Closest-Point Problems in Computational Geometry

Michiel Smid

Fakultät für Informatik

Otto-von-Guericke-Universität Magdeburg

Universitätsplatz 2

D-39106 Magdeburg, Germany

E-mail: `michiel@isg.cs.uni-magdeburg.de`

September 1, 1997

Abstract

This is the preliminary version of a chapter that will appear in the *Handbook on Computational Geometry*, edited by J.-R. Sack and J. Urrutia.

A comprehensive overview is given of algorithms and data structures for proximity problems on point sets in $\mathbb{R}^D$. In particular, the closest pair problem, the exact and approximate post-office problem, and the problem of constructing spanners are discussed in detail.

Geometric proximity problems, or closest point problems, have always been a popular topic in computational geometry. These problems deal with distances in sets in $\mathbb{R}^D$. In this chapter, we will consider proximity problems on point sets. The distance metrics we consider are the Minkowski metrics defined as follows. For $\tau \geq 1$, the L_τ-distance between the points $p = (p_1, p_2, \ldots, p_D)$ and $q = (q_1, q_2, \ldots, q_D)$ is defined by

$$d_\tau(p, q) = \left(\sum_{i=1}^{D} |p_i - q_i|^\tau \right)^{1/\tau},$$

and for $\tau = \infty$, it is defined by

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

For example, if $\tau = 2$, then we have the standard Euclidean metric, and if $\tau = 1$, we have the Manhattan metric.

Throughout this chapter, we assume that the dimension D and the parameter τ, $1 \leq \tau \leq \infty$, are fixed constants. We denote the L_τ-distance between the points p and q by $d(p, q)$.

We now introduce the type of problems that will be considered in this chapter.

The closest pair problem: The simplest version of this problem is stated as follows.

Problem 1.1 *Given a set S of n points in $\mathbb{R}^D$, find a closest pair of S, i.e., two points $P, Q \in S$, such that*

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

This problem appeared for the first time in Shamos [111], one of the first papers in computational geometry. Since there are $\binom{n}{2}$ pairs of distinct points, and since the distance between two points can be computed in $O(D) = O(1)$ time, Problem 1.1 can trivially be solved in $O(n^2)$ time.

In the early years of computational geometry, researchers felt that Problem 1.1 might have a quadratic lower bound. (See Bentley [22].) Since solutions to several other geometric problems "contain" the closest pair, this would have implied a quadratic lower bound on these problems as well. Examples are computing the Voronoi diagram, Delaunay triangulation and minimum spanning tree of a point set. Hence, determining the complexity of Problem 1.1 is an important problem from a theoretical point of view.

At this moment, many algorithms are known that solve Problem 1.1 in $O(n \log n)$ time for any fixed dimension D. In fact, in certain models of computation, the problem can even be solved in linear time. We will give an overview of most of these algorithms. It is fair to say that Problem 1.1 has been solved both from the theoretical and the practical point of view.

There are some obvious generalizations of the closest pair problem that will also be treated in this chapter. Let S be a set of n points in $\mathbb{R}^D$. First, in the *all-nearest-neighbors problem*, we want to compute for each point p of S another point of S that

we are given the set S and a positive real number δ, and we have to report all pairs of points that are at distance at most δ from each other. In the *k closest pairs problem*, we are given S and an integer k, $1 \leq k \leq \binom{n}{2}$, and we have to report among all $\binom{n}{2}$ distances the k smallest ones. We will also consider the *k-point clustering problem*, in which we are given S and an integer k, $2 \leq k \leq n$, and we have to find a subset of S of size k that minimizes some closeness measure. Examples are finding k points whose diameter or enclosing circle is minimum. Finally, we consider different versions of the *dynamic closest pair problem*, in which we have to maintain the closest pair if the set S is dynamically changed by insertions and/or deletions of points.

The closest pair problem and its generalizations arise in areas like statistics, pattern recognition (see the books by Andrews [8] and Hartigan [75]), and molecular biology. As an example, a solution to the fixed-radius near neighbors problem is an essential preprocessing step in the algorithm of Heiden at al.[76] for triangulating the contact surface of a molecule. (See also Lenhof and Smid [86].)

Another application is given by Supowit [124]. He applies a dynamic closest pair data structure to give an efficient implementation of a greedy minimum weight matching algorithm. In this problem, we are given a set of n points, where n is even. Our task is to match these points into $n/2$ pairs such that the sum of the lengths of the distances of these pairs is minimized. The greedy algorithm matches a closest pair and removes these two points from the set. Then, the closest pair of the resulting set is matched, etc., until we have matched all points. In practice, this heuristic gives a good approximation to the optimal matching.

The post-office problem: The post-office problem is due to Knuth [85]. It is stated as follows.

Problem 1.2 *Given a set S of n points in $\mathbb{R}^D$, store it in a data structure such that for any query point $p \in \mathbb{R}^D$, we can efficiently find its* nearest *neighbor, i.e., a point $p^* \in S$ that is closest to p,*

$$d(p, p^*) = \min\{d(p, q) : q \in S\}.$$

The planar version of the post-office problem has been solved optimally, i.e., with $O(\log n)$ query time and $O(n)$ space. In the higher-dimensional case, however, it seems impossible to give a data structure of size $O(n \log^{O(1)} n)$ that answers queries in polylogarithmic time. Moreover, it is an open problem if there exists a dynamic data structure for the planar problem having size $O(n \log^{O(1)} n)$ and polylogarithmic query and update times.

Therefore, we will consider two weaker versions of the problem. First, we consider the post-office problem for "simple" metrics, such as the L_1- or L_∞-metric. Second, we consider the variant in which we do not have to find the *exact* nearest neighbor p^* of the query point p, but are satisfied with an *approximate* neighbor, i.e., a point $q \in S$ such that $d(p, q) \leq (1 + \epsilon)\, d(p, p^*)$, for some positive constant ϵ. As we will see, for both weaker versions, dynamic data structures can be designed having polylogarithmic query and update times and having a size that is close to linear.

data compression. An important example is vector quantization, which is used for compressing speech and images. (See Arya and Mount [10] and the references given there.) Assume we take samples from a signal, and group them into vectors of length D. Also assume that we have constructed a set S of "typical" code vectors in $\mathbb{R}^D$, based on a training set of vectors. Then each new vector is encoded by the index of its nearest neighbor in S. If there are $n = 2^{rD}$ code vectors, then we need rD bits to encode such a new vector. Hence, the rate of this vector quantizer is equal to r. If we fix r, and let D grow, then the performance of this quantizer increases. However, then the number of points n also increases, and we have to solve the post-office problem for a large dimension D. Hence, it is important to have efficient (exact or approximate) data structures for this problem.

Related problems: In the final part of this chapter, we will mention several related results for proximity problems. In particular, we consider the problem of approximating the complete Euclidean graph.

Let S be a set of points in $\mathbb{R}^D$. Consider any graph having the points of S as its vertices. The *weight* of an edge (p, q) in this graph is defined as the distance between p and q. The *weight of a path* in the graph is defined as the sum of the weights of all edges on the path.

Definition 1.1 *Let $t > 1$ be any real constant. A graph having the points of S as its vertices is called a t-spanner if for every pair p, q of points in S there is a path in the graph between p and q of weight at most t times the distance between p and q.*

Spanners were introduced to computational geometry by Chew [39]. Since then, many papers have been published on the problem of efficiently constructing spanners that have one or more additional properties. We will give an overview of the most important results in this area.

Spanners in which the degree of each vertex is bounded by a constant can be used as a data structure for the k closest pairs problem, or the fixed-radius near neighbors problem. (See Arya and Smid [15].) To be more precise, given such a spanner and a query integer k (resp. a query value $\delta \in \mathbb{R}$), we can efficiently enumerate the k closest pairs (resp. all pairs that have distance at most δ.)

We introduce some notation that will be used throughout this chapter. If S is a set of points in $\mathbb{R}^D$, then $d(S)$ denotes the minimum distance in S, i.e., the distance of a closest pair. If $p \in \mathbb{R}^D$, then $d(p, S)$ denotes the minimum distance from p to any point of $S \setminus \{p\}$. Finally, if A and B are sets, then $d(A, B)$ denotes the minimum distance between any point of A and any point point of B.

If δ is a positive real number, then a δ-*grid* is the subdivision of $\mathbb{R}^D$ into axes-parallel cells with sides of length δ, defined by the hyperplanes $x_i = j \cdot \delta$, where $1 \leq i \leq D$, and j ranges over the integers. The *neighborhood of a cell* is defined as the cell itself plus the $3^D - 1$ cells that border it. The *neighborhood of a point p* is defined as the neighborhood of the cell that contains p.

3

implies that all constant factors that appear in Big-Oh notations depend on D, unless stated otherwise. In general, such factors are of the form $(cD)^D$, for some constant c.

2 The static closest pair problem

As mentioned already, many algorithms have been proposed for solving Problem 1.1, i.e., the static closest pair problem. In this section, we give an overview of these algorithms.

2.1 Preliminary remarks

It turns out that the complexity of the closest pair problem heavily depends on the machine model. Most algorithms in this chapter can be implemented in the *algebraic computation tree model*. (See Ben-Or [19].) In this model, the operations $+, -, *, /$ and $\sqrt[k]{x}$, where k comes from a finite subset of the positive integers, take unit time. Note that these operations perform *exact arithmetic* on real numbers. Also, *comparisons* of real numbers takes unit time. We remark that *indirect addressing* is not possible in the algebraic computation tree model.

The closest pair problem has an $\Omega(n \log n)$ lower bound in the algebraic computation tree model. Consider the following ϵ-*closeness problem*: Given $n + 1$ real numbers $x_1, x_2, \ldots, x_n$ and $\epsilon > 0$, decide if there are $i \neq j$ such that $|x_i - x_j| < \epsilon$. Using the lower bound technique of Ben-Or [19], it can be shown that the latter problem takes $\Omega(n \log n)$ time in this model. (See also Preparata and Shamos [99].) This implies the same lower bound for the closest pair problem. In fact, Agarwal et al. [3] even prove an $\Omega(n \log n)$ lower bound for the closest pair problem, if the n points are given as the vertices of a simple polygon.

In this section, we will see several algorithms that solve the closest pair problem in $O(n \log n)$ time. If we use a more powerful machine model, then we can design faster algorithms. More precisely, we have to add *randomization*, the non-algebraic *floor function*, and the power of indirect addressing. (See Section 2.4.)

We close this section with an important *sparseness* lemma which is heavily used in all closest pair algorithms. Basically the lemma says that any box having side lengths that are small compared to the minimum distance of a point set contains only few points of this set.

Lemma 2.1 *Let S be a set of n points in $\mathrm{I\!R}^D$, let δ be the distance of a closest pair in S—in the L_τ-metric—and let c be a positive integer. Then any D-dimensional cube having sides of length $c\delta$ contains at most $(cD + c)^D$ points of S.*

Proof: The proof is by contradiction. Assume the cube contains more than $(cD + c)^D$ points of S. Partition the cube into $(cD + c)^D$ subcubes with sides of length $c\delta/(cD + c)$. Then one of these subcubes contains at least two points of S. These points, however, have L_τ-distance at most the L_τ-diameter of a subcube, which is at most equal to its L_1-diameter, which in turn is equal to $D \cdot c\delta/(cD + c) < \delta$. This is clearly a contradiction. ∎

tree model

2.2.1 An algorithm based on the Voronoi diagram

The first optimal algorithm for solving the planar version of the closest pair problem is due to Shamos [111] and Shamos and Hoey [112]. Their algorithm is as follows. In $O(n \log n)$ time, compute the Voronoi diagram of S. Then, for each edge e of this diagram, compute the distance between the two points whose Voronoi regions share e. This takes only linear time. The smallest distance found in this way determines the closest pair of S. For details, we refer to [16, 111, 112] and the chapter on Voronoi diagrams in this handbook.

2.2.2 A divide-and-conquer algorithm

Bentley and Shamos [23] were the first who gave an optimal $O(n \log n)$-time algorithm for the closest pair problem in any dimension $D \geq 2$. Their algorithm uses the *divide-and-conquer paradigm.* (According to Bentley [22], the planar version of this algorithm is due to Shamos, and the idea of using divide-and-conquer was suggested to him by H.R. Strong.)

For simplicity, let us first consider the planar case. So, let S be a set of n points in the plane. Compute the median m of the x-coordinates of the points of S. Partition S into two subsets A and B of (almost) equal size such that all points of A (resp. B) are on or to the left (resp. on or to the right) of the vertical line $x = m$. Using the same algorithm recursively, solve the closest pair problem for the sets A and B separately. Let δ_A (resp. δ_B) be the minimum distance in A (resp. B), and let δ be the smaller of these two numbers. To compute the closest pair in the overall set S, it remains to find among all pairs $(a, b) \in A \times B$ that have distance less than δ the one having minimum distance. Let A' (resp. B') be the set of those points of A (resp. B) that are to the right (resp. left) of the vertical line $x = m - \delta$ (resp. $x = m + \delta$). Clearly, we only have to consider points of A' and B'. Sort the points of A' and B' according to their y-coordinates. Then, scan along these points. For each point $p = (p_x, p_y)$ of A' (resp. B'), compare p with all points of B' (resp. A') whose y-coordinates are between $p_y - \delta$ and $p_y + \delta$. If there is a pair of points in $A \times B$ that has distance less than δ, then during the scan we will find the pair in $A \times B$ having minimum distance. This pair is the closest pair in the set S. Otherwise, all pairs that are encountered in this "merge step" have distance larger than δ and, therefore, δ is the minimum distance in S.

It is not difficult to see that this algorithm correctly solves the closest pair problem. Let $T(n)$ denote the running time on a set of n points. Lemma 2.1 implies that in the merge step each point of $A' \cup B'$ is compared to at most a constant number of points. Therefore, $T(n)$ satisfies the recurrence relation $T(n) = 2\,T(n/2) + O(n \log n)$. It follows that $T(n) = O(n \log^2 n)$. The merge step consists of two parts. First, there is a sorting step, taking $O(n \log n)$ time. In the second step, we scan along the points of A' and B'. This takes only linear time.

The algorithm can be improved by making a *presorting step.* At the start of the

median m, we walk along the points of S in sorted order, and obtain the sets A and B, both sorted by their y-coordinates. Similarly, we obtain the points of A' and B' in sorted order. In this way, the merge step only takes $O(n)$ time, and the running time $T(n)$—on a set of n points sorted by their y-coordinates—satisfies the recurrence $T(n) = 2\,T(n/2) + O(n)$, which solves to $T(n) = O(n\log n)$. It is easy to see that the entire algorithm uses linear space.

It turns out to be non-trivial to extend the given algorithm to higher-dimensional space. Nevertheless, Bentley and Shamos [23] and Bentley in his Ph.D. Thesis [20], showed that the divide-and-conquer approach does lead to an optimal $O(n\log n)$-time algorithm for any fixed dimension D. (See also Bentley [22].) Let S be a set of n points in $\mathbb{R}^D$. In a presorting step, sort the points of S on *all* coordinates, in $O(D\,n\log n) = O(n\log n)$ time. Then, in $O(n)$ time, compute a hyperplane h perpendicular to one of the D coordinate axes, such that (i) both subsets A and B on either side of h contain at least $n/(4D)$ points of S, and (ii) if δ denotes the smallest of the minimum distances δ_A and δ_B of A and B, respectively, then each of the two slabs of width δ around h contains only $O(n^{1-1/D})$ points. (Note that δ is not known when the hyperplane h is computed.) Given h, recursively compute δ_A and δ_B. Let A' (resp. B') be the points of A (resp. B) that are on either side of h and that are at distance at most δ from h. Then, in the merge step, we have to find among all pairs $(a,b) \in A' \times B'$ that have distance less than δ, a pair with minimum distance. Note that the sets A' and B' are sparse, in the sense that every cube with sides of length δ contains at most a constant number of points. Bentley solves the merge step by using again a divide-and-conquer algorithm. If n_0 denotes the size of the set $A' \cup B'$, then this step takes $O(n_0 \log n_0)$ time. Note that $n_0 = O(n^{1-1/D})$. Therefore, the running time $T(n)$—after the presorting step—of the entire algorithm for computing the closest pair in S satisfies the recurrence $T(n) = 2\,T(n/2) + O(n)$, which solves to $T(n) = O(n\log n)$.

