New techniques for exact and approximate dynamic closest-point problems

Sanjiv Kapoor[*] Michiel Smid[†]

Abstract

Let S be a set of n points in $\mathbb{R}^D$. It is shown that a range tree can be used to find an L_∞-nearest neighbor in S of any query point, in $O((\log n)^{D-1} \log\log n)$ time. This data structure has size $O(n(\log n)^{D-1})$ and an amortized update time of $O((\log n)^{D-1} \log\log n)$. This result is used to solve the $(1+\epsilon)$-approximate L_2-nearest neighbor problem within the same bounds. In this problem, for any query point p, a point $q \in S$ is computed such that the euclidean distance between p and q is at most $(1+\epsilon)$ times the euclidean distance between p and its true nearest neighbor. This is the first dynamic data structure for this problem having close to linear size and polylogarithmic query and update times.

New dynamic data structures are given that maintain a closest pair of S. For $D \geq 3$, a structure of size $O(n)$ is presented with amortized update time $O((\log n)^{D-1} \log\log n)$. For $D = 2$ and any non-negative integer constant k, structures of size $O(n \log n/(\log\log n)^k)$ (resp. $O(n)$) are presented having an amortized update time of $O(\log n \log\log n)$ (resp. $O((\log n)^2/(\log\log n)^k)$). Previously, no deterministic linear size data structure having polylogarithmic update time was known for this problem.

1 Introduction

Closest-point problems are among the basic problems in computational geometry. In such problems, a set S of n points in $\mathbb{R}^D$ is given and we have to store it in a data structure such that a point in S nearest to a query point can be computed efficiently, or we have to compute a closest pair in S, or for each point in S another point in S that is closest to it. Distances are measured in the L_t-metric, for a fixed t, $1 \leq t \leq \infty$. These problems are known as the *nearest-neighbor problem*, the *closest pair problem*, and the *all-nearest-neighbors-problem*. In the dynamic version, the set S is changed by insertions and deletions of points.

The planar version of the nearest-neighbor problem can be solved optimally, i.e., with $O(\log n)$ query time using $O(n)$ space, by means of Voronoi diagrams. In higher dimensions, however, the situation is much worse. The best results known are due to Clarkson [3] and Arya et al.[1]. In [3], a randomized data structure is given that finds a nearest-neighbor of a query point in $O(\log n)$ expected time. This structure has size $O(n^{\lceil D/2 \rceil + \delta})$, where δ is an arbitrarily small positive constant. In [1], the problem is solved with an expected query time of $O(n^{1-1/\lceil (D+1)/2 \rceil}(\log n)^{O(1)})$ using $O(n \log\log n)$ space. It seems that in higher dimensions it is impossible to obtain polylogarithmic query time using $O(n(\log n)^{O(1)})$ space. Moreover, even in the planar case there is no dynamic data

[*]Department of Computer Science, Indian Institute of Technology, Hauz Khas, New Delhi 110016, India. E-mail: skapoor@cse.iitd.ernet.in

[†]Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany. E-mail: michiel@mpi-sb.mpg.de. This author was supported by the ESPRIT Basic Research Actions Program, under contract No. 7141 (project ALCOM II).

10th Computational Geometry 94-6/94 Stony Brook, NY, USA
© 1994 ACM 0-89791-648-4/94/0006..$3.50

structure known that has polylogarithmic query and update times and that uses $O(n(\log n)^{O(1)})$ space.

Therefore, it is natural to ask whether the *approximate nearest-neighbor problem* allows more efficient solutions. Let $\epsilon > 0$. A point $q \in S$ is called a $(1+\epsilon)$-approximate neighbor of a point $p \in \mathbb{R}^D$, if $d_t(p,q) \leq (1+\epsilon)d_t(p,p^*)$, where $p^* \in S$ is the true nearest-neighbor of p.

This approximate neighbor problem was considered in Arya et al.[1, 2]. In the latter paper, the problem is solved optimally: They give a deterministic data structure of size $O(n)$ that can find a $(1+\epsilon)$-approximate neighbor in $O(\log n)$ time, for any positive constant ϵ. At this moment, however, no dynamic data structures are known for this problem. In this paper, we prove the following results.

Theorem 1 *Let S be a set of n points in $\mathbb{R}^D$. There exists a dynamic data structure of size $O(n(\log n)^{D-1})$ that, given a query point $p \in \mathbb{R}^D$, finds an L_∞-neighbor of p in S in $O((\log n)^{D-1}\log\log n)$ time. This data structure has an amortized update time of $O((\log n)^{D-1}\log\log n)$.*

Theorem 2 *Let S be a set of n points in $\mathbb{R}^D$ and let ϵ be a positive constant. There exists a dynamic data structure of size $O(n(\log n)^{D-1})$ that, given a query point $p \in \mathbb{R}^D$, finds a $(1+\epsilon)$-approximate L_2-neighbor of p in S in $O((\log n)^{D-1}\log\log n)$ time. This data structure has an amortized update time of $O((\log n)^{D-1}\log\log n)$.*

We also consider the dynamic closest pair problem, in which we have to maintain a closest pair under insertions and deletions. This problem has been investigated only recently. In Table 1, we give an overview of the currently best known data structures for maintaining a closest pair under insertions and/or deletions of points. For an up-to-date survey, we refer the reader to Schwarz's Ph.D. Thesis [9].

