtained as follows.

We start with a perfectly balanced range tree with slack parameter, storing the N points that are present at that moment. This structure consists of a binary tree,

some nodes of which have a pointer to another binary tree, some nodes of which have a pointer to another binary tree, etc. Deletions are performed in this range tree, without rebalancing. Since initially all binary trees that occur in this data structure have heights $O(\log N)$, and since we perform only $O(N^{1/3})$ deletions in this structure, the heights of these binary trees is always bounded by $O(\log n)$, if n is the current number of points. Therefore, the query time of this range tree remains within the bound of Theorem 1. Furthermore, the worst-case deletion time is bounded by $O((\log n)^2)$. Besides this range tree, we maintain a (one-dimensional) balanced binary search tree T, in which the inserted points are stored. Hence, T contains $O(N^{1/3})$ points. Clearly, this tree can be maintained in $O(\log n)$ time. (If a point p is to be deleted, we first check whether p occurs in T. If it does, we delete it. Otherwise, p is stored in the range tree, and we delete it, without rebalancing.)

A range query is performed as follows. We first query the range tree using its query algorithm. This gives a list l_1 of answers. Then we walk through the binary tree T, and we just check for each of the $O(N^{1/3}) = O(n^{1/3})$ points whether it lies in the query rectangle. If it does, we store it in a list l_2. The answers to the entire query are formed by the union of the two lists of answers obtained. Note that each answer is reported exactly once. This query algorithm runs in

$$O(n^{1/3} \log n + |l_1|) + O(n^{1/3}) = O(n^{1/3} \log n + A)$$

time, if A is the total number of reported answers. Hence, the query time is the same as in Theorem 1.

It follows that we have a data structure having the same complexity as that of Theorem 1, except that a sequence of $O(n^{1/3})$ updates can be performed at the cost of $O((\log n)^2)$ time in the worst case. (Note that the update time is much smaller than the admissible $O(n^{3/2} \log n)$ bound.)

Using this structure, the time bound of Theorem 4 can be made worst-case by applying techniques that are similar to techniques that are given in [6,8]. We leave the details to the reader.

Theorem 5 *There exists a data structure that maintains the minimal L_t-distance of a set of n points in d-space, at the cost of $O(n^{2/3} \log n)$ time per update, in the worst case. The data structure has size $O(n)$ and can be built in $O(n \log n)$ time.*

Remark: The data structure presented here can also be used to maintain the k smallest distances, for some k that is bounded by $O(n^{2/3})$: Just start with the $2k$ smallest distances. If the D-tree stores k distances, start over again. The complexity bounds are the same.

4 Concluding remarks

We have presented an algorithm that computes the ordered sequence of $O(n^{2/3})$ smallest distances in a set of n points in optimal $O(n \log n)$ time. This algorithm

was used to obtain a dynamic data structure that maintains the minimal distance in a set of n points, at the cost of $O(n^{2/3} \log n)$ time per update in the worst case. This is the first algorithm that maintains the minimal distance in sublinear time when arbitrary updates are carried out. In higher dimensions, it is even faster than the best result for—the less general—semi-online updates given in [2].

The update time can be improved if it is possible to compute more smallest distances in $O(n \log n)$ time. Suppose we are able to compute the k smallest distances in $O(n \log n)$ time. Then using the results of this paper, we get a data structure that maintains the minimal distance in

$$O\left(\sqrt{k} \log k + \frac{n \log n}{\sqrt{k}}\right)$$

time. Here, the first term is the time needed if no rebuilding is done, and the second term is the time to rebuild the structures, wich happens once every $\Theta(\sqrt{k})$ updates. Improving the value of k from $n^{2/3}$ to e.g. n, would result in an update time of $O(\sqrt{n} \log n)$. It is an interesting open problem whether this is possible.

Another open problem is to find the complexity of computing the ordered sequence of k smallest distances in a set of n points, for large values of k. At present, the only lower bound is $\Omega(k+n \log n)$. For $n^{2/3} \leq k \leq n$, the best upper bound is the $O(k\sqrt{k} \log k)$ bound of Theorem 3, and for $k \geq n$, the best result is the $O(n\sqrt{k} \log k)$ bound of Theorem 2. Especially interesting is the case $k = \binom{n}{2}$. That is, is it possible to order all distances in $O(n^2)$ time?

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] 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.

[3] K. Hinrichs, J. Nievergelt and P. Schorn. *Plane-sweep solves the closest pair problem elegantly.* Inform. Proc. Lett. **26** (1987/88), pp. 255-261.

[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] P.M. Vaidya. *An optimal algorithm for the all-nearest-neighbor problem.* Proc. 27-th Annual IEEE Symp. on Foundations of Computer Science, 1986, pp. 117-122.