2.2.3 A plane sweep algorithm

Hinrichs, Nievergelt and Schorn [77] gave an elegant algorithm for the planar case that is based on the *plane sweep paradigm*. First, sort the points of S by their x-coordinates. Then, sweep a vertical line over the plane, from left to right. During this sweep, we maintain as an invariant that we have computed the closest pair among all points to the left of the sweep line. Also, if δ is the distance of this closest pair, then all points that are at distance at most δ to the left of the sweep line are stored in the leaves of a balanced binary search tree, sorted by their y-coordinates. The leaves of this tree are linked by pointers. If the sweep line moves to the next point $p = (p_x, p_y)$, then the following actions are performed. Search in the tree for p_y. Then follow pointers that link the leaves, and find all points whose y-coordinates are between $p_y - \delta$ and $p_y + \delta$. (By Lemma 2.1, there are at most a constant number of such points.) Compare the distances between p and all points found in this way, and, if necessary update δ. Then, insert p into the tree, and delete from the tree all points having distance larger than δ from the new position of the sweep line. It is easy to see that this algorithm correctly solves the closest pair problem. Its running time is bounded by $O(n\log n)$.

In the previous section, we saw several algorithms that solve the closest pair problem. For the planar case, these algorithms are easily implementable. Moreover, they are fast in practice. We saw only one algorithm that is optimal for any fixed dimension; that of Bentley. This algorithm, however, is complicated: it uses multi-dimensional divide-and-conquer, and the merge step itself is also based on multi-dimensional divide-and-conquer.

Lenhof and Smid [87] gave a practical algorithm for solving the closest pair problem in any fixed dimension D. The algorithm is based on the fact that it suffices to construct a grid such that (i) at least one cell contains at least 2 points and (ii) each cell contains at most 2^D points. Let δ be the side length of the cells in such a grid. Property (i) implies that the minimum distance in S is at most equal to $D\delta$. Hence, given this δ-grid, it suffices to compare each point p of S with those points that are contained in p's cell or in any one of the $(2D + 1)^D - 1$ neighboring cells. Property (ii) implies that in this way, each point is compared to at most a constant number of other points. Therefore, given the δ-grid, we can compute the closest pair in $O(n \log n)$ time, or even in linear time if we represent the grid in an appropriate way.

The main problem is how to compute a grid size such that (i) and (ii) holds. First, why does such a grid size exist? Let δ^* be the side length of a smallest axes-parallel D-dimensional cube that contains at least $2^D + 1$ points of S. Then any δ-grid, where $\delta \geq \delta^*$, has a cell containing at least two points of S. Also, any δ-grid, where $\delta \leq \delta^*$, does not have any cell containing more than 2^D points of S. In particular, δ^* defines a grid for which (i) and (ii) hold.

Note that δ^* is contained in the set of all differences $|p_i - q_i|$, where $1 \leq i \leq D$ and p and q range over all points of S. Therefore, we do a binary search in this set. Of course, this set has size $\Theta(n^2)$. We can maintain it, however, implicitly, in only linear space, using a technique that is similar to one developed by Johnson and Mizoguchi [80]. For $1 \leq i \leq D$, let A_i be an array that contains the i-th coordinates of the points in sorted order. With each element $A_i[j]$, we store two integers l_{ij} and r_{ij} such that the following *invariant* is maintained: δ^* is contained in the set of all differences

$$d_{ij} := A_i[j] - A_i[j'], \text{ where } l_{ij} \leq j' \leq r_{ij}. \tag{1}$$

Initially, we have $l_{ij} = j + 1$ and $r_{ij} = n$. Now we make a sequence of $O(\log n)$ iterations. In one iteration, we do the following. For each $1 \leq i \leq D$, and each $1 \leq j < n$, such that $l_{ij} \leq r_{ij}$ take the difference

$$A_i[\lfloor (l_{ij} + r_{ij})/2 \rfloor] - A_i[j]$$

and give it weight $r_{ij} - l_{ij} + 1$. This gives at most $D(n - 1)$ weighted differences. In $O(n)$ time, we compute the weighted median δ of these differences. Now, we construct a δ-grid, and compute the maximum number M of points that are contained in any cell of this grid. There are three cases. First, if $2 \leq M \leq 2^D$, then we have found a correct grid size, and we can stop. Second, assume that $M > 2^D$. Then $\delta > \delta^*$. Therefore, for each i and j such that $d_{ij} \geq \delta$, we can assign $r_{ij} := \lfloor (l_{ij} + r_{ij})/2 \rfloor - 1$. This guarantees that the invariant still holds. Finally, if $M < 2$, then we know that $\delta < \delta^*$. In this

invariant is correctly maintained in this way.

Since we choose the weighted median, the set of all differences in (1) is reduced by a factor of at least one quarter. Therefore, in a logarithmic number of iterations, the algorithm has found a correct grid size. (Note that the algorithm does not necessarily find δ^*. Any grid size for which (i) and (ii) hold is good for our purposes.) After a presorting step, each iteration can be performed in linear time. Therefore, the entire algorithm takes $O(n \log n)$ time.

There are some remarks to be made about the machine model used. We need indirect addressing to access the entries of the arrays A_i. It is not clear if this can be avoided.

In the algorithm as presented, we also need the non-algebraic floor function in order to count the maximum number of points in any grid cell. (We need the floor function for determining the grid cell containing a given point.) This can be avoided by using a *degraded grid* that has basically the same properties as a standard grid. We can build and search in a degraded grid, however, without using the floor-function.

2.3.1 The degraded grid

To give an intuitive idea, in a standard δ-grid, we divide D-space into slabs of width δ. The grid is then defined by fixing an arbitrary point of $\mathbb{R}^D$ to be a lattice point of the grid. So, if e.g. $(0, \ldots, 0)$ is a lattice point, then for $1 \leq i \leq D$, a slab along the i-th axis consists of the set of all points in D-space that have their i-th coordinates between $j\delta$ and $(j + 1)\delta$ for some integer j. In a degraded δ-grid, we also have slabs. The difference is that slabs do not necessarily start and end at multiples of δ. Moreover, slabs have width at least δ, and slabs that contain points of S have width exactly δ. That is, while a δ-grid may be defined independently of the point set by fixing an arbitrary point of $\mathbb{R}^D$ to be a lattice point, the degraded δ-grid is defined in terms of the point set stored in it. We give a formal definition, treating the case $D = 1$ first.

Definition 2.1 *Let S be a set of n real numbers and let δ be a positive real number. Let $a_1, a_2, \ldots, a_l$ be a sequence of real numbers such that*

1. *for all $1 \leq j < l$, $a_{j+1} \geq a_j + \delta$,*

2. *for all $p \in S$, $a_1 \leq p < a_l$,*

3. *for all $1 \leq j < l$, if there is a point $p \in S$ such that $a_j \leq p < a_{j+1}$, then $a_{j+1} = a_j + \delta$.*

The collection of intervals $[a_j : a_{j+1})$, $1 \leq j < l$, is called a one-dimensional degraded δ-grid for S.

To construct a one-dimensional degraded δ-grid, sort the elements of S. Let $p_1 \leq p_2 \leq \ldots \leq p_n$ be the sorted sequence. Let $a_1 := p_1$. Let $j \geq 1$, and assume that $a_1, \ldots, a_j$ are defined already.

set $a_{j+1} := a_j + \delta$. Otherwise, we set a_{j+1} to the value of the smallest element in S that is larger than a_j. This construction stops if we have visited all elements of S.

We extend the definition of a degraded grid to higher dimensions.

Definition 2.2 *Let S be a set of n points in $\mathbb{R}^D$ and let δ be a positive real number. For $1 \leq i \leq D$, let S_i be the set of i-th coordinates of the points in S. Let*

$$[a_{ij} : a_{i,j+1}), \quad 1 \leq j < l_i,$$

be a one-dimensional degraded δ-grid for the set S_i. The collection of D-dimensional cells

$$\prod_{i=1}^{d} [a_{ij_i} : a_{i,j_i+1}), \ where \ 1 \leq j_i < l_i,$$

is called a D-dimensional degraded δ-grid for S.

See Figure 1 for an example. The following lemma follows immediately.
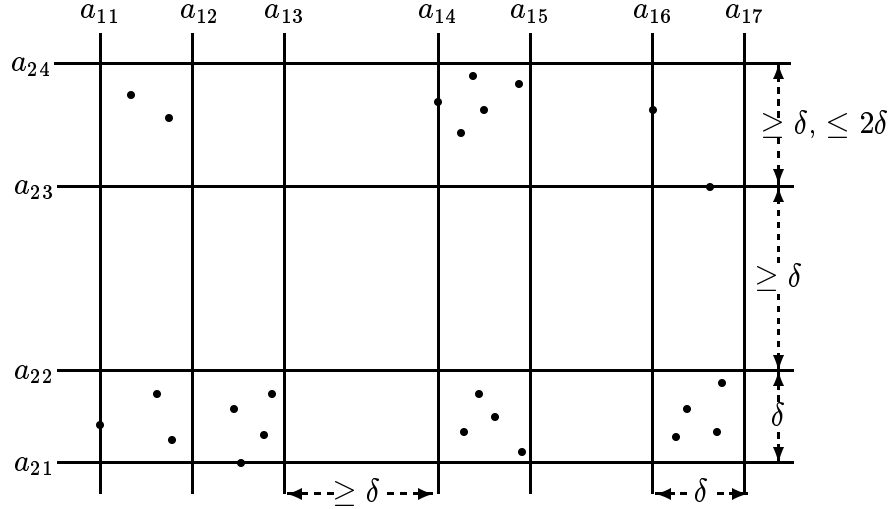
Figure 1: Example of a degraded δ-grid.

Lemma 2.2 *Let p be a point of S and let B be the cell in the degraded δ-grid for S that contains p. Let c be an integer. All points of S that are within distance $c\delta$ from p are contained in B and in the $(2c + 1)^D - 1$ cells that surround B.*

We now give a simple algorithm for constructing a D-dimensional degraded δ-grid. Assume the points of S are stored in a one-dimensional array $\mathcal{S}$. For each $1 \leq i \leq D$, sort the elements of S_i. Give each element in S_i a pointer to its occurrence in $\mathcal{S}$.

For each $1 \leq i \leq D$, construct a one-dimensional degraded δ-grid $[a_{ij} : a_{i,j+1})$, $1 \leq j < l_i$, for the set S_i using the algorithm given above. During this construction,

9

that $a_{ij} \leq p_i < a_{i,j+1}$, follow the pointer to $\mathcal{S}$. Store with the point p in $\mathcal{S}$ the numbers a_{ij} and j.

At the end, each point in $\mathcal{S}$ stores with it two vectors of length D. If point p has vectors $(b_1, b_2, \ldots, b_D)$ and $(j_1, j_2, \ldots, j_D)$, then p is contained in the cell with lower-left corner $(b_1, b_2, \ldots, b_D)$. This cell is part of the j_i-th δ-slab along the i-th axis.

These vectors implicitly define the degraded δ-grid. Note that each j_i is an integer in the range from 1 to n. Hence, we can sort the vectors $(j_1, j_2, \ldots, j_D)$ in $O(n)$ time by using radix-sort. This gives the non-empty cells of the degraded grid, sorted in lexicographical order.

2.4 Randomized algorithms

In this section, we will give algorithms that solve the closest pair problem in $o(n \log n)$ time. Of course, these algorithms are implemented on a machine model that is more powerful than algebraic computation trees. Most algorithms of this section use randomization, the floor function, and indirect addressing.

For most of these algorithms, the basic approach is similar to that of Section 2.3. In a first step, a grid is constructed such that (i) the closest pair is either contained in one grid cell or in two neighboring cells and (ii) the total number of pairs (p, q), where $p \in S$ and q is contained in p's cell or in any one of the neighboring cells is bounded (with high probability, or in the worst case) by $O(n)$. Then, in the second step, we compute the closest pair by taking the minimum distance among all these pairs (p, q). Using perfect hashing [66], this second step takes only $O(n)$ expected time. Hence, the main problem is how to find a good grid size.

2.4.1 Rabin's algorithm

The oldest randomized closest pair algorithm dates back to 1976 and is due to Rabin [101]. (In fact, [101] is considered as the seminal paper on randomized algorithms.) This algorithm uses *random sampling*. Let S_1 be a random subset of S having size $n^{2/3}$, and let δ be the minimum distance in S_1. Consider the grid with cells of size δ. Let N be the total number of pairs (p, q), where $p \in S$ and q is contained in p's cell or in one of the $3^D - 1$ neighboring cells. Note that N is a random variable. Rabin proves that N is bounded by $O(n)$ with probability at least $1 - 2^{-n^{\Omega(1)}}$. That is, with a very high probability, δ gives a good grid size.

How do we compute the value of δ? Rabin proposes choosing a random subset S_2 of S_1 of size $|S_1|^{2/3} = n^{4/9}$, and to compute the minimum distance δ' of S_2 by a brute force algorithm, taking $O((n^{4/9})^2) = O(n)$ time. Since with a very high probability, δ' gives a good grid size for the set S_1, we can use δ' to find δ in $O(n^{2/3})$ expected time.

Hence, the entire algorithm takes $O(n)$ expected time. (We remark that when Rabin wrote his paper, the implementation of the hashing procedure was left open. Only in 1984, this gap was filled by the perfect hashing scheme of Fredman, Komlós and Szemerédi [66].) Recently, Dietzfelbinger et al. [54] gave a complete description of an implementation of Rabin's algorithm. In particular, they provide all details of

needed.

Fortune and Hopcroft [64] gave an alternative algorithm to find a good grid size. They assume that a special operation $Findbucket(\delta, p)$ is available which computes in unit time the cell in the δ-grid that contains the point p. In this model, they give a recursive algorithm that computes a good grid size in $O(n \log \log n)$ worst-case time. Moreover, given this grid size, the closest pair can be computed in linear time. We remark that if unbounded space is available, this gives a closest pair algorithm with $O(n \log \log n)$ worst-case running time.

Here is a brief description of the algorithm that computes the good grid size. Start with a grid size δ equal to D/n times the side length of the smallest axes-parallel cube containing S. Insert the points into the δ-grid until one cell contains $\sqrt{n}$ points, or all points have been inserted. For each cell containing more than one point, call the algorithm recursively on these points. After all recursive calls have been completed, set δ to the smallest value returned by these recursive calls. Then start the algorithm again with this new value of δ. After all points of S have been inserted into the current grid, call the algorithm recursively on all cells containing more than one point.

2.4.2 A sieve algorithm

Khuller and Matias [84] give a randomized *sieve* algorithm to find a good grid size. Their algorithm iteratively discards points from S that are known to be "far away" from all other points. Clearly, such points do not play a role for determining the closest pair. Here is a description of the algorithm. Recall the notion of *neighborhood* in a grid, as defined at the end of Section 1. We make a sequence of iterations. Initially, we set $S_1 := S$ and $i := 1$. During the i-th iteration, we pick a random point $p_i \in S_i$, and compute its nearest neighbor q_i in S_i—by brute force. Let d_i be the distance between p_i and q_i. Then—using perfect hashing—we store the points of S_i in a $d_i/(4D)$-grid and determine the set A of all points of S_i that do not contain any other points of S_i in their neighborhoods. Then, we set $S_{i+1} := S_i \setminus A$. If $S_{i+1} \neq \emptyset$, then we proceed with the next iteration. Otherwise, the algorithm stops and outputs d_i.

Consider the i-th iteration. Let p be a point of S_i. The following two properties hold.

1. If $d(p, S_i) \leq d_i/(4D)$, then p has another point of S_i in its neighborhood and, therefore, p belongs to S_{i+1}.

2. If $d(p, S_i) > d_i/2$, then the neighborhood of p is empty and, hence, p does not belong to S_{i+1}.

First, we claim that $d_{i+1} \leq d_i/2$. This is proved as follows. Since $p_{i+1} \in S_{i+1}$, there is a point $q \in S_i$ that is in the neighborhood of p_{i+1}—in the $d_i/(4D)$-grid. Note that $d(p_{i+1}, q) \leq d_i/2$. Also, q is contained in the set S_{i+1}. As a result, we have $d_{i+1} = d(p_{i+1}, S_{i+1}) \leq d(p_{i+1}, q) \leq d_i/2$. In particular, the sequence $d_1, d_2, \ldots$ is decreasing.