Note that all data structures of Table 1 are deterministic and can be implemented in the algebraic computation tree model. If we add randomization to this model, then there is a data structure of size $O(n)$ that maintains a closest pair in $O((\log n)^2)$ expected time per update. (See Golin et al.[5].)

In this paper, we give new deterministic data structures for the dynamic closest pair problem. These structures are based on our solution to the L_∞-neighbor problem, on data structures for maintaining boxes of constant overlap, and on a new transformation. Given *any* dynamic closest pair data structure having more than linear size, this transformation produces another dynamic closest pair structure that uses less space. The complexity bounds of the new structures are shown in the last three lines of Table 1. The results of the last two lines are obtained by applying the transformation repeatedly. Note that we obtain the first linear size deterministic data structure that maintains a closest pair in polylogarithmic time. As a by-product, we prove:

Theorem 3 *Let S be a set of n points in $\mathbb{R}^D$. There exists a dynamic data structure of size $O(n(\log n)^{D-1})$ that maintains an L_∞-neighbor of each point in S. This data structure has an amortized update time of $O((\log n)^{D-1}\log\log n)$.*

All our algorithms use classical and well-understood data structuring techniques, such as range trees, segment trees and dynamic fractional cascading.

The rest of this paper is organized as follows. In Section 2, we show how range trees can be used to solve the L_∞-neighbor problem. We also give the data structure for solving the approximate L_2-nearest neighbor problem.

Our first data structure for maintaining the closest pair stores a dynamically changing set of boxes that are of constant overlap, i.e., there is a constant c such that each box contains the centers of at most c boxes in its interior. We have to maintain such boxes under insertions and deletions such that for any query point $p \in \mathbb{R}^D$, we can find all boxes that contain p. In Section 3, we give two data structures for this problem. The first one is based on segment trees. Therefore, its size is superlinear. We also give a linear size solution having a slightly worse update time in the planar case. This solution is based on skewer trees. (See [4, 11].)

In Section 4, we use the results obtained to give a new data structure that maintains the closest pair in a point set. This structure has size $O(n(\log n)^{D-1})$ and an amortized update time of

mode	dimension	update time	w/a	space	reference
insertions	$D \geq 2$	$\log n$	w	n	[9, 10]
deletions	$D \geq 2$	$(\log n)^D$	a	$n(\log n)^{D-1}$	[13]
fully dynamic	$D \geq 2$	$\sqrt{n} \log n$	w	n	[8]
fully dynamic	$D \geq 2$	$(\log n)^D \log \log n$	a	$n(\log n)^D$	[12]
fully dynamic	$D \geq 3$	$(\log n)^{D-1} \log \log n$	a	n	this paper
fully dynamic	2	$\log n \log \log n$	a	$n \log n / (\log \log n)^k$	this paper
fully dynamic	2	$(\log n)^2 / (\log \log n)^k$	a	n	this paper

Table 1: Deterministic data structures for the dynamic closest pair problem. Distances are measured in the L_t-metric, for a fixed $1 \leq t \leq \infty$. In the last two lines, k is an arbitrary non-negative integer constant. The update times are either worst-case (w) or amortized (a).

$O((\log n)^{D-1} \log \log n)$. It immediately gives the dynamic data structure for the all-L_∞-nearest-neighbors problem. In Section 5, we give the transformation that reduces the size of a dynamic closest pair structure. Applying this transformation repeatedly to the structure of Section 4 gives the new results mentioned in Table 1.

2 The L_∞-neighbor problem

Let S be a set of n points in $\mathbb{R}^D$. We want to store this set into a data structure such that for any query point $p \in \mathbb{R}^D$, we can find a point in S having minimal L_∞-distance to p. Such a point is called an *L_∞-neighbor* of p in S.

Let q be an L_∞-neighbor of p in the set $\{s \in S : s_1 \geq p_1\}$. We call q a *right-L_∞-neighbor* of p in S. Similarly, a point r is called a *left-L_∞-neighbor* of p in S, if it is an L_∞-neighbor of p in the set $\{s \in S : s_1 \leq p_1\}$. In order to guarantee that both neighbors always exist, we add the $2D$ artificial points $(a_1, \ldots, a_D)$, where all a_i are zero, except for one which is either ∞ or $-\infty$, to the set S.

The data structure that solves the L_∞-neighbor problem is just a D-dimensional *range tree* storing the points of S and the artificial points. (See [7].) We recall an important property of range trees: Let p be any point in $\mathbb{R}^D$. Consider the set $\{r \in S : r_1 \geq p_1\}$, i.e., the set of all points in S having a first coordinate that is at least equal to p's first coordinate. Using the range tree, we can decompose this set into $O(\log n)$ canonical subsets: Initialize $M := \emptyset$. Starting in the root of the main tree, search for the leftmost leaf storing a point whose first coordinate is at least equal to p_1. During this search, each time we move from a node v to its left son, add the right son of v to the set M. Let v be the leaf in which this search ends. If the point stored in this leaf has a first coordinate that is at least equal to p_1, then add v to the set M. It is well known that the final set M satisfies $\{r \in S : r_1 \geq p_1\} = \bigcup_{v \in M} S_v$. Here, S_v denotes the set of points that are stored in the subtree of v. Range trees can be maintained under insertions and deletions of points. The update algorithms use dynamic fractional cascading. (See [6].) A D-dimensional range tree for a set of n points has $O(n(\log n)^{D-1})$ size and can be maintained in $O((\log n)^{D-1} \log \log n)$ amortized time per insertion and deletion.

