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

Michiel Smid†

April 10, 1990

1 Introduction

Let V be a set of n points in d-dimensional space. We are interested in maintaining the minimal distance of the points in V, when points are inserted and deleted in V. Distances are measured in the (Minkowski) L_t-metric, where $1 \leq t \leq \infty$ is fixed throughout this paper.

Dobkin and Suri [2] considered the problem for a restricted type of updates, so-called *semi-online* updates. They showed that in the plane, the minimal L_2-distance can be maintained at the cost of $O((\log n)^2)$ time per semi-online update. For arbitrary updates on the minimal euclidean distance of a set of planar points, the best result is by Aggarwal et al.[1]: they show that in a Voronoi diagram, points can be inserted and deleted in $O(n)$ time. This leads to an update time of $O(n)$ for the minimal distance

In this paper, we give a dynamic data structure that maintains the minimal L_t-distance of a set of n points in d-dimensional space at the cost $O(n^{2/3} \log n)$ time per update.

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].

2 Computing the k smallest distances

Let V be a set of n points in d-space. These points define $\binom{n}{2}$ distances, one for each pair of points. Given an integer k, we want to compute the k smallest distances, sorted in increasing order. We assume in this paper that all $\binom{n}{2}$ distances are different.

We need a lemma. A *d-cube having side-lengths δ* is the 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 δ_k be the k-th smallest L_t-distance in the set V. 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 $l\,\delta_k$. A d-cube having side-lengths δ_k can be covered by $(d+1)^d$ copies of C. This d-cube C has an L_1-diameter equal to $d\,l\,\delta_k < \delta_k$. Therefore, cube C also has L_t-diameter less than δ_k. Now assume that

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

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

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 k distances, that are all smaller than δ_k. This is a contradiction. □

We need a data structure for the orthogonal range searching problem:

Theorem 1 (Mehlhorn [3]) *Let V be a set of n points in d-space. A range tree with slack parameter $\lceil(\log n)/(3(d-1))\rceil$, storing V has size $O(n)$, can be built in $O(n \log n)$ time, and has an amortized update time of $O((\log n)^2)$. Given an axis-parallel hyperrectangle in d-space, all A points of V that are in this rectangle, can be found in $O(n^{1/3}\log n + A)$ time.*

We denote by $\delta(p,q)$ the distance between p and q in the L_t-metric. 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.

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 $X_1, \ldots, X_i$. $\delta_k =$ maximum(D-tree). All points $X_1, \ldots, X_i$ are stored in the R-tree.

Initialization: Set $i := \lceil 2\sqrt{k}\rceil$, and build an R-tree for $X_1, \ldots, X_i$. Compute all distances between these i points. The k smallest of these distances are stored in the D-tree, in increasing order. Set $\delta_k :=$ maximal(D-tree).

The algorithm: For $i = \lceil 2\sqrt{k}\rceil, \ldots, n-1$, 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 $\delta_k :=$ maximum(D-tree).

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

Theorem 2 *The algorithm computes the ordered sequence of k smallest distances in time $O(n^{4/3}\log n + n\sqrt{k}\log k)$, using $O(n+k)$ space.*

Proof: 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 lie in a d-cube centered at p, having side lengths $2\delta_k$. Hence, 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. This proves the correctness of the algorithm.

The initialization of the algorithm takes $O(k \log k)$ time. Consider the rest of the algorithm. With each iteration, we do a range query in the R-tree. 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, which are computed in $O(n^{1/3}\log n+\sqrt{k})$ time. For each answer, we spend $O(\log k)$ time in the D-tree. Therefore, in each iteration, we spend $O(n^{1/3}\log n+\sqrt{k}\log k)$ time. For all iterations together, this takes $O(n^{4/3}\log n+n\sqrt{k}\log k)$ time. □

An improved algorithm: Assume $1 \leq k \leq n$. Compute for each point in V its nearest neighbor, as in [4]. This gives n distances. Select the k smallest ones. This gives a set of k pairs of points, and hence a set V' of at most $2k$ points. Then compute the k smallest distances in this set V', using the algorithm given above.

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: The algorithm is correct, because the k smallest distances in the set V' are equal to those in the set V. It takes $O(n\log n)$ time to compute for each point in the set V its nearest neighbor. (See [4].) The time needed to select all points that will be put in the set V' is bounded by $O(k\log k)$. 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. □

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 optimal $O(n\log n)$ time and $O(n)$ space.*

3 Maintaining the minimal distance

Let V be a set of N points in d-space. Let $k = \lfloor N^{2/3} \rfloor$. The data structure consists of the following.

1. There is a balanced binary search tree—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 $1 \leq l \leq k$. δ = minimum(D-tree), D = maximum(D-tree).

2. All points that are currently present are stored in a d-dimensional range tree of Theorem 1, called the R-tree.

Initialization: The D-tree is built using the improved algorithm of Section 2. We set $l := k$; $\delta :=$ minimum(D-tree); $D :=$ maximum(D-tree). The R-tree is built using the algorithm given in [3].

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

1. In the R-tree, do a range query with query-cube $[p_1 - D : p_1 + D] \times \ldots \times [p_d - D : p_d + D]$. For each answer q, such that $\delta(p,q) \leq D$, delete $\delta(p,q)$ from the D-tree; set $l := l - 1$; set $D :=$ maximum(D-tree).

2. Set $\delta :=$ minimum(D-tree), and delete p from the R-tree.

The insert algorithm is the same as the algorithm in Section 2.

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

Proof: After the initialization, the D-tree contains the $k = \lfloor N^{2/3} \rfloor$ smallest distances. If a point p is inserted, new distances are introduced. All distances that have to be stored in the D-tree are caused by p and by points that lie in an L_t-ball around p with radius D. These points surely lie in a d-cube centered at p, having side lengths $2D$. Hence, 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. When a point p is deleted, we delete all distances that are caused by p and that are smaller than the current value of D. In this case, the D-tree will store less distances than before the deletion. All distances that are stored, however, are the 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: After $\lfloor N^{1/3} \rfloor$ updates, the D-tree will have been rebuilt. When a point is inserted, the number of distances that are stored in the D-tree does not change. It follows from Lemma 1 that with a deletion, $O(N^{1/3})$ 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: Consider an update such that the data structure is not rebuilt. Since the number of answers to each 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. For each answer, we spend $O(\log k) = O(\log N)$ time in the D-tree. 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. 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 above by $O(n^{1/3} \log n) + O((n \log n)/n^{1/3}) = O(n^{2/3} \log n)$. □

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

[4] 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.