Let ℓ be the number of iterations made by the algorithm, i.e., $S_\ell \neq \emptyset$, but $S_{\ell+1} = \emptyset$. We claim that $d_\ell/(4D) \leq d(S) \leq d_\ell$. The right inequality trivially holds, because d_ℓ

11

(resp. j) be the index such that $P \in S_i \setminus S_{i+1}$ (resp. $Q \in S_j \setminus S_{j+1}$). Assume w.l.o.g. that $i \leq j$. Then $Q \in S_i$ and $d(S) = d(P,Q) = d(P,S_i)$. Note that $d(P,S_i) > d_i/(4D)$. (Otherwise, by the first property, P would belong to S_{i+1}.) Therefore, $d(S) > d_i/(4D) \geq d_\ell/(4D)$.

Hence, the algorithm computes a grid size d_ℓ that approximates the minimum distance $d(S)$ up to a constant factor. Given this grid size we can find the closest pair using hashing techniques, in linear expected time.

We analyze the expected running time of the sieve algorithm. First note that using perfect hashing, the i-th iteration takes $O(|S_i|)$ expected time. For $1 \leq i \leq \ell$, let s_i be the expected size of S_i. Moreover, define $s_{\ell+1} := s_{\ell+2} := \ldots := s_n := 0$. (Note that $\ell \leq n$.) We will show that $s_i \leq s_{i+1}/2$. This implies that $s_i \leq n/2^{i-1}$. Hence, the total expected running time of all iterations is proportional to

$$E(\sum_{i=1}^{n} |S_i|) = \sum_{i=1}^{n} E(|S_i|) \leq \sum_{i=1}^{n} n/2^{i-1} \leq 2n.$$

That is, the entire algorithm takes linear expected time.

It remains to show that $s_{i+1} \leq s_i/2$. If $s_i = 0$, then also $s_{i+1} = 0$ and the claim holds. Assume that $s_i \neq 0$. Consider the conditional expectation $E(|S_{i+1}| \mid |S_i| = k)$. Let r be a point of S_i such that $d(r, S_i) \geq d_i$. Then the second property implies that $r \notin S_{i+1}$.

Take the points in S_i and label them $r_1, r_2, \ldots, r_k$ such that $d(r_1, S_i) \leq d(r_2, S_i) \leq \ldots \leq d(r_k, S_i)$. The point p_i is chosen randomly from the set S_i, so it can be any of the r_j's with equal probability. Thus $E(|S_{i+1}| \mid |S_i| = k) \leq k/2$, from which it follows that $s_{i+1} = \sum_k E(|S_{i+1}| \mid |S_i| = k) \cdot \Pr(|S_i| = k) \leq s_i/2$.

2.4.3 A randomized incremental algorithm

The final algorithm of this section is due to Golin et al. [71]. It follows the *randomized incremental construction paradigm*. (See Seidel [110].) Number the points of S randomly $p_1, p_2, \ldots, p_n$. For $2 \leq i \leq n$, let $S_i := \{p_1, \ldots, p_i\}$. The algorithm computes $d(S_2), d(S_3) \ldots, d(S_n)$, in this order. Assume that $\delta := d(S_i)$ has been computed already. We assume that the points of S_i are stored in a δ-grid. To compute $d(S_{i+1})$, we do the following. Let p be the point of $S_{i+1} \setminus S_i$. If the minimum distance of S_{i+1} is less than δ, then there must be points in the neighborhood of p's cell. By Lemma 2.1, there are at most a constant number of points in this neighborhood. We find all these points using perfect hashing, and compute their distances to p. If all these distances are at least equal to δ, then we know that $d(S_{i+1}) = \delta$. In this case, we just add the point p to the grid, and proceed with the next iteration. Otherwise, the smallest new distance found is equal to $d(S_{i+1})$. In this case, we store all points of S_{i+1} in a *new* $d(S_{i+1})$-grid.

In the worst case, we have to build a new grid during each iteration, leading to a quadratic running time. As we will show now, however, the expected running time of this algorithm is only linear.

For the sake of analysis, assume we run our algorithm *backwards*. Then, one iteration can be viewed by picking a random point p of S_{i+1}, deleting it, and possibly

where either the point p is not part of any closest pair of S_{i+1}, or the set S_{i+1} also contains a closest pair p is not part of. In this case, the current grid is not dismantled when deleting p. Hence, this iteration takes constant expected time. The second case is where the set S_{i+1} contains several closest pairs which all contain the point p. Then, the removal of p increases the minimum distance and, hence, the current grid is dismantled, taking $O(i+1)$ expected time. Since p is a random point, this happens only with probability $1/(i+1)$. Hence, this iteration also takes constant expected time. The remaining case is where the set S_{i+1} contains exactly one closest pair and p is part of it. Again, the removal of p increases the minimum distance and, hence, the current grid is dismantled, in $O(i+1)$ expected time. Since p is a random point, this happens only with probability $2/(i+1)$. Hence, the iteration takes constant expected time.

It follows that the entire algorithm takes linear expected time. It can be shown that with high probability, the running time is bounded by $O(n \log n / \log \log n)$.

In [71], it is also shown that this algorithm can be implemented in the randomized algebraic computation tree model, using degraded grids. Then the expected running time increases to $O(n \log n)$, even with high probability. This is optimal in this model. (See Schwarz [107].)

2.5 Extensions of the closest pair problem

2.5.1 The all-nearest-neighbors problem

Let S be a set of n points in $\mathbb{R}^D$. Until now, we only looked at the problem of computing the minimum distance in S. An obvious generalization is to compute for each point p of S its *nearest neighbor*, i.e., another point of S that is closest to p.

Let us consider the planar case first. Shamos and Hoey's algorithm of Section 2.2.1 can easily be extended to solve the all-nearest-neighbors problem optimally, i.e., in $O(n \log n)$ time using linear space. In [78], Hinrichs, Nievergelt and Schorn extend their plane sweep algorithm that was given in Section 2.2.3 so that it solves the problem within the same complexity bounds.

The all-nearest-neighbors problem turns out to be more complicated for higher dimensions. Bentley [20, 22] shows how his divide-and-conquer algorithm of Section 2.2.2 can be adapted such that it solves the D-dimensional problem in $O(n \log^{D-1} n)$ time. (In particular, this is optimal if $D = 2$.) An important property that is used in the merge step is the fact that any point can be the nearest neighbor of at most a constant number of other points. (See [20, 51] for a proof of this.)

The first $O(n \log n)$-time algorithm for the all-nearest-neighbors problem for an arbitrary dimension D was given by Clarkson [40]. His algorithm uses randomization— hence, the running time is expected—and the floor function. Vaidya [125] solves the problem deterministically, in $O(n \log n)$ time. His algorithm can be implemented in the algebraic computation tree model and is, therefore, optimal. The algorithms in [40] and [125] are based on carefully constructed subdivisions of $\mathbb{R}^D$ into axes-parallel rectangles. Callahan and Kosaraju [31] defined the so-called *well-separated pair decomposition*, which captures the important notions on which these two algorithms

proximity problems such as the all-nearest-neighbors problem. In view of this, we only give the algorithm that is based on the well-separated pair decomposition.

2.5.2 The well-separated pair decomposition

The well-separated pair decomposition appeared for the first time in Callahan and Kosaraju [31]. (See [34] for the full version of this paper. Callahan's Ph.D. Thesis [30] contains a detailed discussion. We also mention that Salowe [104] already used this notion implicitly.)

Intuitively, a well-separated pair decomposition of a point set S is a partition of the set of all pairs p, q of distinct points into a collection of pairs of sets (A, B) such that all distances between points in A and points in B are large compared to the distances within A and B.

To define this notion more precisely, let s denote a fixed positive constant, called the *separation constant*. Let X and Y be two sets of points, and let $R(X)$ and $R(Y)$ be the smallest axes-parallel rectangles containing X and Y, respectively. The sets X and Y are said to be *well-separated* if there are two D-dimensional balls B_X and B_Y having the same radius—say r—and containing the rectangles $R(X)$ and $R(Y)$, respectively, such that the minimal distance between these balls is at least equal to sr.

Definition 2.3 *Let S be a set of n points in $\mathbb{R}^D$, and let $s > 0$ be any constant. A well-separated pair decomposition (WSPD) of S is a set of pairs of nonempty subsets of S,*

$$\{\{A_1, B_1\}, \{A_2, B_2\}, \ldots, \{A_m, B_m\}\},$$

such that

1. *$A_i \cap B_i = \emptyset$, for all $i = 1, 2, \ldots, m$.*

2. *For each unordered pair of distinct elements $\{a, b\}$ of S, there exists a unique pair $\{A_i, B_i\}$ in the decomposition such that $a \in A_i$ and $b \in B_i$.*

3. *A_i and B_i are well-separated, for all $i = 1, 2, \ldots, m$.*

The integer m is called the *size* of the WSPD. We now describe the algorithm of Callahan and Kosaraju for constructing such a decomposition for which $m = O(n)$. They first construct a binary tree, called the *fair split tree*, which is recursively defined as follows. If S consists of only one point, then the tree consists of one node. Assume that S contains more than one point. Consider the smallest axes-parallel rectangle $R(S)$ that contains S, and let i, $1 \leq i \leq D$, be the dimension along which this rectangle has the longest side. Split $R(S)$ into two rectangles by cutting the i-th interval into two equal parts. Let S_1 and S_2 be the subsets of S that are contained in these two new rectangles. (Note that both S_1 and S_2 are non-empty.) Then the fair split tree for S consists of a root, together with two subtrees which are fair split trees for S_1 and S_2, respectively.

The leaves of the fair split tree are in one-to-one correspondence with the points of S. If v is a node of this tree, and S_v is the subset of S that corresponds to the leaves in the subtree of v, then we say that v *represents S_v*.

14

a naive implementation of the recursive definition leads to a quadratic running time. Callahan and Kosaraju show how to reduce the construction time to $O(n \log n)$.

Let a *partial fair split tree* be defined in the same way as the fair split tree, except that sets represented by the leaves may have size larger than one. The main idea of the efficient construction is to construct a partial fair split tree in which each leaf represents at most $n/2$ points. Then, for each leaf, a fair split tree is constructed for the points represented by this leaf, using the same algorithm recursively. To be more precise, consider the set S. We make a sequence of splits. Each split results in two non-empty subsets. The next split in the sequence is performed on the larger of these two sets. Hence, if for $j \geq 0$, S_j is the larger of the two sets that result after the j-th split, then the $(j+1)$-st split consists of splitting the smallest axes-parallel rectangle containing S_j into two equal parts along its longest side. We stop this process if the set S_j has size at most $n/2$.

If for each $1 \leq i \leq D$, we have a list storing the points of S sorted by their i-th coordinates, together with cross-references between these lists, then this sequence of splits can be performed in $O(n)$ time. Here, a key observation is that if we split along the i-th dimension, then by walking from both ends of the i-th list, this split takes time proportional to the size of the smaller of the two resulting subsets.

This algorithm correctly computes a fair split tree, in $O(n \log n)$ time. For a detailed proof, we refer the reader to [30, 31, 34].

We now sketch how to use the fair split tree in order to obtain a WSPD of S. First, we need some notation. If A is a set of points, then $l(A)$ denotes the length of the longest side of the smallest axes-parallel rectangle that contains A. Let A and B be two sets of points that are represented by two different nodes of the fair split tree. Then we will write $A \prec B$ if either $l(A) < l(B)$, or $l(A) = l(B)$ and the node representing A precedes the node representing B in postorder. Extend this ordering to $\preceq$ in the obvious way. If v is an internal node of the fair split tree, then we denote by A_v and B_v the sets of points that are represented by the two children of v.

The WSPD is obtained by calling the procedure $findpairs(A_v, B_v)$ for each internal node v of the fair split tree. This procedure does the following. If A_v and B_v are well-separated, then $findpairs(A_v, B_v)$ returns the pair $\{A_v, B_v\}$ and terminates. Otherwise, assume w.l.o.g. that $B_v \prec A_v$. (Otherwise, swap A_v and B_v.) Note that A_v contains more than one point, because otherwise we would have $l(B_v) = l(A_v) = 0$ and, therefore, A_v and B_v would be well-separated. Therefore, the node representing A_v has two children. Call these v_1 and v_2. Recursively call the procedures $findpairs(A_{v_1}, B_v)$ and $findpairs(A_{v_2}, B_v)$, and return the union of their outputs as the output of $findpairs(A_v, B_v)$.

The pairs that are reported by all these procedure calls gives a WSPD for the entire set S. Note that each pair can be represented by just two pointers to the appropriate nodes in the fair split tree.

To analyze the size of the computed WSPD and the running time of the algorithm itself, we use the following claim which follows from a packing argument: If A is any subset of S that is represented by some non-root node of the fair split tree, then we denote by $p(A)$ the subset that is represented by the parent of this node. The claim is

15

WSPD contains at most a constant number of pairs of the form $\{A, B\}$, such that $B \prec p(A) \preceq p(B)$. This implies that the WSPD that is computed has size $O(n)$. Also, the time needed to compute it—given the fair split tree—is bounded by $O(n)$.

Hence, given a set S of n points in $\mathbb{R}^D$, we can in $O(n \log n)$ time compute a well-separated pair decomposition for S. We now show that this is optimal. First, we need a lemma, whose proof is trivial.

Lemma 2.3 *Assume the separation constant s is larger than two. Let $\{A, B\}$ be a pair in any WSPD of S. Assume there is a pair of points (a, b) such that $a \in A$, $b \in B$, and b is a nearest neighbor of a. Then $A = \{a\}$.*

Let P, Q be a closest pair in S. Then this lemma implies that $\{\{P\}, \{Q\}\}$ is a pair in any WSPD of S. Hence, given any WSPD, we can find the closest pair in time proportional to the size of the WSPD. This proves that computing any WSPD takes $\Omega(n \log n)$ time in the algebraic computation tree model.

How do we use the WSPD for solving the all-nearest-neighbors problem? By Lemma 2.3, we only have to consider pairs of the WSPD, one of whose sets is a singleton. An important observation is the following lemma, which is a generalization of the fact that a point can be the nearest neighbor at most a constant number of other points.

Lemma 2.4 *Let A and B be sets of points in $\mathbb{R}^D$, such that for all $a \in A$, $\{a\}$ and B are well-separated. Also, assume that for all pairs a, a' of distinct points in A, the distance from a to the smallest sphere containing B is at most equal to the distance between a and a'. Then the size of A is bounded by a constant that only depends on the dimension D and the separation constant s.*

For any node v of the fair split tree, let $f(v)$ be the set of all points $a \in S$ such that the pair $\{\{a\}, B_{v'}\}$ is contained in the WSPD for some ancestor v' of v. Also, define $N(v)$ as the set of all points $a \in f(v)$ such that the distance from a to the smallest sphere containing B_v is at most equal to the smallest distance between a and any point of $f(v)$. Then Lemma 2.4 implies that this set $N(v)$ has size $O(1)$.

We compute the sets $N(v)$ top down. If v is the root of the fair split tree, then $N(v)$ is empty. Let u be any node with parent v. Then initially we set $N(u)$ to the union of $N(v)$ and the set of all points a such that $\{\{a\}, B_u\}$ is a pair of the WSPD. Then we remove elements that do not satisfy the definition of $N(u)$. Computing $N(u)$ from $N(v)$ takes constant time. Hence, overall we need linear time to compute all these sets.

Let a be a point of S, let b be its nearest neighbor, and let v be the leaf of the fair split tree that represents b. Then it is easy to see that a must be contained in the set $N(v)$. Hence, by considering all leaves, we find all nearest-neighbors, in linear time. This proves that we can solve the all-nearest-neighbors problem in $O(n \log n)$ time.

16

In this version of the problem, we are given a set S of n points in $\mathbb{R}^D$ and an integer k, $1 \leq k \leq \binom{n}{2}$, and we have to compute the k smallest distances in the set S.