We use the following notations. For any point $p = (p_1, p_2, \ldots, p_D) \in \mathbb{R}^D$, we denote by p' the point $(p_2, \ldots, p_D)$ in $\mathbb{R}^{D-1}$. If S is a set of points in $\mathbb{R}^D$, then $S' = \{p' : p \in S\}$. In Figure 1, our recursive algorithm is given that finds an L_∞-neighbor of any query point $p \in \mathbb{R}^D$.

Lemma 1 *1. If the variable stop has value false after the while-loop of Stage 2 has been completed, then the set C contains a right-L_∞-neighbor of p in S.*

2. If the variable stop has value true after the while-loop of Stage 2 has been completed, then the set $C \cup S_v$ contains a right-L_∞-neighbor of p in S.

3. During the while-loop of Stage 3, the set $C \cup S_v$ contains a right-L_∞-neighbor of p in S.

```
Algorithm Neighbor($p, S, D$)   (* returns an $L_\infty$-neighbor of $p$ in $S$ *)
begin
1. Assume $D = 1$.
   $q^L :=$ maximal element in the 1-dimensional range tree that is less than $p$;
   $q^R :=$ minimal element in the 1-dimensional range tree that is at least equal to $p$;
   if $|p - q^R| \leq |p - q^L|$ then return $q^R$ else return $q^L$ fi;
2. Assume $D \geq 2$.
2a. Compute a right-$L_\infty$-neighbor:
    Stage 1. Compute the set $M$ of nodes of the main tree, see above.
             Number these nodes $v_1, v_2, \ldots, v_m$, $m = |M|$, where $v_i$ is closer to
             the root than $v_{i-1}$, $2 \leq i \leq m$.
    Stage 2. (* one of the sets $S_{v_i}$ contains a right-$L_\infty$-neighbor of $p$ *)
             $C := \emptyset$; $i := 1$; $stop := false$;
             while $i \leq m$ and $stop = false$
             do $x' :=$ Neighbor($p', S'_{v_i}, D - 1$);
                $r :=$ the point stored in the rightmost leaf of the subtree of $v_i$;
                if $d_\infty(p', x') > |p_1 - r_1|$
                then $C := C \cup \{x\}$; $i := i + 1$
                else $v := v_i$; $stop := true$
                fi
             od;
             if $stop = false$
             then $q^R :=$ a point of $C$ having minimal $L_\infty$-distance to $p$;
                  goto 2b
             fi;
    Stage 3. (* the set $C \cup S_v$ contains a right-$L_\infty$-neighbor of $p$ *)
             while $v$ is not a leaf
             do $w :=$ left son of $v$;
                $x' :=$ Neighbor($p', S'_w, D - 1$);
                $r :=$ the point stored in the rightmost leaf of the subtree of $w$;
                if $d_\infty(p', x') > |p_1 - r_1|$
                then $C := C \cup \{x\}$; $v :=$ right son of $v$
                else $v := w$
                fi
             od;
             $q^R :=$ a point of $C \cup S_v$ having minimal $L_\infty$-distance to $p$;
2b. Compute a left-$L_\infty$-neighbor:
    In a completely symmetric way, compute a left-$L_\infty$-neighbor $q^L$ of $p$;
2c. if $d_\infty(p, q^R) \leq d_\infty(p, q^L)$ then return $q^R$ else return $q^L$ fi
end
```

Figure 1: Finding an L_∞-neighbor.

This lemma implies that point q^R which is computed in part **2a** is a right-L_∞-neighbor of p in S. Similarly, point q^L computed in part **2b** is a left-L_∞-neighbor of p. This proves that the point that is returned in part **2c** is an L_∞-neighbor of p.

Algorithm *Neighbor(p, S, D)* can be implemented using fractional cascading, such that its running time is bounded by $O((\log n)^{D-1})$. Hence, we have a data structure for the L_∞-neighbor problem that has a query time of $O((\log n)^{D-1})$ and that uses $O(n(\log n)^{D-1})$ space. Range trees can be maintained under insertions and deletions of points. Because of dynamic fractional cascading, the query time increases by a factor of $O(\log\log n)$. This proves Theorem 1.

Let S be a set of n points in $\mathbb{R}^D$ and let $\epsilon > 0$ be a fixed constant. For any point $p \in \mathbb{R}^D$, we denote by p^* its L_2-neighbor in S. A point $q \in S$ is called a $(1+\epsilon)$-*approximate* L_2-*neighbor* of p if $d_2(p,q) \leq (1+\epsilon)d_2(p,p^*)$. We want to store the set S in a data structure such that for any query point $p \in \mathbb{R}^D$ we can find a $(1+\epsilon)$-*approximate* L_2-*neighbor* of it.