The first algorithms for this problem are due to Smid [118]. He gives an incremental algorithm using a space efficient variant of range trees [119] that solves the problem in $O(n^{4/3} \log n + n\sqrt{k} \log k)$ time. For the planar case, this can be improved to $O(n \log n + n\sqrt{k} \log k)$ by using a straightforward generalization of the sweep algorithm given in Section 2.2.3.

For the case where $1 \leq k \leq n$, this result can further be improved by first computing for each point its nearest neighbor and then selecting among these n pairs those k whose distance is smallest. This gives a set of at most $2k$ points that contains the k closest pairs of the set S. We apply the above algorithms on this small set. The result is an algorithm for solving the k closest pairs problem in $O(n \log n + k\sqrt{k} \log k)$ time.

Dickerson, Drysdale and Sack [52] give a simple algorithm for the planar case that uses the Delaunay triangulation. Here is a description of their algorithm. Given a set S of n points in the plane, compute the Delaunay triangulation DT of S. For each point p of S, sort the edges of DT that are incident to p in increasing order of their length. Also, sort all edges of DT by their length. Next, insert the k shortest edges of DT into a heap. (If k is larger than the number of edges in DT, then insert all edges into the heap.) Now we can start with enumerating the k closest pairs. For $i = 1, 2, \ldots, k$, do the following: Delete the pair, say (p, q), that is stored in the heap and that has minimum distance, and report this pair as being the i-th closest pair. For all Delaunay edges of the form (q, x) such that (i) $d(q, x) \leq d(p, q)$ and (ii) the pair (p, x) has not been reported, insert (p, x) into the heap. Similarly, for all Delaunay edges of the form (x, v) such that (i) $d(x, v) \leq d(p, q)$ and (ii) the pair (x, q) has not been reported, insert (x, q) into the heap.

The correctness of this algorithm is based on the following property of the Delaunay triangulation. Let p and q be any two points of S. Then either (p, q) is an edge of DT, or there are points $x_1, x_2, \ldots, x_m$ such that (i) (p, x_1), (x_m, q) and (x_i, x_{i+1}), $1 \leq i < m$, are edges of DT, (ii) $d(x_i, x_{i+1}) < d(p, q)$ for $1 \leq i < m$, (iii) $d(p, x_i) < d(p, q)$ and $d(x_i, q) < d(p, q)$ for $1 \leq i \leq m$, and (iv) $d(p, x_1) \leq d(x_1, q)$ or $d(q, x_m) \leq d(x_m, p)$. Using this property, Dickerson et al. show that if (p, q) is the i-th closest pair, then the pair (p, q) will be contained in the heap by the time all distances smaller than $d(p, q)$ have been reported. They also show that the running time of the algorithm is bounded by $O(n \log n + k \log k)$.

The efficiency of the algorithm of Dickerson et al. heavily depends on the fact that the Delaunay triangulation has linear size. This does not hold for dimensions greater than two. Dickerson and Eppstein [53], however, circumvent this by using the following result of Bern, Eppstein and Gilbert [26]: Given a set S of n points in $\mathbb{R}^D$, there is a superset S' of S having size $O(n)$, such that the Delaunay triangulation DT' of S' has size $O(n)$ and bounded degree. Such a superset can be computed in $O(n \log n)$ time. Applying the algorithm given above to DT' gives the k closest pairs of S. (Clearly, the correctness proof is more complicated than in the planar case, because we use the Delaunay triangulation of S' instead of S. See [53].) The entire algorithm solves the k closest pairs problem in $O(n \log n + k \log k)$ time. Note that this algorithm reports

17

pairs—in no particular order—in $O(n \log n + k)$ time. This is optimal in the algebraic computation tree model. We remark that Arya and Smid [15] have shown that the algorithm of [52] also works if we replace the Delaunay triangulation by any bounded degree *spanner*. (See Section 6.1.)

The first optimal algorithm for the k closest pairs problem is due to Salowe [105]. He first combines a variant of Vaidya's algorithm [125] with the parametric search technique to compute the k-th smallest L_∞-distance. Then, again using a variant of Vaidya's algorithm, he enumerates all pairs of points that have distance at most equal to $D\delta$. The number of these pairs is bounded by $O(n + k)$, and the k closest pairs in S are among them. Hence, using a selection algorithm we get the k closest pairs. The entire algorithm takes $O(n \log n + k)$ time.

Although Salowe's algorithm is optimal, it is rather complicated because it uses the parametric search technique. Lenhof and Smid [87] give another algorithm that does not use any complicated data structures. Their algorithm is basically the same as the one presented in Section 2.3. Consider a degraded δ-grid for the set S. (See Section 2.3.1.) Number the cells of this grid arbitrarily. Let n_i denote the number of points of S that are contained in the i-th cell, and let $\Sigma(\delta) := \sum_i \binom{n_i}{2}$.

The algorithm of Lenhof and Smid first computes a grid size δ such that $k \leq \Sigma(\delta) \leq 3^D(k + n/2)$. This grid size is computed in $O(n \log n)$ time by a simple variant of the search procedure sketched in Section 2.3. Then, a degraded $(D\delta)$-grid is constructed and all pairs of points that are contained in the same cell or in neighboring cells are enumerated. The number of enumerated pairs is bounded by $O(n + k)$ and they include the k closest pairs. The total running time of this algorithm is $O(n \log n + k)$. We remark that indirect addressing is used and, therefore, the algorithm falls outside the algebraic computation tree model. (The same approach solves the fixed-radius near neighbors problem: Given $\delta > 0$, all pairs of points that are at distance at most δ are enumerated in time proportional to $n \log n$ plus the number of reported pairs. See [86].)

The algorithms of [52] and [87] have been implemented by von Zülow [129]. It turns out that the algorithm of [87] is faster than that of [52]. Also, the algorithm of [87] performs very well for higher dimensions.

We finally mention that the well-separated pair decomposition (WSPD) can also be used to solve the k closest pairs problem optimally. Recall that if A is a set of points, then $R(A)$ denotes the smallest axes-parallel rectangle that contains A. Let $\{\{A_1, B_1\}, \{A_2, B_2\}, \dots, \{A_m, B_m\}\}$ be a WSPD for S, listed in increasing order of $d(R(A_i), R(B_i))$. To report the k closest pairs, we first find the smallest index i such that $\sum_{j=1}^{i} |A_j| \cdot |B_j| \geq k$. Then, we compute all pairs $\{A, B\}$ of the WSPD for which $d(R(A), R(B)) \leq (1 + 4/s) \, d(R(A_i), R(B_i))$, where s is the separation constant. This gives us a sequence of $O(n+k)$ pairs of points that contains the k closest pairs. Hence, again using a selection algorithm, we find the k closest pairs. For details, see [30].

2.5.4 k-point clustering problems

In the previous sections, we used closeness measures that are based on distances among *pairs* of points. In this section, we consider k-point clustering problems, in which the

set S of n points in $\mathbb{R}^D$ and an integer k, $2 \leq k \leq n$, we want to find a subset S' of S of size k that minimizes some closeness measure. For example, we may want to minimize the diameter of the k points, its smallest enclosing circle, it smallest enclosing axes-parallel cube, etc. Clearly, different closeness measures lead to different optimal k-point subsets.

We mention some of the results in this area. Dobkin, Drysdale and Guibas [56] use the following technique to find a k-point subset for which the perimeter of their convex hull is minimized. First, they give a polynomial time algorithm for solving this problem. Then, they give an improved algorithm that first constructs the (ck)-order Voronoi diagram of the points of S, for some suitable constant c. They show that the optimal k-point subset is contained in the set of ck points corresponding to a region in this diagram. Hence, for each of the $O(kn)$ regions, they apply their first algorithm to the corresponding subset of size ck. Aggarwal et al. [5] improved this technique, by showing that it suffices to consider only $O(n)$ regions of the (ck)-order Voronoi diagram. In [5], also closeness measures such as the diameter, enclosing square, or perimeter of the enclosing rectangle are considered.

In Smid [122], a simple plane sweep algorithm is given for finding the k points in a set of n planar points whose enclosing axes-parallel square is minimal. Efrat, Sharir and Ziv [61] apply the parametric search technique for finding k points whose enclosing circle is minimal. Matoušek [90] gives a simpler algorithm for the same problem in which he replaces parametric search by a randomized search technique.

Most algorithms for k-point clustering problems are inefficient for large values of k. In [91], Matoušek gives algorithms that are especially efficient if k is close to n. These are based on generalizing *LP-type problems* [114] to optimization problems with k violated constraints.

Eppstein and Erickson [62] improve the general framework of [56, 5]. Their main idea is to replace the expensive $O(k)$-order Voronoi diagram by sets of $O(k)$ nearest neighbors to each of the points of S. In this way, the number of $O(k)$-size subsets to which an "expensive" algorithm is applied is reduced from $O(n)$ in [5] to only $O(n/k)$.

The framework was further improved by Datta et al. [50]. Consider a closeness measure μ, and let S' be the k-point subset that minimizes this measure. The technique of [50] works provided the measure μ satisfies the condition that the value of $\mu(S')$ is proportional to the size of the smallest axes-parallel D-dimensional cube that contains k points of S. Examples for $\mu(A)$ are the diameter of A, the radius of the enclosing ball of A, the size of the enclosing axes-parallel cube and the perimeter of the enclosing rectangle. (The area of the enclosing rectangle does *not* satisfy this condition.)

The algorithm of [50] first constructs a degraded δ-grid such that (i) there is a cell that contains at least k points of S, and (ii) each cell contains at most $2^D k$ points of S. This grid is computed in $O(n \log n)$ time, by applying the same search technique as in Section 2.3. Then, for each cell C of this grid, the set S_C of points that are contained inside this cell or within a constant number of neighboring cells are collected. If $|S_C| \geq k$, then we compute the optimal k-point subset of S_C using some other "expensive" algorithm. The k-point subset found in this way having minimum μ-value is the optimal subset of the entire set S. It is clear that in this way, the expensive

applications to several closeness measures is given in [50].

3   The on-line closest pair problem

In this section, we give algorithms for the *on-line closest pair problem*. In this problem, we have to maintain the closest pair of the set S under insertions of points. That is, we want to design a data structure that efficiently updates the closest pair if points are inserted into S. Given such a data structure, we can compute the closest pair of a point set whose elements become available one after another. When we get the next point, we just insert it into our data structure, and update the closest pair.

We saw that in the algebraic computation tree model, there is an $\Omega(n \log n)$ lower bound for the static closest pair problem. This immediately implies an $\Omega(\log n)$ lower bound per insertion for the on-line version of the problem. The main result of this section is a data structure of linear size that matches this lower bound.

Let S be the current point set, let δ be the minimum distance in S, and let p be the point to be inserted. We assume w.l.o.g. that p is not contained in S. Clearly, if the minimum distance in the new set $S \cup \{p\}$ is less than δ, p must be part of a closest pair. Therefore, in order to update the closest pair, we have to perform the following steps.

1. Find out if there is a point q in S such that $d(p, q) < \delta$. If there is no such point, then the closest pair does not change during the insertion of p. Otherwise, find a point q of S that is closest to p. In this case, (p, q) is the new closest pair.

2. Insert the new point p into the data structure.

Let B be the L_τ-ball with radius δ centered at p. Lemma 2.1 implies that B contains at most a constant number of points of S. Our strategy for implementing the first step will be to find a subset S' of S that contains all points of $S \cap B$ and whose size is "small". (Ideally, this size is bounded by a constant, although a size logarithmic in $|S|$ suffices.) Given this subset, we can trivially update the closest pair by computing the distances between p and all points of S'. Finding the subset S' efficiently is achieved by maintaining a subdivision of $\mathbb{R}^D$ into cells such that each cell contains only "few" points, and the ball B overlaps only "few" cells. The set S' is then obtained by performing point location queries in this subdivision.

After having performed the first step, we add point p to the cell of the subdivision that contains p and, if necessary, update the subdivision.

We now turn to concrete implementations of this insertion procedure.

3.1   Algorithms based on the logarithmic method

The first algorithms are based on applying Bentley's logarithmic method [21]. This application first appeared in Smid [120]. (See also Schwarz [107].)

Let S be the current set of points in $\mathbb{R}^D$, and let n denote its size. Write n in the binary number system, $n = \sum_{i \geq 0} a_i \, 2^i$, where $a_i \in \{0, 1\}$. Partition S (arbitrarily) into

each such i, there is a real number δ_i such that $d(S) \le \delta_i \le d(S_i)$. The data structure consists of the following.

1. The closest pair of S and its distance δ.

2. For each i such that $a_i = 1$, a δ_i-grid storing the points of S_i. We assume that the non-empty cells of this grid are stored in a balanced binary search tree, sorted lexicographically according to their "lower left" corners.

Now consider the insertion of a new point p. For each i such that $a_i = 1$, we find the cell of the δ_i-grid that contains p together with the $3^D - 1$ neighboring cells. Then we compute the distances between p and all points of S_i that are contained in these cells. If we find a distance less than δ, then we update δ and the closest pair.

It remains to update the rest of the data structure. Let j be the index such that $a_0 = a_1 = \ldots = a_{j-1} = 1$ and $a_j = 0$. Let $S_j := \{p\} \cup S_0 \cup S_1 \cup \ldots \cup S_{j-1}$ and $\delta_j := \delta$. We build a δ_j-grid for the set S_j, and discard the grids for the sets $S_0, S_1, \ldots, S_{j-1}$ (thereby implicitly making these sets empty).

To prove the correctness of this algorithm, note that since $d(S) \le \delta_i$, it suffices to compare p with all points of S_i that are in p's neighborhood—in the δ_i-grid. Hence, the closest pair is updated correctly. Also, the new value δ_j satisfies $d(S \cup \{p\}) \le \delta_j \le d(S_j)$, and for all $i > j$, we have $d(S \cup \{p\}) \le \delta_i \le d(S_i)$. Finally, the updated data structure contains a grid storing 2^i points for each i such that the binary representation of $n + 1$ contains a one at position i.

We analyze the complexity of the insertion algorithm. First note that since $\delta_i \le d(S_i)$, the neighborhood of p in the δ_i-grid contains at most a constant number of points of S_i. Hence, we spend $O(\log n)$ time for each grid. Since the data structure contains a logarithmic number of grids, the first part of the insertion algorithm takes $O(\log^2 n)$ time.

Consider the second part of the algorithm, in which we build the δ_j-grid. This step takes $O(|S_j| \log |S_j|)$ time, because of sorting. If we store the points in an appropriate sorted order, however, then this grid can be built in $O(|S_j|) = O(2^j)$ time. (See [107, 108] for details.) Since j can take any value between zero and $\lfloor \log n \rfloor$, the insertion time fluctuates widely. We claim, however, that the *amortized* time for the second step is bounded by $O(\log n)$.

To prove this claim, assume we start with an empty set S and perform a sequence of n insertions. Let k_j be the number of times that during these insertions, we build a grid for a subset of size 2^j. Then k_j is at most equal to the number of integers consisting of at most $1 + \lfloor \log n \rfloor$ bits whose j least significant bits are equal to one, and whose $(j + 1)$-st bit is equal to zero. That is, we have

$$ k_j \le 2^{\lfloor \log n \rfloor - j} \le n/2^j. $$

The total time spent for the second step during the n insertions is bounded by

$$ O\left(\sum_{j=0}^{\lfloor \log n \rfloor} k_j \cdot 2^j \right) = O\left(\sum_{j=0}^{\lfloor \log n \rfloor} \frac{n}{2^j} \cdot 2^j \right) = O(n \log n), $$

We have shown that the running time of the entire algorithm for maintaining the closest pair is bounded by $O(\log^2 n)$ worst-case time plus $O(\log n)$ amortized time per insertion. Using standard techniques (see Overmars [97, pages 102–105]), we can transform the data structure such that the second step takes logarithmic time in the worst case. Hence, we have a data structure that maintains the closest pair in $O(\log^2 n)$ worst-case time per insertion. The structure has size $O(n)$.

Note that the first step of the algorithm takes $O(\log^2 n)$ time, whereas the second step takes only $O(\log n)$ time. This suggests that an improvement is possible. Indeed, instead of writing n in the binary number system, we use the number system with base $\log n$. (See Overmars [97, pages 108–115] or Schwarz [107].) In this way, both steps take $O(\log^2 n / \log \log n)$ time, whereas the space used remains linear.

Hence, we have a data structure of linear size that maintains the closest pair in $O(\log^2 n / \log \log n)$ worst-case time per insertion. Note that the algorithm uses the floor function in order to find the grid cell containing the new point p. By replacing the grid by a degraded grid, the algorithm can be implemented in the algebraic computation tree model.

It is still possible to improve the above data structure. Consider again the data structure based on the representation of n in the binary number system. The main observation is the following: If we insert a point p, then for each i such that $a_i = 1$, we find the cell of the δ_i-grid that contains p (plus the neighboring cells). That is, we perform point location queries in a logarithmic number of grids, but always with the *same* query point p. In Schwarz and Smid [108], it is shown that the *fractional cascading technique* [38] can be extended so that all these queries together can be solved in $O(\log n \log \log n)$ time. The main problem is that we have grids with *different* grid sizes. Therefore, an ordering on the grid cells has to be introduced that is "compatible" with all these sizes. In [108] such an ordering is defined. As a result, locating the point p in all grids takes $O(\log n)$ comparisons. Since the ordering is quite complicated, however, each comparison takes $O(\log \log n)$ time. Overall, we get a data structure of size $O(n)$ that maintains the closest pair in $O(\log n \log \log n)$ amortized time per insertion. Note that we need the floor function for this result. This can probably be made worst-case, but the details will be tedious. It is not clear if the ordering on the grid cells can also be defined if we use degraded grids instead of standard grids.

We finally mention an extension of the above data structure. The structure as described above heavily uses the fact that we only insert points. It turns out, however, that the structure can be adapted for dealing with *semi-online updates*, as defined by Dobkin and Suri [58]. A sequence of updates is called semi-online if the insertions are on-line—i.e., they arrive in an unknown order—but when a point is inserted, we are told after how many updates from the moment of insertion it will be deleted. This extra information about the deletions can be used to guarantee that when a point is deleted, it is always contained in a grid storing a small subset. Because in a deletion the minimum distance may increase, we store extra information in the data structure for updating the closest pair efficiently. In this way, we get a data structure of size $O(n)$ that maintains the closest pair in $O(\log^2 n)$ worst-case time per insertion. For details, we refer the reader to Dobkin and Suri [58] and Smid [116, 117, 120].

In this section, we give a data structure that maintains the closest pair in $O(\log^{D-1} n)$ amortized time per insertion. The algorithms can be implemented in the algebraic computation tree model and, therefore, the result is optimal for the planar case. The results in this section appeared in Smid [123].

In the previous section, we maintained a sequence of subdivisions of $\mathrm{I\!R}^D$, where each subdivision was a grid. Now we store the point set in only one subdivision that is maintained in such a way that we do not need the floor function.

We will use the *skewer tree*, see Edelsbrunner, Haring and Hilbert [60]. This data structure stores a collection of m non-overlapping axes-parallel D-dimensional rectangles, such that point location queries can be performed in $O(\log^{D-1} m)$ time. The skewer tree uses $O(m)$ space. In [123], it is shown that this data structure can be made dynamic such that the following operations can be performed in $O(\log^2 m)$ amortized time: Insert a rectangle into the collection, such that the rectangles in the new collection are still non-overlapping; split a rectangle into two axes-parallel rectangles. (Deletions and merges can also be supported, but then the complexity increases by a factor of $\log \log m$ because of the usage of *dynamic fractional cascading* [94].)

Let S be the current set of points, and let n denote its size. The data structure consists of the following.

1. The closest pair in S and its distance δ.

2. A subdivision of $\mathrm{I\!R}^D$ into non-overlapping D-dimensional axes-parallel rectangles. Each rectangle in this subdivision has sides of length at least δ, and contains at least one and at most $(2D)^D \log^{D-1} n$ points of S.

3. The rectangles of the subdivision are stored in a skewer tree. With each rectangle R, we store a list of those points in S that are contained in R.

Assume we insert a new point $p = (p_1, \ldots, p_D)$ into S. We update the data structure as follows. First, we perform 3^D point location queries in the skewer tree, with query points $(p_1 + \epsilon_1, \ldots, p_D + \epsilon_D)$, for $\epsilon_1, \ldots, \epsilon_D \in \{-\delta, 0, \delta\}$. Then we compute the distances between p and all points of S that are contained in the rectangles that are found. If we find a distance less than δ, then we update δ and the closest pair accordingly. (See Figure 2.)

Next, we insert p into the list of the rectangle R it belongs to. If this updated list contains at least $(2D)^D \log^{D-1} n$ points, we perform a sequence of at most D split operations on R such that (i) the second property of the data structure is satisfied, and additionally (ii) each of the new rectangles contains at most half of the points that were contained in R. For the details of these split operations, we refer the reader to [123]. (In the next section, we will describe the split operation for a slightly modified subdivision. There, one split operation suffices.)

By the above mentioned property of dynamic skewer trees, it follows that the entire insertion procedure takes $O(\log^{D-1} n + \log^2 n)$ amortized time. Consider a new rectangle of the subdivision. At this moment, it contains at most $(1/2)(2D)^D \log^{D-1} n$ points. Therefore, if this rectangle is split, there must have been this many insertions
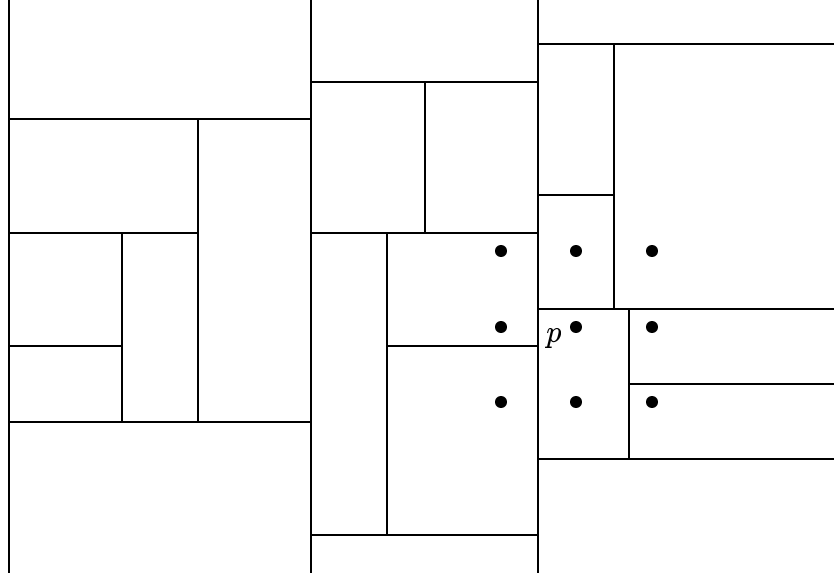
23

Figure 2: The 9 point location queries in the planar case.

into it. During those insertions, no split operation is necessary. Hence, the amortized insertion time of the entire data structure is bounded by $O(\log^{D-1} n)$. It is clear that it uses linear space.

3.3 An algorithm that is optimal for all dimensions

In this section, we give an optimal data structure for maintaining the closest pair under insertions. This result is due to Schwarz, Smid and Snoeyink [109]. We will use basically the same subdivision as in the previous section. The improvement stems from the use of a better point location data structure.

During the sequence of insertions, we maintain a hierarchical collection of axes-parallel D-dimensional rectangles. The data structure stores the following information.

1. The closest pair in S and its distance δ.

2. A binary tree T representing the hierarchical collection. Each node of T stores a rectangle. The rectangles stored in the leaves form a partition of $\mathbb{R}^D$ and they represent the "current" subdivision. With each leaf, we also store a list of all points of S contained in the corresponding rectangle. For each internal node of T, the rectangle stored in it is equal to the union of the two rectangles stored in its children.

3. The current subdivision consists of non-overlapping rectangles such that (i) each rectangle has sides of length at least δ, and (ii) each rectangle contains at least one and at most $(2D + 2)^D$ points of S.

Note that in order to solve a point location query in the current subdivision, we can walk down the tree T. The query time is proportional to the height of T, which, as we

24

algorithm.

To insert a new point p into S, we perform the same 3^D point location queries as in Section 3.2, and update the closest pair. Afterwards, we insert p into the rectangle R containing it. If this rectangle now contains $1 + (2D + 2)^D$ points, we split it as follows.

Let R have the form $[a_1, b_1] \times \ldots \times [a_D, b_D]$, and let S' be the points of the new set S that are contained in R. For $1 \le i \le D$, compute the value m_i (resp. M_i) which is the minimum (resp. maximum) i-th coordinate of any point of S'. Let i be an index such that $M_i - m_i > 2\delta$. (It follows from Lemma 2.1 that such an index exists.) Let $c_i := m_i + (M_i - m_i)/2$. Then we split R into two rectangles

$$R_l = [a_1, b_1] \times \ldots \times [a_{i-1}, b_{i-1}] \times [a_i, c_i] \times [a_{i+1}, b_{i+1}] \times \ldots \times [a_D, b_D]$$

and

$$R_r = [a_1, b_1] \times \ldots \times [a_{i-1}, b_{i-1}] \times [c_i, b_i] \times [a_{i+1}, b_{i+1}] \times \ldots \times [a_D, b_D].$$

In the tree T, we give the leaf storing R two new children corresponding to R_l and R_r, and give these two new leaves the appropriate lists of points.

It is not hard to see that this algorithm correctly maintains the closest pair. To estimate the running time, we observe that we perform a constant number of point location queries and at most one split operation. Since the tree T may have a linear height, the worst-case running time of the insertion algorithm is $O(n)$.

There are two possibilities to improve the running time. First, we can use a *centroid decomposition* to represent the tree T as a balanced tree. A *β-centroid* is a node of T whose removal results in three components each containing at most a fraction β of all nodes. The centroid decomposition is obtained by repeatedly choosing such centroids in each of the components. Guibas et al. [74] show how this decomposition can be computed in linear time. Given this centroid decomposition, point location queries can be solved in logarithmic time. To maintain it efficiently, we apply the partial rebuilding technique. (See Lueker [89] and Overmars [97, Chapter IV].) In this way, a split operation takes $O(\log n)$ amortized time. Hence, overall we get an algorithm for maintaining the closest pair in $O(\log n)$ amortized time per insertion, and using $O(n)$ space. This version of the algorithm has been implemented by Hintz [79].

A second improvement is obtained by storing the hierarchical subdivision using *dynamic trees* [115]. Cohen and Tamassia [44] give a general technique to maintain hierarchical subdivisions such that point location queries and several other operations—such as the split operation—take $O(\log n)$ time in the worst case. (For a detailed discussion of the application of this technique to our problem, see Schwarz [107].) Hence using this technique, we get our main result: A data structure that maintains the closest pair in $O(\log n)$ worst-case time per insertion. This data structure uses linear space, and the algorithms fit into the algebraic computation tree model. Therefore, this solves the on-line closest pair problem optimally.

We finally mention that the techniques presented in this section can be generalized to get efficient on-line solutions to k-point clustering problems, for a large class of closeness measures. (See Section 2.5.4, Datta et al. [50] and Schwarz [107].)

We now turn to the fully *dynamic closest pair problem*. Here, we have to maintain the closest pair in a point set if both insertions and deletions are allowed. Note that intuitively, insertions are easier to handle than deletions: If a point is inserted, we only have to check the neighborhood of this new point. On the other hand, if we delete a point that is part of the closest pair, then we have to find the new closest pair.

4.1 The deletions-only case

Supowit [124] showed that the divide-and-conquer algorithm of Section 2.2.2 can be turned into a data structure that maintains the closest pair if only deletions have to be supported.

We sketch this structure for the planar case. Let S be a set of n points in the plane. We store the points of S in the leaves of a balanced binary search tree, sorted by their x-coordinates. For any node w of this tree, we denote by S_w the subset of S that is stored in the subtree of w. Every internal node w contains additional information. Let u and v be the left and right child of w, respectively.

We store with w the minimum distance $d(S_w)$ of the set S_w, the value $\delta_w :=$ $\min(d(S_u), d(S_v))$, and a value m_w that is between the maximum x-coordinate in S_u and the minimum x-coordinate in S_v. Also, we store with w a tree T_w storing all points of S_w that are within distance δ_w of the vertical line $x = m_w$. The points are stored in the leaves of this tree, sorted by their y-coordinates. Finally, we store with w a heap H_w, storing all distances $d(a, b)$, where a and b are points of T_w that are within five positions of each other in this tree. Note that $d(S_w)$ is the minimum of δ_w and the smallest element in H_w.

We also maintain a heap H that contains the smallest elements of all heaps H_w. It is clear that the smallest element stored in H is the minimum distance in the overall set S.

To delete a point p, we search in the tree with its x-coordinate. For each node w encountered, we update all relevant information. The important observation is that the values $d(S_w)$ and δ_w can only increase. Therefore, a point of S_w will be inserted into the tree T_w at most once. If such a point is inserted into T_w, it causes a constant number of updates in the heaps H_w and H.

Supowit shows that the data structure maintains the closest pair in $O(\log^2 n)$ amortized time per deletion. Clearly, the structure uses $O(n \log n)$ space.

This result can be generalized to any dimension D, in much the same way as the algorithm of Section 2.2.2 generalizes. The result is a data structure of size $O(n \log^{D-1} n)$ that maintains the closest pair in $O(\log^D n)$ amortized time per deletion.

4.2 A fully dynamic data structure with sublinear update time

Since a planar Voronoi diagram can be maintained in $O(n)$ time per insertion and deletion (see Overmars [96, 97] and Aggarwal et al. [4]), we can also maintain the closest pair in a planar point set in $O(n)$ time per update. The first fully dynamic

26

It is based on the following idea. Instead of maintaining only the minimum distance, we start with the sorted list L of n smallest distances. If a point p is deleted then all distances in L in which p "occurs" have to be deleted. It can easily be shown that there are at most $O(\sqrt{n})$ such distances. Hence, during $\Theta(\sqrt{n})$ deletions the list L will be non-empty, and it will contain the minimum distance of the current set. Once L gets empty, we compute a new list and continue as above. The initial list L can be computed in $O(n \log n)$ time. The distances that have to be deleted from L because of a deletion can be found using a space efficient variant of the range tree. (See Smid [119].) Using the same range tree, insertions of points can also be supported easily. The resulting data structure maintains the closest pair in $O(\sqrt{n} \log n)$ amortized time per insertion and deletion, and it uses $O(n)$ space. Using standard techniques, the update time can be made worst-case.

4.3 A solution with polylogarithmic update time

The first fully dynamic closest pair data structure having polylogarithmic update time is due to Smid [121]. We describe it in detail here, because the same technique can be used for solving other problems, such as dynamic k-point clustering problems and constructing spanners. At this moment, it is not clear if other dynamic closest pair data structures can be used for solving these other problems.

The description of the data structure given below follows Arya and Smid [15]; it is simpler than that in [121]. For simplicity, we assume in this section that $D = 2$. The data structure is based on the *range tree*, see Lueker [89] and Willard and Lueker [127].

Let S be a set of n points in the plane. Our data structure has the form of a 3-layered range tree. There is a balanced binary search tree—called the *layer-1 tree*—storing the points of S in its leaves, sorted by their x-coordinates. Let v be any node of this tree and let S_v be the subset of S that is stored in the subtree of v. Then v contains a pointer to the root of a balanced binary search tree—called a *layer-2 tree*—storing the points of S_v in its leaves, sorted by their y-coordinates.

Before we can define the third layer of the data structure, we need to introduce some notation. Let u be any node of a layer-2 tree. Let r be the root of this layer-2 tree. Then the node of the layer-1 tree that contains a pointer to r will be denoted by u'. Moreover, let x_u be a real number that is between the maximum x-coordinate in the left subtree of u' and the minimum x-coordinate in the right subtree of u'. Similarly, let y_u be a real number that is between the maximum y-coordinate in the left subtree of u and the minimum y-coordinate in the right subtree of u. We denote the point with coordinates (x_u, y_u) by o_u.