Let $p \in \mathbb{R}^D$ and let q be an L_∞-neighbor of p in S. It is easy to show that q is a $\sqrt{D}$-approximate L_2-neighbor of p. We extend this as follows. Let $(\mathcal{F}_i)$ be a family of orthonormal coordinate systems all sharing the same origin. Assume that for any point x in $\mathbb{R}^D$ there is an index i such that in $\mathcal{F}_i$ all coordinates of x are almost equal, i.e., within $1+\epsilon$ from each other. In Yao [15], it is shown how such a family consisting of $O((c/\epsilon)^{D-1})$ coordinate systems can be constructed. In the planar case, this family is obtained by rotating the X- and Y-axes over an angle of $i \cdot \phi$, $0 \leq i < 2\pi/\phi$, where $0 < \phi < \pi/4$ is such that $\tan\phi = \epsilon/(2+\epsilon)$.

The data structure for approximate L_2-neighbor queries: For each index i, let S_i denote the set of points in S with coordinates in the system $\mathcal{F}_i$. The data structure consists of a collection of range trees; the i-th range tree stores the set S_i.

Finding an approximate L_2-neighbor: Let $p \in \mathbb{R}^D$ be a query point. For each index i, use the i-th range tree to find an L_∞-neighbor $q^{(i)}$ of p in S_i. Report an L_∞-neighbor that has minimal L_2-distance to p.

It can be shown that this query algorithm reports a $(1+\epsilon)$-approximate L_2-neighbor of p. This proves Theorem 2.

3 The containment problem for boxes of constant overlap

A *box* is a D-dimensional axes-parallel cube, i.e., it is of the form $[a_1 : a_1 + \delta] \times [a_2 : a_2 + \delta] \times \ldots \times [a_D : a_D + \delta]$, for real numbers $a_1, a_2, \ldots, a_D$ and $\delta > 0$. The *center* of this box is the point $(a_1 + \delta/2, a_2 + \delta/2, \ldots, a_D + \delta/2)$.

Let S be a set of n boxes in $\mathbb{R}^D$ that are of *constant overlap*, i.e., there is an integer constant c_D such that each box B of S contains the centers of at most c_D boxes in its interior. (Here, we also count the center of B itself. Note that many boxes may have their center on the boundary of B.)

We want to store these boxes in a data structure such that for any query point $p \in \mathbb{R}^D$, we can find all boxes that contain p.

Remark: Note that we distinguish between "being contained in a box" and "being contained in the interior of a box".

In the full paper it is shown that any point $p \in \mathbb{R}^D$ is contained in at most $2^{1+D^2}c_D$ boxes of S. (The proof is technical.)

3.1 A solution based on segment trees

We start with the one-dimensional case. Let S be a set of n intervals $[a_j : b_j]$, $1 \leq j \leq n$, that are of constant overlap with constant c. We store the intervals in the leaves of a balanced binary search tree T, sorted by their right endpoints. The leaves of this tree are threaded in a doubly-linked list. Let $p \in \mathbb{R}$ be a query element. Search in T for the leftmost leaf containing a right endpoint that is at least equal to p. Starting in this leaf, walk along the leaves to the right and report all intervals encountered that contain p. Stop walking as soon as $4c$ intervals have been reported, or c intervals have been encountered that do not contain p. It is clear that this data structure solves the one-dimensional problem in $O(n)$ space with a query time of $O(\log n)$. Moreover, intervals can be inserted and deleted in $O(\log n)$ time.

We next consider the D-dimensional case, where $D \geq 2$. Let S be a set of n boxes in $\mathbb{R}^D$ that are of constant overlap with constant c_D. If $B = B_1 \times B_2 \times \ldots \times B_D$ is a box in $\mathbb{R}^D$, then B' denotes the box $B_2 \times \ldots \times B_D$ in $\mathbb{R}^{D-1}$. Similarly, S' denotes the set $\{B' : B \in S\}$. Recall that we use a similar notation for points.

The D-dimensional structure for $D \geq 2$: Let $a_1 < a_2 < \ldots < a_m$, where $m \leq 2n$, be the sorted sequence of all distinct endpoints of the intervals of the first coordinates of the boxes in S. We store the *elementary intervals* $(-\infty : a_1), [a_1 : a_1], (a_1 : a_2), [a_2 : a_2], \ldots, (a_{m-1} : a_m), [a_m : a_m], (a_m : \infty)$ in this order in the leaves of a balanced binary search tree, called the *main tree.* Each node v of this tree has associated with it an interval I_v being the union of the elementary intervals of the leaves in the subtree of v. Let S_v be the set of all boxes $B = B_1 \times B_2 \times \ldots \times B_D$ in S such that $I_v \subseteq B_1$ and $I_{f(v)} \not\subseteq B_1$. ($f(v)$ is the father of v.)

Each node v of the main tree contains (a pointer to) an *associated structure* for the set S_v: Partition this set into S_{vl}, S_{vc} and S_{vr}, consisting of those boxes of S_v whose centers lie to the left of the "vertical" slab $I_v \times \mathbb{R}^{D-1}$, in or on the boundary of this slab, and to the right of this slab, respectively.

The associated structure of v consists of three recursively defined $(D-1)$-dimensional structures for the sets S'_{vl}, S'_{vc} and S'_{vr}.

The query algorithm: Let $p = (p_1, p_2, \ldots, p_D) \in \mathbb{R}^D$ be a query point. Search in the main tree for the elementary interval that contains p_1. For each node v on the search path, recursively perform a $(D-1)$-dimensional query with the point p' in the three structures that are stored with v.

At the last level of the recursion, a one-dimensional query is performed in a binary search tree. In this tree, the algorithm stops if it has reported $2^{1+D^2}c_D$ boxes of S that contain p, or if it has encountered c_D boxes of S that do not contain p.

Lemma 2 *The query algorithm is correct.*

Proof: It is well known that the set of all boxes $B = B_1 \times \ldots \times B_D$ such that $p_1 \in B_1$ is exactly the union of all sets S_v, where v is a node on the search path to p_1. Hence, we only have to consider the nodes on this search path.

Let v be a node on the path to p_1. We have to find all boxes in S_v whose last $D-1$ intervals contain $(p_2, \ldots, p_D)$. We claim that the recursive queries in the three structures stored with v find these boxes. We know that there are at most $2^{1+D^2}c_D$ boxes that contain p. Hence, at the last level of the recursion, the query algorithm can stop as soon as it has reported this many boxes. It remains to prove that, at the last level, the query algorithm can stop if it has encountered c_D boxes that do not contain p.

Consider such a last level. That is, let $v_1, v_2, \ldots, v_{D-1}$ be nodes such that (1) $v_1 = v$, (2) v_i is a node of the main tree of one of the three structures that are stored with v_{i-1}, and (3) v_i lies on the search path to p_i.

The algorithm makes one-dimensional queries with p_D in the three structures—binary search trees—that are stored with v_{D-1}. Consider one such query. The algorithm searches for p_D. Let r be the leaf in which this search ends. Starting in r, the algorithm walks along the leaves to the right. During this walk, it encounters boxes that do or do not contain p. Assume that in leaf s, the c_D-th box is encountered that does not contain p. We have to show that the algorithm can stop in s. That is, we must show that all leaves to the right of s store boxes that do not contain p.

Assume this is not the case. Then, there is a box $A = [a_1 : a_1 + \alpha] \times [a_2 : a_2 + \alpha] \times \ldots \times [a_D : a_D + \alpha]$ that is stored in a leaf to the right of s and that contains p. Let $B^{(j)} = [b_{j1} : b_{j1} + \beta_j] \times [b_{j2} : b_{j2} + \beta_j] \times \ldots \times [b_{jD} : b_{jD} + \beta_j]$, $1 \leq j \leq c_D$, be the encountered boxes between r and s that do not contain p. In the full paper, we prove that A contains the centers of all these boxes in its interior. This is a contradiction. ∎

Theorem 4 *Let S be a set of n boxes in $\mathbb{R}^D$ of constant overlap. There exists a dynamic data structure of size $O(n(\log n)^{D-1})$ such that for any point $p \in \mathbb{R}^D$, we can find all boxes of S that contain p in $O((\log n)^{D-1} \log\log n)$ time. This data structure has an amortized update time of $O((\log n)^{D-1} \log\log n)$.*

3.2 A solution based on skewer trees

In this section, we give a linear space solution to the box containment problem. This solution is based on skewer trees, introduced by Edelsbrunner et al. [4].

Let S be a set of n boxes in $\mathbb{R}^D$ that are of constant overlap with constant c_D. For $D = 1$, the data structure is the same as in the previous subsection.

The D-dimensional structure for $D \geq 2$: If S is empty, then the data structure is also empty. Assume that S is non-empty. Let γ be the median of the set $\{(a_1+b_1)/2 : [a_1 : b_1] \times [a_2 : b_2] \times \ldots \times [a_D : b_D] \in S\}$, and let ϵ be the largest element that is less than γ in the set of all elements a_1, $(a_1 + b_1)/2$ and b_1, where $[a_1 : b_1] \times \ldots \times [a_D : b_D]$ ranges over S. Let $\gamma_1 := \gamma - \epsilon/2$ and let σ be the hyperplane in $\mathbb{R}^D$ with equation $x_1 = \gamma_1$.

Let $S_<$, S_0 and $S_>$ be the set of boxes $[a_1 : b_1] \times \ldots \times [a_D : b_D]$ in S such that $b_1 < \gamma_1$, $a_1 \leq \gamma_1 \leq b_1$ and $\gamma_1 < a_1$, respectively. The D-dimensional data structure for the set S is an augmented binary search tree—called the *main tree*—having the following form:

1. The root contains the hyperplane σ.

2. The root contains pointers to its left and right sons, which are D-dimensional structures for the sets $S_<$ and $S_>$, respectively.

3. The root contains (a pointer to) an *associated structure* for the set S_0: Partition this set into S_{0l} and S_{0r}, consisting of those boxes in S_0 whose centers lie to the left of the hyperplane σ, and to the right of σ, respectively. The associated structure of the root consists of two $(D-1)$-dimensional structures for the sets S'_{0l} and S'_{0r}.