Now we can define the third layer of the data structure. Let u be any node of any layer-2 tree, and consider the corresponding point o_u. Let S_u be the set of points that is stored in the subtree of u. Moreover, for $1 \le i \le 4$, let S_{ui} be the subset of S_u that is contained in the i-th quadrant w.r.t. the point o_u. Finally, for $1 \le i \le 4$, let S'_{ui} be the subset of S_{ui} consisting of the five points that are closest to o_u, measured in the L_∞-metric. Then node u contains pointers to

1. four balanced binary search trees. For $1 \le i \le 4$, the i-th tree contains the

27

2. a variable $\eta_3(u)$ having value

$$\eta_3(u) = d\left(\bigcup_{i=1}^{4} S'_{ui}\right),$$

3. and, in case, $\eta_3(u) < \infty$, a pair of points of $\bigcup_i S'_{ui}$ that realizes $\eta_3(u)$.

We are almost done with the description of the data structure. We saw that for each layer-3 structure there is a corresponding η_3-value. Let $1 \leq i \leq 2$ and let v be any node of a layer-i tree. If v is a leaf then v stores a variable $\eta_i(v)$ having value ∞. If v is not a leaf, then let v_l and v_r be the left and right child of v, respectively. Also, let $\eta_{i+1}(v)$ be the variable that is stored with the layer-$(i+1)$ structure that corresponds to v. Then node v stores a variable $\eta_i(v)$ having value

$$\eta_i(v) = \min(\eta_i(v_l), \eta_i(v_r), \eta_{i+1}(v)), \tag{2}$$

and, in case $\eta_i(v) < \infty$, a pair of points that realizes $\eta_i(v)$. This concludes the description of the data structure.

Lemma 4.1 *Let η be the value that is stored with the root of the layer-1 tree. Then $\eta = d(S)$.*

Proof: First note that all $\eta_i(\cdot)$-variables have value either ∞ or $d(A)$ for some subset A of S. Therefore, $d(S) \leq \eta$.

The entire data structure contains many layer-3 structures as substructures. If we can show that the η_3-variable of one of these layer-3 structures has value $d(S)$, then it follows that $\eta \leq d(S)$ and, hence, $\eta = d(S)$.

We show that such a layer-3 structure exists. Consider the closest pair P, Q of S. Let u_1 be the highest node in the layer-1 tree such that P and Q are contained in different subtrees of u_1. Similarly, let u_2 be the highest node in the layer-2 tree that is pointed to by u_1 such that P and Q are contained in different subtrees of u_2.

Node u_2 contains a pointer to a layer-3 structure DS. We claim that the variable η_3 that is stored with DS has value $d(S)$. This will complete the proof of the lemma.

Consider the point o_{u_2}. We assume w.l.o.g. that this point is the origin of our coordinate system. For $1 \leq i \leq 4$, let S_{u_2i} be the set of points stored in DS that are contained in the i-th quadrant. Note that P and Q are both contained in $\bigcup_i S_{u_2i}$. For $1 \leq i \leq 4$, consider the subset S'_{u_2i} of S_{u_2i}. Note that $\eta_3 = d(\bigcup_i S'_{u_2i})$. If we can show that P and Q are both contained in $\bigcup_i S'_{u_2i}$, then we must have $\eta_3 = d(S)$.

Assume this is not the case. Moreover, assume w.l.o.g. that P does not belong to $\bigcup_i S'_{u_2i}$, and that P lies in the first quadrant, i.e., all coordinates of P are positive.

Note that S'_{u_21} contains five points. Let δ be the maximum L_∞-distance between any point of S'_{u_21} and the origin. Then, S'_{u_21} is contained in the box $[0 : \delta]^2$. Partition it into four subboxes, each with sides of length $\delta/2$. One of these subboxes contains at least two points of S. Hence, $d(S) \leq \delta$.

$\ell \in \{1, 2\}$ be an index such that P_ℓ, i.e., the ℓ-th coordinate of P, is at least equal to δ. Our choice of the nodes u_1 and u_2 implies that the ℓ-th coordinate of Q is negative. Hence, the distance between P and Q is greater than δ, which implies that $\delta < d(S)$. This is a contradiction. Hence, we have shown that P and Q are both contained in $\bigcup_i S'_{u_2 i}$. This completes the proof. ∎

Hence, this data structure stores the minimum distance of our point set S. To insert or delete a point p, we search for the x-coordinate of p in the layer-1 tree. For each node on the search path, we search for the y-coordinate of p in the corresponding layer-2 tree. For each node encountered, we update the corresponding layer-3 structure. Afterwards, we walk back along all these paths, and update the values $\eta_i(\cdot)$ according to (2). To guarantee that the data structure remains balanced, we take the binary trees from the class of BB$[\alpha]$-trees, and use rotations. (See Mehlhorn [92, page 198].) In this way, the amortized update time of the entire data structure is bounded by $O(\log^3 n)$. It is easy to show that the structure uses $O(n \log^2 n)$ space. It turns out that the orderings used in the layer-3 structures are similar enough so that dynamic fractional cascading [94] can be applied. In this way, the amortized update time is reduced to $O(\log^2 n \log \log n)$. For details, the reader is referred to [121].

Using standard techniques, this result can be generalized to any fixed dimension D. The result is a data structure of size $O(n \log^D n)$ that maintains the closest pair in $O(\log^D n \log \log n)$ amortized time per insertion and deletion.

In Datta et al. [50], it is shown how the data structure presented in this section can be extended to the dynamic k-point clustering problem, in which we maintain the optimal k-point subset under insertions and deletions of points. (See also Section 2.5.4 and Schwarz [107].) Finally, Arya and Smid [15] use the data structure for constructing spanners. (See Section 6.1.2.)

4.4 A randomized data structure with polylogarithmic update time using linear space

The data structure of the previous section has polylogarithmic update time, but it uses more than linear space. Therefore, after [121] appeared, the goal was to obtain polylogarithmic update time using only $O(n)$ space. This goal was achieved by Golin et al. [70], in the randomized sense. Basically, they extend the static closest pair algorithm of Khuller and Matias given in Section 2.4.2. As in the algorithm of Khuller and Matias, the data structure is built by making a sequence of iterations. In order to make the algorithm dynamic, however, we maintain information that is computed during this sequence.

The data structure is defined by the following randomized procedure. Let $S_1 := S$ and $i := 1$. During the i-th iteration, we pick a random point $p_i \in S_i$—called the *pivot*—and compute its nearest neighbor q_i in S_i. Let d_i be the distance between p_i and q_i. Using perfect hashing, we store the points of S_i in a $d_i/(4D)$-grid. Then, we determine the set S'_i consisting of all points of S_i that do not contain any other points of S_i in their neighborhoods. Again using perfect hashing, we store the set S'_i in a

$$d_i^*(p) := \min \left(d_i, d(p, \bigcup_{1 \leq j \leq i} S_j') \right).$$

We store all these values in a heap H_i. If $S_i = S_i'$, then the algorithm stops. Otherwise, we set $S_{i+1} := S_i \setminus S_i'$ and $i := i + 1$, and proceed with the next iteration.

The results of Section 2.4.2 imply that $d_{i+1} \leq d_i/2$. Let ℓ be the number of iterations made by the algorithm. Then $d_\ell/(4D) \leq d(S) \leq d_\ell$. Finally, the entire algorithm takes linear expected time.

It is not difficult to see that the minimum distance in the overall set S is equal to the smallest element contained in any of the heaps H_i, $1 \leq i \leq \ell$. However, we can say more. First, we have

$$d_i^*(p) = \min \left(d_i, d(p, S_{i-D}' \cup S_{i-D+1}' \cup \ldots \cup S_i') \right).$$

Hence, to compute such a value $d_i^*(p)$, it suffices to perform a constant number of search operations in only $D + 1$ grids (rather than in i grids). Therefore, this value can be computed in constant time.

Second, the minimum distance in S is equal to the smallest element contained in any of the heaps $H_{\ell-D}, H_{\ell-D+1}, \ldots, H_\ell$.

What happens if we insert a new point q into S? Our goal is to update the data structure in such a way that at the end it "looks like" it has been built by the above procedure. That is, the updated data structure should have the same statistical properties as a data structure for $S \cup \{q\}$ that has been built from scratch using the above procedure.

Here is a brief and intuitive description of the insertion algorithm. By assumption, p_1—the pivot of S_1—is a random element of $S_1 = S$. To generate a pivot for $S_1 \cup \{q\}$, it suffices to retain p_1 as pivot with probability $|S_1|/(|S_1| + 1)$ and to choose q instead with probability $1/(|S_1| + 1)$. If q is chosen, then we discard everything and build a completely new data structure using the above procedure. In this case, the insertion algorithm terminates. Note that this happens, however, only with probability $1/(|S_1| + 1)$ and so the expected cost is $O(1)$.

Assume now that p_1 remains unchanged as the pivot. We now check if q_1—the nearest neighbor of p_1—and, hence, d_1 have to be changed. It is known that q can be the nearest neighbor of at most 3^D points in S_1. (See [51].) This means that d_1 changes only if p_1 is one of these points. Since p_1 was chosen uniformly from S_1, it follows that the probability of d_1 changing is at most $3^D/|S_1|$. If d_1 changes, we use the above algorithm to build a completely new data structure and terminate the procedure. The expected cost of this is $O(1)$.

We are left with the case where p_1, q_1 and d_1 remain unchanged. Let us denote $S \cup \{q\}$ by $\widetilde{S}$. We need to determine the set $\widetilde{S}_2$ containing the points in $\widetilde{S}_1 = \widetilde{S}$ that contain some other points in their neighborhood. If q does not contain points of S_1 in its neighborhood, then it will go into $\widetilde{S}_1'$, and nothing further needs to be done. In this case, we can terminate the procedure. Otherwise, $\widetilde{S}_2$ contains q and possibly some points from S_1'. The set of points which are deleted from S_1' due to the insertion of q

such that $D_2 \cup \{q\}$ is inserted into S_3, etc.

Hence, during the insertion algorithm, certain points move down some levels in the data structure. It can be shown that the total number of points that move is bounded by a constant. In particular, the total number of query and update operations in the grids is bounded by $O(\log n)$, and the total number of update operations in the heaps is bounded by a constant. Hence, if we implement the grids using dynamic perfect hashing [55], the entire insertion algorithm takes $O(\log n)$ expected time.

The deletion algorithm is basically the reverse of the insertion algorithm. In particular, the points that move to lower levels during an insertion of q move back to their previous locations when q is deleted directly afterwards. For details, we refer the reader to [70, 107]. There it is shown that the expected time for a deletion is also bounded by $O(\log n)$, if we use dynamic perfect hashing.

The result is a randomized data structure that stores a set S of n points in $\mathbb{R}^D$ such that the minimum distance $d(S)$ can be found in $O(1)$ time. The expected size of this structure is $O(n)$, and we can maintain it under insertions and deletions in $O(\log n)$ expected time per update. The algorithms as presented here assume that the floor function is available at unit cost. Also, because we use dynamic perfect hashing, it is assumed that all points come from a bounded region in $\mathbb{R}^D$.

Note that in this result randomization is only w.r.t. the coin flips that are made during the algorithms. In particular, no assumption is made about the points that are inserted and deleted (except that they come from a bounded region); these can be arbitrary.

We can use degraded grids to remove the floor function from the machine model. In this way, searching and updating a grid takes $O(\log n)$ time. As a result, we get a randomized data structure having $O(n)$ expected size that maintains the closest pair in $O(\log^2 n)$ expected time per insertion and deletion. (The details of this modified data structure are tedious, see [70, 107].) This data structure can be implemented in the randomized algebraic computation tree model. We finally remark that for the data structures presented in this section high probability bounds can be proved. (See [70, 107].)

4.5   A deterministic solution with polylogarithmic update time using linear space

The first deterministic data structure that maintains the closest pair in polylogarithmic time using only linear space is due to Kapoor and Smid [81]. They first give a deterministic data structure for maintaining the closest pair that improves the result of Section 4.3. Then they present a "space saving transformation", which transforms any dynamic closest pair data structure of super-linear size into another one that uses less space. Applying this transformation several times leads to a dynamic closest pair data structure using only linear space.

To describe the first result in [81], we need to introduce some notation. Let S be a set of n points in $\mathbb{R}^D$. For any point $p \in \mathbb{R}^D$, $box(p)$ denotes the smallest axes-parallel cube centered at p that contains at least $(2D + 2)^D$ points of $S \setminus \{p\}$. In other words,

(resp. $(2D+2)^D$-th) L_∞-neighbor, if $p \in S$ (resp. $p \notin S$). Also, for any point $p \in \mathbb{R}^D$, $N(p)$ denotes the set of points of $S \setminus \{p\}$ that are contained in the interior of $box(p)$.

The data structure is based on the following lemma, whose proof is straightforward.

Lemma 4.2 *The set $\{(p,q) : p \in S, q \in N(p)\}$ contains a closest pair of S.*

This lemma implies that it suffices to maintain the distances $d(p,q)$, $p \in S$, $q \in N(p)$, in a heap. The smallest element contained in this heap is the minimum distance in S. Note that the heap stores only a linear number of elements. In order to maintain the heap—if points are inserted and deleted in S—we also have to maintain the set $\{box(p) : p \in S\}$.

Let us see what has to be done when we insert a new point p into S. We must be able to compute $box(p)$, and the corresponding set $N(p)$. These can be found by computing the $(2D+2)^D$ L_∞-neighbors of p. In Section 5.1.2, we will see that the standard *range tree* can be used for this. We also have to find all boxes $box(q)$ that contain p. (There are at most a constant number of such boxes.) Each such box has to replaced by a smaller box—one that contains $(2D+2)^D$ points of the new point set. To support these operations, we can use the standard *segment tree* (see [99]) or a dynamic variant of the *skewer tree* (see [60, 123]), extended to support dynamic fractional cascading [94]. The efficiency of the query algorithms in these data structures depends on the following fact: The set $\{box(q) : q \in S\}$ is of *constant overlap*, in the sense that each box contains the centers of at most $(2D+2)^D$ boxes in its interior. These centers are precisely the points of $N(q) \cup \{q\}$. Using the same data structures, we can efficiently update the heap when we delete a point from the set S.

Assume first that we use a segment tree to store the elements of the set $\{box(q) : q \in S\}$. During the insertion or deletion of a point in S, we perform at most a constant number of query and update operations in the range tree, the segment tree and the heap. Therefore, the amortized update time of the entire data structure is bounded by $O(\log^{D-1} n \log \log n)$. Moreover, the size of the data structure is bounded by $O(n \log^{D-1} n)$. (See [81].)

To reduce the space complexity, some additional techniques are needed. We first remark that in the solution just sketched both the range tree and the segment tree use $O(n \log^{D-1} n)$ space. We need the range tree for computing the boxes $box(p)$. Unfortunately, no linear space solution for the latter problem having polylogarithmic query and update times is known. Hence, in order to reduce the space complexity, we should avoid using the range tree. We can replace the segment tree by a skewer tree—which uses only linear space. Then, however, the update time increases slightly.

In [81], a transformation is presented that, given *any* dynamic closest pair data structure DS having more than linear size, produces another dynamic closest pair structure that uses less space. The transformed data structure is composed on two sets A and B that partition the current point set S. The boxes $box(p)$ of the points in A remain fixed during a sequence of updates—hence we do not need the range tree for this set—whereas the set B is stored in the dynamic data structure DS. To reduce space, B is a subset of the entire set and contains points involved in $o(n)$ updates

periodically rebuilt.

The result is as follows. Let DS be any data structure for the dynamic closest pair problem. Let $S(n)$ and $U(n)$ denote the size and (amortized or worst-case) update time of DS, respectively. Let $1 \leq f(n) \leq n/2$ be any non-decreasing integer function. Assume that $S(n)$, $U(n)$ and $f(n)$ are smooth. (A function g is called *smooth* if $g(\Theta(n)) = \Theta(g(n))$.) The transformation produces a data structure for the dynamic closest pair problem having

1. size $O(n + S(f(n)))$, and

2. an amortized update time of $O(\log^{D-1} n + U(f(n)) + (n \log n)/f(n))$.

Applying this transformation twice to the above closest pair data structure, taking $f(n) = n/(\log^{D-2} n \log \log n)$ and $f(n) = n \log \log n / \log n$, respectively, we get a data structure of size $O(n)$ that maintains the closest pair in $O(\log^{D-1} n \log \log n)$ amortized time per insertion and deletion, for $D \geq 3$.

For $D = 2$, we can apply the transformation k times for any constant k, with appropriate choices for the function f. This gives data structures of sizes $O(n)$ and $O(n \log n/(\log \log n)^k)$, having amortized update times of $O(\log^2 n/(\log \log n)^k)$ and $O(\log n \log \log n)$, respectively.

4.6 A dynamic solution based on simplicial cones

This section describes a solution to the dynamic closest pair problem that is due to Bespamyatnikh [27]. The main idea is to reduce the entire problem to certain range searching problems, and then use standard data structures to solve the latter. (This technique appeared for the first time in Gabow, Bentley and Tarjan [68]. They use it for solving the post-office problem for "simple" metrics. See Section 5.1.1.)

For simplicity, we only consider the planar case. So, S denotes a set of n points in the plane. Let k be a sufficiently large integer constant, and let $\theta = 2\pi/k$. Rotate the positive x-axis over angles $i \cdot \theta$, $0 \leq i < k$. This gives k rays. Let $C_1, C_2, \ldots, C_k$ be the cones that are bounded by any two successive rays. Also, for $1 \leq i \leq k$, let l_i be a fixed ray that emanates from the origin and that is contained in C_i.

For each $1 \leq i \leq k$, we define an approximate distance function δ_i, as follows. Let p and q be two points in the plane. If $q - p$ is not contained in the cone C_i, then $\delta_i(p, q) := \infty$. Otherwise, if $q - p \in C_i$, then $\delta_i(p, q)$ is defined as the L_r-distance between the origin and the orthogonal projection of $q - p$ onto the ray l_i. (Note that in general, $\delta_i(p, q) \neq \delta_i(q, p)$.)

These distance functions have the following property. Let p, q and r be three points such that $q - p$ and $r - p$ are both contained in the cone C_i. Also, assume that $\delta_i(p, r) \leq \delta_i(p, q)$. Then $d(r, q) < d(p, q)$. (For a proof of this, see [14, 27, 102].)

For $1 \leq i \leq k$, let E_i be the set of all pairs (p, q) such that

$$\delta_i(p, q) = \min\{\delta_i(p, r) : r \in S \setminus \{p\}\},$$

and

$$\delta_i(q, p) = \min\{\delta_i(q, r) : r \in S \setminus \{q\}\},$$

contains the closest pair. Hence, in order to maintain the closest pair, it suffices to maintain the pairs of E in a heap, where the ordering is by their L_τ-distances.

Let p be any point in the plane, let $1 \leq i \leq k$, and let C_i' be the cone C_i translated such that its apex is at p. A point $q \in S \setminus \{p\}$ such that $\delta_i(p,q)$ is minimal can be computed by selecting among all points that are between the two bounding rays of C_i' a point that is furthest to the "left", where we consider the ray l_i as being "horizontal". Hence, we can use range searching techniques to compute such a point q.

Unfortunately, it seems to be difficult to maintain the set E efficiently. However, Bespamyatnikh shows that a subset of E that still contains the closest pair can be maintained such that during each insertion and deletion at most a constant number of query and update operations on the range searching data structure have to be performed.

This technique can be generalized to any fixed dimension D by using simplicial cones of angular diameter θ. Each such cone is bounded by D hyperplanes and, hence, again range searching methods can be applied. Moreover, a constant—depending on the dimension D—number of such cones suffice. For an explicit construction of these cones, see Bespamyatnikh [27] or Yao [128].

If we use the standard range tree, then we get a dynamic closest pair data structure of size $O(n \log^{D-1} n)$ with an update time of $O(\log^D n)$. On the other hand, if we use a data structure due to Chazelle [37], then we get a data structure of size $O(n \log^{D-2} n)$ that maintains the closest pair in $O(\log^{D+1} \log \log n)$ time per insertion and deletion.

4.7 A dynamic solution based on the well-separated pair decomposition

We saw already in Sections 2.5.2 and 2.5.3 that the well-separated pair decomposition (WSPD) can be used to solve several proximity problems. In order to apply it to dynamic problems, we clearly need efficient ways to update the WSPD. Callahan and Kosaraju [33, 30] show how to maintain the fair split tree of Section 2.5.2 in logarithmic time per insertion and deletion, using linear space. The main idea is very similar to one developed by Bespamyatnikh [28], and that will be described in the next section. Therefore, we only give the latter solution.

One of the main problems in maintaining information such as the closest pair is the fact that there may be points that are contained in many pairs. Callahan and Kosaraju show how to get around this problem by introducing *dummy* points. They also present an abstract framework that describes the main properties of algorithms that use the WSPD, and show how to maintain computations in this framework. They show that the problem of maintaining the closest pair fits into this framework. As a result, they get a dynamic closest pair data structure of linear size and $O(\log^2 n)$ update time.

4.8 An optimal dynamic closest pair data structure

The problem of designing an optimal dynamic closest pair data structure was solved in 1995 by Bespamyatnikh [28]: He shows how to maintain the closest pair in $O(\log n)$

algebraic computation tree model and are, therefore, optimal.

An important ingredient used is a hierarchical subdivision of $\mathbb{R}^D$ into axes-parallel rectangles. Before we can define this subdivision, we need some definitions.

Let B be an axes-parallel rectangle, and for $1 \leq i \leq D$, let s_i be the side length of B along the i-th dimension. We call B a *c-box* if $1/3 \leq s_i/s_j \leq 3$ for all $1 \leq i, j \leq D$.

Let $B = [a_1, b_1] \times \ldots \times [a_D, b_D]$ be a rectangle, let $1 \leq i \leq D$, and let $c_i \in [(2a_i + b_i)/3, (a_i + 2b_i)/3]$. The two boxes that are obtained from B by replacing the i-th interval by $[a_i, c_i]$ and $[c_i, b_i]$, respectively, are set to be *obtained from B by a c-cut*.

Let A and B be two axes-parallel rectangles. We say that A is an *s-subbox* of B if they are equal or there exists a sequence $B_0, B_1, B_1', \ldots, B_l, B_l'$, for some $l \geq 1$, such that $B = B_0$, $A = B_l$, and for $1 \leq i \leq l$, the boxes B_i and B_i' are obtained from B_{i-1} by a c-cut.

The hierarchical subdivision is represented by a binary tree T. With each node v in T, we store two boxes $B(v)$ and $SB(v)$. During the algorithms, the following conditions are satisfied.

1. For any node v of T, $B(v)$ and $SB(v)$ are c-boxes.

2. For any node v of T, $SB(v)$ is an s-subbox of $B(v)$.

3. For any node v of T, $SB(v)$ and $B(v)$ contain the same points of S.

4. If w is any internal node of T, and u and v are the two children of w, then the boxes $B(u)$ and $B(v)$ are obtained from $SB(w)$ by a c-cut.

5. For any leaf v of T, $B(v) = SB(v)$ and these boxes contain exactly one point of S.

If v is a leaf of T and p is the point of S that is contained in $B(v)$, then $B(p)$ will denote the box $B(v)$.

Let us see how we can maintain the tree T if we insert and delete points in S. Assume we want to delete the point p. Then, by walking down T, we find the leaf u such that $p \in B(u)$. Let w be the parent of u, and let v be the other child of w. If v is a leaf, then we set $SB(w) := B(w)$ and delete the leaves u and v. Otherwise, if v is not a leaf, then we delete the leaf u, set $B(v) := B(w)$, and delete node w by making v a child of w's parent.

Insertions are more difficult to describe. Assume we want to insert a point p into S. Then we walk down T and stop if one of the following three cases occurs: (i) p is not contained in $B(r)$ where r is the root of T, (ii) we reach a leaf w and p is contained in $B(w)$, or (iii) we reach an internal node w and p is contained in $B(w)$ but not in $SB(w)$.

We explain how the third case can be handled. (The other cases are easier.) Our goal is to find a c-box A together with two boxes A_1 and A_2 obtained from A by a c-cut, such that (i) A is an s-subbox of $B(w)$, (ii) $SB(w)$ is an s-subbox of A_1, and (iii) p is contained in A_2. Given this box A, we update the tree T as follows: Let w' and w'' be the two children of w. We give w two new children called u and v, and set

and w'' as its two children.

It is easy to see that in this way the tree T is correctly maintained. So it remains to describe how the box A is computed. This is done by performing at most $2D$ iterations. Initially, we set $A := B(w)$. We maintain the invariant that (i) A is a c-box and an s-subbox of $B(w)$, (ii) $SB(w)$ is an s-subbox of A and (iii) p is contained in A. We describe one iteration.

Let i be an index such that the side of A along the i-th dimension is maximal. Let $[a_i, b_i]$ (resp. $[d_i, e_i]$) be the interval of $SB(w)$ (resp. A) along the i-th dimension. If a_i (resp. b_i) is contained in the interval $[(2d_i + e_i)/3, (d_i + 2e_i)/3]$, then we set $x_i := a_i$ (resp. $x_i := b_i$). Otherwise, the definition of c-cut implies that the interval $[a_i, b_i]$ does not intersect the interval $[(2d_i + e_i)/3, (d_i + 2e_i)/3]$. Assume w.l.o.g. that $[a_i, b_i]$ is contained in $[d_i, (2d_i + e_i)/3]$. In this case, we set $x_i := \max((2d_i + e_i)/3, 2b_i - d_i)$.

Now we partition the box A along the i-th dimension using the value x_i. If one of the two resulting boxes contains both $SB(w)$ and p, then we take this box as the new A and proceed with the next iteration. Otherwise, $SB(w)$ and p are contained in two different subboxes of A. In this case, we have found the box A we were looking for, and the iteration can terminate.

This completes the description of the hierarchical subdivision and the way it is updated. The time needed to insert or delete a point into T is bounded by a constant plus the time for walking down T in order to find the node in which the local changes have to be made.

How do we use this tree T for maintaining the closest pair? A pair (p, q) of points in S is called a *neighbor pair* if p is the nearest neighbor of q, and q is the nearest neighbor of p. We will maintain a set E consisting of $O(n)$ pairs of points that contains all neighbor pairs. Clearly, E then contains the closest pair. Hence, if we store all distances between the pairs of E in a heap, then the smallest element in this heap is equal to the minimum distance in S.

We denote the diameter of a box B, i.e., the maximum distance between any two points of $B \cap \mathbb{R}^D$, by $diam(B)$. Also, the minimum (resp. maximum) distance between a point p and any point of $B \cap \mathbb{R}^D$ is denoted by $d^m(p, B)$ (resp. $d^M(p, B)$).

During the algorithm, the following invariant will hold: For any $p, q \in S$, $p \neq q$, such that the pair (p, q) is not contained in E, there is a node v in the tree T, such that

1. $B(p) \cap B(v) = \emptyset$,

2. $diam(B(v)) \leq 2 \cdot diam(B(p))$,

3. $d^m(p, B(v)) \leq 3 \cdot diam(B(v))$,

4. $d^M(p, B(v)) < d(p, q)$.

It is clear that any such set E contains all neighbors pairs.

For each point p of S, let E_p denote the set of all points $q \in S$ such that the pair (p, q) belongs to E. In order to guarantee that the set E that is maintained has linear size, Bespamyatnikh proves the following result.

of S, and assume that E_p contains more than $(27D + 1)^D$ elements. Then there is a point $q \in E_p$ such that the invariant also holds for the set $E \setminus \{(p, q)\}$.

Hence, there is a set E that satisfies the invariant such that each point occurs in at most a constant number of pairs of E. This set E clearly has linear size. Of course, the problem is how to maintain such a set.

Assume we want to delete a point p from S. First, we delete all distances $d(p, q)$, $q \in E_p$, from the heap, and delete the set E_p itself. Then, we update the tree T as described above. Note that we delete two nodes from T. Consider the deletion of a node v. In order to restore the invariant, assume that before the deletion of p, the pair (a, b) was not contained in E, because of the existence of node v. That is, we have $diam(B(a)) \geq diam(B(v))/2$ and $d^m(a, B(v)) \leq 3 \cdot diam(B(v))$. It follows from a simple packing argument that the number of such points a is bounded by a constant. We denote this set by $A(v)$, i.e.,

$$A(v) := \{a \in S : diam(B(a)) \geq diam(B(v))/2 \text{ and } d^m(a, B(v)) \leq 3 \cdot diam(B(v))\}.$$

Using the tree T, this set can be computed by searching in the neighborhood of the box $B(v)$. Then for each point a in $A(v)$, we again use T and search in the neighborhood of a to compute its set E_a, and insert the corresponding distances into the heap.

The algorithm for inserting a point p is similar. Now, we have to compute the set E_p and insert the new distances into the heap. At this moment, the invariant holds, but there may be points that occur more than $(27D + 1)^D$ times in E. These points must be in the neighborhood of p and, therefore, have been found already. For these points, we remove the corresponding pairs from E and the heap.

To implement the algorithm efficiently, we use dynamic trees to represent the tree T, in a similar way as in Section 3.3. For details, we refer the reader to [28]. There, it is shown that the entire algorithm for inserting or deleting a point takes $O(\log n)$ time in the worst case. Moreover, the entire data structure uses linear space.

5   The post-office problem

We now turn to Problem 1.2, the post-office problem. Let S be a set of n points in $\mathbb{R}^D$. We can solve the post-office problem as follows. Construct the Voronoi diagram of S. Then, given any query point $p \in \mathbb{R}^D$, its nearest neighbor is found by locating the Voronoi region that contains p. The point of S associated with this region is p's nearest neighbor.

In order to implement this solution efficiently, we need fast algorithms for (i) constructing the Voronoi diagram, and (ii) solving point location queries. In the planar case, the Voronoi diagram can be constructed in $O(n \log n)$ time. Also, a data structure can be constructed in $O(n \log n)$ time such that point location queries can be solved in logarithmic time. The entire data structure uses linear space. This gives an optimal solution for the planar post-office problem. In higher dimensions, the situation is more complicated, because the Voronoi diagram of n points in $\mathbb{R}^D$ can have size $\Theta(n^{\lceil D/2 \rceil})$.

37

to the chapter on Voronoi diagrams in this handbook. We just mention here that the best results currently known either use a large amount of space, or their query time is almost linear. More precisely, Clarkson [42] gives a randomized data structure 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 [11], Arya and Mount solve the problem with an expected query time of $O(n^{1-1/\lceil(D+1)/2\rceil}\log^{O(1)}n)$ using $O(n\log\log n)$ space.

It seems that in dimension $D > 2$ it is impossible to obtain a solution for the post-office problem having polylogarithmic query time while using $O(n\log^{O(1)}n)$ space. Moreover, even in the planar case there is no dynamic data structure known that has polylogarithmic query and update times and that uses $O(n\log^{O(1)}n)$ space.

In view of these negative results, it is natural to study weaker versions of the post-office problem. Two such weaker versions have been considered in the literature. In the first version, distances are measured in the "simple" L_1- or L_∞-metric instead of the "complicated" L_τ-metric. In the second version, it suffices to find an *approximate* nearest neighbor of the query point rather than the *exact* nearest neighbor. In the next sections, we consider both versions of the post-office problem in detail.

5.1 The post-office problem for simple metrics

5.1.1 A solution based on the quadrant approach

In this section, we first consider the post-office problem for the L_1-metric. The basic approach is to reduce the problem to that of "range searching for minimum". This approach was used for the first time by Gabow, Bentley and Tarjan [68]. Later, it was used again by Bespamyatnikh [27]. In fact, we used this approach already in Section 4.6.

Let us describe the planar case first. Let $p = (p_1, p_2)$ be any query point. Consider the four quadrants defined by this point. Clearly, if we have the L_1-neighbor of p in each of these quadrants, then we can compute the overall L_1-neighbor of p in constant time. The L_1-neighbor in, say, the north-east quadrant of p is determined as follows: Among all points that are above the line $y = p_2$ and to the right of the line $x = p_1$, it is that point $q = (q_1, q_2)$ for which $q_1 + q_2$ is minimal. It follows that we can apply standard range searching techniques to solve the problem.