The query algorithm and its analysis is similar to that of the previous subsection. The details can be found in the full paper. (See also Smid [11].)

Theorem 5 *Let S be a set of n boxes in $\mathbb{R}^D$ of constant overlap. There exists a data structure of size $O(n)$ such that for any point $p \in \mathbb{R}^D$, we can find all boxes of S that contain p in $O((\log n)^{D-1})$ time. This static data structure can be built in $O(n \log n)$ time.*

For the dynamic version of the problem, the query time is $O((\log n)^{D-1} \log\log n)$ and the size of the structure is $O(n)$. Boxes can be inserted and deleted in amortized time $O((\log n)^2 \log\log n)$.

4 Maintaining the closest pair

In this section, we apply the results obtained so far to maintain a closest pair of a point set under insertions and deletions. Let S be a set of n points in $\mathbb{R}^D$ and let $1 \leq t \leq \infty$ be a real number. We denote the L_t-distance between any two points p and q in $\mathbb{R}^D$ by $d(p,q)$. The pair $P, Q \in S$ is called a *closest pair* of S if $d(P,Q) = \min\{d(p,q) : p, q \in S, p \neq q\}$.

We introduce the following notations. For any point $p \in \mathbb{R}^D$, $box(p)$ denotes the smallest box centered at p that contains at least $(2D+2)^D$ points of $S \setminus \{p\}$. In other words, the side length of $box(p)$ is twice the L_∞-distance between p and its $(1 + (2D+2)^D)$-th resp. $(2D+2)^D$-th L_∞-neighbor, if $p \in S$ resp. $p \notin S$.

Let $N(p)$ be the set of points of $S \setminus \{p\}$ that are contained in the interior of $box(p)$. Note that $N(p)$ has size less than $(2D+2)^D$. In fact, $N(p)$ may even be empty.

Lemma 3 *The set $\{(p,q) : p \in S, q \in N(p)\}$ contains a closest pair of S.*

Proof: Let (P,Q) be a closest pair of S. We have to show that $Q \in N(P)$. Assume this is not the case. Let δ be the side length of $box(P)$. Since Q lies outside or on the boundary of this box, we have $d(P,Q) \geq \delta/2$.

Partition $box(P)$ into $(2D+2)^D$ subboxes with sides of length $\delta/(2D+2)$. Since $box(P)$ contains at least $1 + (2D+2)^D$ points of S, one of these subboxes contains at least two points. These two points have distance at most $D \cdot \delta/(2D+2) < \delta/2$, which is a contradiction, because (P,Q) is a closest pair of S. ∎

The set $\{box(p) : p \in S\}$ is of constant overlap: each box contains the centers of at most $(2D+2)^D$ boxes in its interior. These centers are precisely the points of $N(p) \cup \{p\}$. This fact and the above lemma suggest the following data structure.

The closest pair data structure:

1. The points of S are stored in a range tree.

2. The distances of the multiset $\{d(p,q) : p \in S, q \in N(p)\}$ are stored in a heap. With each distance, we store the corresponding pair of points. (Note that both $d(p,q)$ and $d(q,p)$ may occur in the heap.)

3. The points of S are stored in a dictionary. With each point p, we store a list containing the elements of $N(p)$. For convenience, we also call this list $N(p)$. With each point q in $N(p)$, we store a pointer to the occurrence of $d(p,q)$ in the heap.

4. The set $\{box(p) : p \in S\}$ is stored in the dynamic data structure of Theorem 4. This structure is called the *box tree*.

It follows from Lemma 3 that the pair of points that is stored with the minimal element of the heap is a closest pair of S. The update algorithms are rather straightforward. We only give the insertion algorithm.

The insertion algorithm: Let $p \in \mathbb{R}^D$ be the point to be inserted. Assume $p \notin S$.

1. Using the range tree, find the $(2D+2)^D$ L_∞-neighbors of p in S. The point among these neighbors having maximal L_∞-distance to p determines $box(p)$. The neighbors that are contained in the interior of $box(p)$ form the list $N(p)$.

2. Insert p into the range tree and insert the distances $d(p,q)$, $q \in N(p)$ into the heap. Then, insert p—together with the list $N(p)$—into the dictionary. With each point q in $N(p)$, store a pointer to $d(p,q)$ in the heap. Finally, insert $box(p)$ into the box tree.

3. Using the box tree, find all boxes that contain p. For each reported element $box(q)$, $q \neq p$, that contains p in its interior, do the following:

3.a. Search in the dictionary for q. Insert p into $N(q)$, insert $d(q,p)$ into the heap and store with p a pointer to $d(q,p)$.

3.b. If $N(q)$ has size less than $(2D+2)^D$, then the insertion algorithm is completed. Otherwise, if $N(q)$ has size $(2D+2)^D$, let $r_1, \ldots, r_l$ be all points in $N(q)$ that have maximal L_∞-distance to q. For each $1 \leq i \leq l$, delete r_i from $N(q)$ and delete $d(q,r_i)$ from the heap. Finally, delete $box(q)$ from the box tree and insert the box centered at q having r_1 on its boundary as the new $box(q)$.

During the update algorithms, we perform a constant number of query and update operations in the range tree, the box tree, the heap and the dictionary. Therefore, Theorems 1 and 4 imply:

Theorem 6 *Let S be a set of n points in $\mathbb{R}^D$ and let $1 \leq t \leq \infty$. There exists a data structure of size $O(n(\log n)^{D-1})$ that maintains an L_t-closest pair of S in $O((\log n)^{D-1} \log\log n)$ amortized time per insertion and deletion.*

Consider again our data structure. The box $box(p)$ that is associated with point p contains an L_∞-neighbor of p in $S \setminus \{p\}$. Therefore, the data structure can easily be adapted such that it maintains for each point in S its L_∞-neighbor. This proves Theorem 3.

5 A transformation for reducing the space complexity

The closest pair data structure of the previous section uses more than linear space. This raises the question if the same update time can be obtained using only linear space. Note that we have a linear space solution for maintaining the set $\{box(p) : p \in S\}$. (See Theorem 5.) For the L_∞-neighbor problem, however, no linear space solution having polylogarithmic query and update times is known. Hence, in order to reduce the space complexity, we should avoid using the range tree.

We will give a transformation that, given *any* dynamic closest pair data structure having more than linear size, produces another dynamic closest pair structure that uses less space.

Let DS be a data structure that maintains a closest pair in a set of n points in $\mathbb{R}^D$ under insertions and deletions. Let $S(n)$ and $U(n)$ denote the size and update time of DS, respectively. We assume that $S(n)$ and $U(n)$ are non-decreasing. Let $f(n)$ be a non-decreasing integer function such that $1 \leq f(n) \leq n/2$.

Let $S \subseteq \mathbb{R}^D$ be the current set of points. The cardinality of S is denoted by n. Our transformed data structure will be completely rebuilt after a sufficiently long sequence of updates. Let S_0 be the set of points at the moment of the most recent rebuilding and let n_0 be its size at that moment.

As in the previous section, for each $p \in \mathbb{R}^D$, $box(p)$ denotes the smallest box centered at p that contains at least $(2D+2)^D$ points of $S \setminus \{p\}$. The set of all points of $S \setminus \{p\}$ that are in the interior of this box is denoted by $N(p)$. If $p \in S_0$, then $box_0(p)$ denotes the smallest box centered at p that contains at least $(2D+2)^D$ points of $S_0 \setminus \{p\}$.

The transformed closest pair data structure:

1. The set S is partitioned into sets A and B such that $A \subseteq \{p \in S : p \in S_0 \wedge box(p) \subseteq box_0(p)\}$.

2. The distances of the multiset $\{d(p,q) : p \in A, q \in N(p)\}$ are stored in a heap. With each distance, we store the corresponding pair of points.

3. The boxes of the set $\{box_0(p) : p \in S_0\}$ are stored in a list, called the *box list*. With each element $box_0(p)$ in this list, we store a bit having value *true* if and only if $p \in A$. Moreover, if $p \in A$, we store with $box_0(p)$ the box $box(p)$.

4. The boxes of the set $\{box_0(p) : p \in S_0\}$ are stored in the *static* data structure of Theorem 5. This structure is called the *box tree.* With each box in this structure, we store a pointer to its occurrence in the box list.

5. The points of S are stored in a dictionary. With each point p, we store a bit indicating whether p belongs to A or B. If $p \in A$, then we store with p

5.a. a pointer to the occurrence of $box_0(p)$ in the box list, and

5.b. a list containing the elements of $N(p)$. For convenience, we also call this list $N(p)$. With each point q in $N(p)$, we store a pointer to the occurrence of $d(p,q)$ in the heap.

6. The set B is stored in the dynamic data structure DS, called the *B-structure.*

Lemma 4 *Let δ be the minimal distance stored in the heap and let δ' be the distance of a closest pair in B. Then, $\min(\delta, \delta')$ is the distance of a closest pair in the set S.*

Proof: Let (P,Q) be a closest pair in S. We distinguish two cases.

Case 1: At least one of P and Q is contained in A. Assume that $P \in A$. Since $box(P)$ contains at least $1 + (2D+2)^D$ points of S, Q is contained in the interior of this box! Hence, $Q \in N(P)$ and the distance $d(P,Q)$ is stored in the heap. Therefore, $\delta = d(P,Q)$. Clearly, $d(P,Q) \leq \delta'$. This proves that $d(P,Q) = \min(\delta, \delta')$.

Case 2: Both P and Q are contained in B. Then $\delta' = d(P,Q)$ and $d(P,Q) \leq \delta$. ■

Initialization: At the moment of initialization, $S = S_0 = A$ and $B = \emptyset$. Using Vaidya's algorithm [14], compute for each point p in S its $1 + (2D+2)^D$ L_∞-neighbors. The point among these neighbors having maximal L_∞-distance to p determines $box(p) = box_0(p)$. The neighbors (except p itself) that are contained in the interior of this box form the list $N(p)$. It is clear how the rest of the data structure can be built.

The insertion algorithm: Let $p \in \mathbb{R}^D$ be the point to be inserted.

1. Insert p into the dictionary and store with p a bit saying that p belongs to B. Then, insert p into the B-structure.

2. Using the box tree, find all boxes that contain p. For each reported element $box_0(q)$, follow the pointer to its occurrence in the box list. If the bit of $box_0(q)$ has value *true*, then check if p is contained in the interior of $box(q)$. If so, do the following:

2.a. Search in the dictionary for q. Insert p into $N(q)$, insert $d(q,p)$ into the heap and store with p a pointer to $d(q,p)$.

2.b. If $N(q)$ has size less than $(2D+2)^D$, then the insertion algorithm is completed. Otherwise, let $r_1, \ldots, r_l$ be all points of $N(q)$ that are at maximal L_∞-distance from q. For each $1 \leq i \leq l$, delete r_i from $N(q)$ and delete $d(q, r_i)$ from the heap. Finally, replace $box(q)$—which is stored with $box_0(q)$ in the box list—by the box centered at q having r_1 on its boundary, being the new $box(q)$.

Note that since $A \subseteq \{p \in S : p \in S_0 \wedge box(p) \subseteq box_0(p)\}$, all boxes $box(q)$, $q \in A$, that contain p are found in Step 2.

The deletion algorithm: Let $p \in S$ be the point to be deleted.

1. Search for p in the dictionary. If $p \in B$, then delete p from this dictionary and from the B-structure. Otherwise, if $p \in A$, follow the pointer to $box_0(p)$ in the box list and set its bit to *false.* Moreover, for each point q in $N(p)$, delete the distance $d(p,q)$ from the heap. Then, delete p from the dictionary.

2. Using the box tree, find all boxes that contain p. For each reported element $box_0(q)$, follow the pointer to its occurrence in the box list. If the bit of $box_0(q)$ has value *true*, then check if p is contained in $box(q)$. If so, do the following: Set the bit of $box_0(q)$ to *false.* Search in the dictionary for q. For each point r in $N(q)$, delete $d(q,r)$ from the heap. Then, delete the pointer from q to $box_0(q)$, delete the list $N(q)$, and store with q a bit saying that it belongs to B. Finally, insert q into the B-structure.

Rebuild: Recall that n_0 is the size of S at the moment we initialize the structure. After $f(n_0)$ updates have been performed, we discard the entire structure and initialize a new data structure for the current S.

Theorem 7 *Let DS be any data structure for the dynamic closest pair problem. Let $S(n)$ and $U(n)$ denote the size and update time of DS, respectively. Let $1 \leq f(n) \leq n/2$ be a non-decreasing integer function. We can transform DS into another data structure for the dynamic closest pair problem having size $O(n + S(f(n)))$, and an amortized update* *time of* $O((\log n)^{D-1} + U(f(n)) + (n \log n)/f(n))$.

If we apply this theorem twice (for the case $D \geq 3$), respectively k times (for the case $D = 2$), with appropriate choices for the function $f(n)$, then we get the results given in the last three lines of Table 1.

References

[1] S. Arya and D.M. Mount. *Approximate nearest neighbor queries in fixed dimensions.* Proc. SODA 1993, 271-280.

[2] S. Arya, D.M. Mount, N.S. Netanyahu, R. Silverman, A. Wu. *An optimal algorithm for approximate nearest neighbor searching.* Proc. SODA 1994, pp. 573-582.

[3] K.L. Clarkson. *A randomized algorithm for closest-point queries.* SIAM J. Comput. **17** (1988), 830-847.

[4] H. Edelsbrunner, G. Haring and D. Hilbert. *Rectangular point location in d dimensions with applications.* The Computer Journal **29** (1986), 76-82.

[5] M. Golin, R. Raman, C. Schwarz and M. Smid. *Randomized data structures for the dynamic closest-pair problem.* Proc. SODA 1993, 301-310.

[6] K. Mehlhorn and S. Näher. *Dynamic fractional cascading.* Algorithmica **5** (1990), 215-241.

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

[8] J.S. Salowe. *Shallow interdistance selection and interdistance enumeration.* International Journal of Computational Geometry & Applications **2** (1992), 49-59.

[9] C. Schwarz. *Data structures and algorithms for the dynamic closest pair problem.* Ph.D. Thesis. Universität des Saarlandes, Saarbrücken, 1993.

[10] C. Schwarz, M. Smid and J. Snoeyink. *An optimal algorithm for the on-line closest pair problem.* Proc. ACM Symp. on Computational Geometry, 1992, 330-336.

[11] M. Smid. *Rectangular point location and the dynamic closest pair problem.* Proc. 2nd Annual International Symp. on Algorithms, Lecture Notes in Computer Science, Vol. 557, Springer-Verlag, Berlin, 1991, 364-374.

[12] M. Smid. *Maintaining the minimal distance of a point set in polylogarithmic time.* Discrete Comput. Geom. **7** (1992), 415-431.

[13] K.J. Supowit. *New techniques for some dynamic closest-point and farthest-point problems.* Proc. SODA 1990, 84-90.

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

[15] A.C. Yao. *On constructing minimum spanning trees in k-dimensional spaces and related problems.* SIAM J. Comput. **11** (1982), 721-736.