The D-dimensional version of the L_1-post-office problem is solved by using 2^D data structures. For $1 \le i \le 2^D$, the i-th data structure is used to find the L_1-neighbor of the query point p that is contained in its i-th quadrant. Gabow, Bentley and Tarjan [68] give a static data structure for this problem that uses $O(n\log^{D-1}n)$ space and has a query time of $O(\log^{D-1}n)$. Alternatively, we can use the standard range tree to get a dynamic solution. This data structure also uses $O(n\log^{D-1}n)$ space. Its query and update times are bounded by $O(\log^D n)$. (Chan and Snoeyink [35] also obtain the latter bounds, using a variation of the technique of [27, 68].) Finally, using Chazelle's data structure [37], we get a data structure of size $O(n\log^{D-2}n)$ that has query and update times of $O(\log^{D+1}\log\log n)$.

The same approach works for the L_∞-metric. This is clear for $D = 2$, because this

38

case, Gabow, Bentley and Tarjan reduce the problem to $2^D \cdot D!$ subproblems of range searching for minima. Bespamyatnikh [27] reduced this number to $2 \cdot D!$. Hence, for this metric, the same complexity can be obtained as for the L_1-metric.

5.1.2   A solution for the L_∞-metric that uses range trees

In this section, we show that the standard range tree [89, 127] can be used for solving the L_∞-post-office problem. The query algorithm is due to Kapoor and Smid [81]. Finding the L_∞-neighbor of a query point p can be visualized by growing an axes-parallel cube centered at p until its boundary hits at a point of S. As we will see, this growing process can be simulated by the range tree.

We describe the query algorithm for the planar case. Let S be a set of n points in the plane. First, we recall the range tree. This data structure consists of a *main tree*, which is a balanced binary search tree storing the points of S in its leaves, sorted by their x-coordinates. If v is a node of this main tree, then S_v denotes the subset of S that is stored in the subtree of v. Every node v of the main tree contains a pointer to the root of a balanced binary search tree—called a *secondary tree*—storing the points of S_v in its leaves, sorted by their y-coordinates.

Let $p = (p_1, p_2)$ be any query point. We give an algorithm for computing the L_∞-neighbor of p among all points that are to the right of p. We call this neighbor the *right-neighbor* of p. In a symmetric way, the L_∞-neighbor to the left of p can be computed. Clearly, given these two neighbors, we can determine the overall L_∞-neighbor of p in constant time.

In the first stage, we decompose the set of all points of S that are to the right of p into $O(\log n)$ pairwise disjoint subsets: Initialize $M := \emptyset$. Starting in the root of the main tree, search for the leftmost leaf storing a point whose x-coordinate is at least equal to p_1. During this search, each time we move from a node v to its left child, add the right child 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. Number the nodes of the final set M as $v_1, v_2, \ldots, v_m$, such that v_i is closer to the root than v_{i-1}, $2 \leq i \leq m$. It is easy to see that $\{r \in S : r_1 \geq p_1\} = \bigcup_{i=1}^m S_{v_i}$. Hence, at this moment, we know that one of the sets S_{v_i} contains the right-neighbor of p.

In the second stage of the algorithm, we make a sequence of at most m iterations. To start this sequence, we initialize $C := \emptyset$, $i := 1$, and *stop* := *false*. Then, as long as $i \leq m$ and *stop* = *false*, we do the following.

Search with p_2 in the associated structure of v_i, and find the point, call it a, whose y-coordinate is nearest to that of p. Then, find the point r that is stored in the rightmost leaf in the main subtree of v_i. Let $\delta := r_1 - p_1$ and let R be the rectangle $[p_1, r_1] \times [p_2 - \delta, p_2 + \delta]$. If a is not contained in R, then insert this point into the set C, and increase i by one. Otherwise, if $a \in R$, set $v := v_i$ and *stop* := *true*.

This concludes the description of the second stage. First assume that the variable *stop* has value *false* after this stage has been completed. Then the set C contains the right-neighbor of p. Hence, in this case we can complete the query algorithm by going through the set C and taking the point having minimum L_∞-distance to p.

39

case, the right-neighbor of p is either contained in C or in S_v, and we proceed with the third—and last—stage of the algorithm.

During the third stage, we again make a sequence of iterations. We maintain the invariant that the set $C \cup S_v$ contains the right-neighbor of p. As long as v is not a leaf, we do the following.

Let w be left child of v. Search with p_2 in the associated structure of w, and find the point, call it a, whose y-coordinate is nearest to that of p. Also find the point r that is stored in the rightmost leaf in the main subtree of w. Let $\delta := r_1 - p_1$ and let R be the rectangle $[p_1, r_1] \times [p_2 - \delta, p_2 + \delta]$.

First assume that a is not contained in R. If the right-neighbor of p is contained in the subtree of w, then it must be equal to a. Therefore, we insert a into the set C, set $v :=$ right child of v, and proceed with the next iteration.

Now assume that a is contained in R. In this case, the right-neighbor of p is not contained in the subtree of the right child of v. Therefore, we set $v := w$ and proceed with the next iteration.

After the last iteration, v is a leaf. Moreover, we know that the right-neighbor of p is either contained in C or stored in v. Hence, we find this right-neighbor by going through the set $C \cup S_v$ and taking the point having minimum L_∞-distance to p.

This concludes the description of the algorithm for finding the right-neighbor of p. For a correctness proof, the reader is referred to [81]. It is easy to see that each of the stages can be implemented in $O(\log^2 n)$ time. Using fractional cascading [38], this can be improved to $O(\log n)$. The range tree can be maintained dynamically in $O(\log n \log \log n)$ amortized time per insertion and deletion. Then, since we need dynamic fractional cascading [94], the query time also becomes $O(\log n \log \log n)$.

The generalization to the D-dimensional L_∞-post-office problem is straightforward. We get a dynamic data structure of size $O(n \log^{D-1} n)$ having $O(\log^{D-1} n \log \log n)$ query time and $O(\log^{D-1} n \log \log n)$ amortized update time. (See [81] for details.)

5.2   The approximate post-office problem

Let S be a set of n points in $\mathbb{R}^D$, and let ϵ be any positive constant. For any point $p \in \mathbb{R}^D$, we denote by p^* the point of S that is closest to p. A point $q \in S$ is called a $(1 + \epsilon)$-*approximate neighbor* of p, if $d(p, q) \leq (1 + \epsilon) d(p, p^*)$.

In this section, we give several data structures for solving this approximate post-office problem. As we will see, for this problem, there are solutions with polylogarithmic query and update times that use $O(n \log^{O(1)} n)$ space. Recall that these bounds seem to be impossible for the *exact* post-office problem; see the discussion in the beginning of Section 5.

5.2.1   A data structure based on quad trees

The first result for the approximate post-office problem is due to Bern [25]. By scaling, we may assume w.l.o.g. that the set S is contained in the cube $[3/8, 7/8]^D$.

We build a *quad tree* for the set S. (See Finkel and Bentley [63].) Each node v of this tree stores an axes-parallel cube, such that all points in the subtree of v are

any node v, and let C_v be its cube. We split C_v into 2^D equal sized subcubes, and give v 2^D children, one for each subcube. A node v is a leaf if C_v contains at most one point of S. With such a leaf, we store the point of S—if it exists—that is contained in C_v.

In order to save space, we remove all nodes whose cubes do not contain any points of S. We also remove all nodes that have only one child, by attaching the child directly to its grandparent. Clearly, this results in a tree having linear size.

We use this data structure in the following way. Consider a query point p. If p is not contained in $[0, 1]^D$, then return any point q of S. Otherwise, follow a path in the tree in the obvious way, and stop in the node v storing the smallest cube containing p. (Note that v is not necessarily a leaf.) Now return any point q that is contained in the subtree of v.

Let us see how good an approximation the returned point q is. First assume that the query point p lies outside $[1/4, 1]^D$. Then we have $d(p, p^*) \geq 1/8$ and $d(p, q) \leq d(p, p^*) + d(p^*, q) \leq d(p, p^*) + D/2$. It follows that $d(p, q)/d(p, p^*) \leq 1 + 4D$, i.e., the returned point q is a $(1 + 4D)$-approximate neighbor of p.

There is one more case where q is a reasonably good approximation. Before we can describe this case, we need some definitions. Consider the cube C_v. Split this cube into 2^D equal sized subcubes. One of these subcubes contains p. We call this subcube the *answer cube* of p. Let A be an axes-parallel cube with sides of length ℓ and center a. The point p is said to be *central* in A if it is contained in the cube centered at a that has sides of length $2\ell/3$.

We claim that q is a good approximation if p is central in its answer cube. To prove this, let p's answer cube have sides of length ℓ. Since q is contained in C_v, we have $d(p, q) \leq 2\ell D$. Moreover, since p is central in A, and since A does not contain any points of S, we have $d(p, p^*) \geq \ell/6$. Hence, $d(p, q)/d(p, p^*) \leq 12D$, i.e., the point q is a $(12D)$-approximate neighbor of p.

Of course, it might be the case that p is not central in its answer cube. To solve this problem, we do the following. For $1 \leq i \leq 2^D$, let $i_1 i_2 \ldots i_D$ be the binary representation of $i - 1$. Let v_i be the vector $(i_1/3, i_2/3, \ldots, i_D/3)$. We build 2^D quad trees, where for $1 \leq i \leq 2^D$, the root cube of the i-th tree is $[0, 1]^D$ translated by v_i. The claim now is that if the query point p is contained in $[1/4, 1]^D$, then p is central in its answer cube in one of these 2^D quad trees.

Hence, given a query point $p \in \mathbb{R}^D$, we query each of the quad trees. If q_i is the point found in the i-th tree, $1 \leq i \leq 2^D$, then we compute the one that is closest to p, and return this point. This point is a $(12D)$-approximate neighbor of p. We remark that for special metrics, the approximation can be much better. For example, if we use the Euclidean metric, then this solution computes a $(12\sqrt{D})$-approximate neighbor of p.

If we use a centroid decomposition to represent the quad trees, see Section 3.3, then the query algorithm takes $O(\log n)$ time. The entire data structure uses linear space. Finally, using an algorithm that is similar to that of Section 2.5.2, the entire data structure can be built in $O(n \log n)$ time.

The first efficient data structure for solving the $(1+\epsilon)$-approximate post-office problem for any constant $\epsilon > 0$ was given by Arya and Mount [11].

The data structure of Arya and Mount is a directed graph that is constructed in a randomized manner. For each point q of S we cover the space $\mathbb{R}^D$ with a constant number of cones having q as their apex. Then we apply the following procedure that adds for each $q \in S$ and each cone C with apex q an expected number of $O(\log n)$ directed edges of the form (q, r), where $r \in C$: Let $r_1, r_2, \ldots, r_{n-1}$ be a random permutation of the point set $S \setminus \{q\}$. For each cone C with apex q, consider all points that are contained in C. If $r_i \in C$, then the data structure contains a directed edge (q, r_i) if r_i is the point of the set $C \cap \{r_1, r_2, \ldots, r_i\}$ having minimum distance to q.

The graph that is constructed in this way is called the *randomized neighborhood graph*. Standard arguments from probability theory imply that each point has expected out-degree $O(\log n)$. Hence, the expected size of the data structure is bounded by $O(n \log n)$. The data structure can be built in $O(n^2)$ time in a straightforward way.

Let $p \in \mathbb{R}^D$ be any query point, and let q be any point of S that is not a $(1 + \epsilon)$-approximate neighbor of p. Arya and Mount show that by choosing the cones such that their angular diameter is small enough, there is an edge (q, r) in the data structure for which $d(p, r) < d(p, q)$. This implies the following algorithm for solving approximate post-office queries. Start in any point q of S. Consider all edges (q, r), and choose the point r that is closest to p. If $d(p, r) < d(p, q)$, then set $q := r$, and repeat this process. Otherwise, if $d(p, r) \geq d(p, q)$, report the point q as being a $(1 + \epsilon)$-approximate neighbor of p.

Unfortunately, it seems very difficult to prove a good bound on the expected running time of this query algorithm. Therefore, Arya and Mount propose the following modification. We associate each directed edge (a, b) of the data structure with the cone having a as its apex and that contains b. A directed path $q_1, q_2, \ldots, q_k$ is called *pseudo-linear*, if for each $1 < i < k$, the cone associated with (q_{i-1}, q_i) contains the cone associated with (q_i, q_{i+1}). Such a path resembles a path in a skip list [100] and, therefore, it is relatively easy to analyze it. In particular, let p be a query point and let q be a point of S that is not a $(1 + \epsilon)$-approximate neighbor of p. Then there is a pseudo-linear path containing an expected number of $O(\log n)$ edges that starts in q and ends in a point r such that $d(p, r) < d(p, q)$. This path can be computed in $O(\log^2 n)$ expected time. Moreover, if N_q (resp. N_r) denotes the number of points x such that $d(p, x) < d(p, q)$ (resp. $d(p, x) < d(p, r)$), then $N_r \leq N_q/2$ with probability at least $1/2$. Hence, after having computed an expected number of $O(\log n)$ such pseudo-linear paths, we have found a $(1+\epsilon)$-approximate neighbor of p. The expected running time of the complete query algorithm is bounded by $O(\log^3 n)$.

Arya and Mount also give a practical variant of this algorithm. They made several experiments with this variation and compared its running time with that of other practical algorithms based on $k - d$ trees and simple bucketing techniques, for dimensions up to 16. For details, we refer to [10, 11].

Kapoor and Smid [81] gave the first efficient dynamic data structure for the $(1 + \epsilon)$-approximate post-office problem. Their algorithm, however, only works for approximating the Euclidean neighbor of a query point.

Let $p \in \mathbb{R}^D$ be any query point, let $p^* \in S$ be the Euclidean neighbor of p, let $q \in S$ be the L_∞-neighbor of p, and let $\delta := d_\infty(p, p^*)$. Consider the axes-parallel cube centered at p having sides of length 2δ. Clearly, q lies inside or on the boundary of this cube. Therefore, $d_2(p, q) \leq \sqrt{D} \cdot \delta$. Since $\delta = d_\infty(p, p^*) \leq d_2(p, p^*)$, we infer that $d_2(p, q) \leq \sqrt{D} \cdot d_2(p, p^*)$, i.e., q is a $\sqrt{D}$-approximate L_2-neighbor of p.

Hence, we can use the range tree and the corresponding algorithm given in Section 5.1.2 to solve the $\sqrt{D}$-approximate Euclidean post-office problem.

We can improve this solution, by making the following observations. Consider the planar case. First, the point q above is a much better approximation if the angle between the vector pp^* and the positive x-axis is almost $\pi/4$. Next, the L_∞-metric depends on the coordinate system. That is, if we rotate the xy-system, then the L_∞-metric changes. The Euclidean metric, however, is invariant under such rotations.

We store the set S in a constant number of range trees, where each range tree stores the points according to its own coordinate system. Given a query point p, we use the range trees to compute L_∞-neighbors in all coordinate systems. By choosing the coordinate systems in an appropriate way, one of these L_∞-neighbors is a $(1 + \epsilon)$-approximate Euclidean neighbor of p. (In the planar case, we take care that there is always one coordinate system in which the angle between the vector pp^* and the positive x-axis is "almost" $\pi/4$.)

For the planar case, the coordinate systems are obtained as follows. Let $0 < \phi < \pi/4$ be such that $\tan \phi = \epsilon/(2 + \epsilon)$. For $0 \leq i < 2\pi/\phi$, let x_i (resp. y_i) be the directed line that makes an angle of $i \cdot \phi$ with the positive x-axis (resp. y-axis). Then, the i-th coordinate system is w.r.t. the axes x_i and y_i. For the higher-dimensional case, we use the cones as given in Yao [128].

Overall, we get a data structure for the $(1 + \epsilon)$-approximate Euclidean post-office problem having size $O(n \log^{D-1} n)$, and whose query and amortized update times are bounded by $O(\log^{D-1} n \log \log n)$.

Bespamyatnikh [27] gives an alternative solution using the same approximate distance functions as in Section 4.6. His technique applies to any L_τ-metric. The problem of finding a $(1 + \epsilon)$-approximate L_τ-neighbor is reduced to that of range searching for minimum. Hence, using range trees, we get a dynamic data structure of size $O(n \log^{D-1} n)$ with query and update times of $O(\log^D n)$. Using Chazelle's data structure [37] gives a data structure of size $O(n \log^{D-2} n)$ with query and update times of $O(\log^{D+1} \log \log n)$.

A completely different approach was given by Chan and Snoeyink [35]. Their result applies to any L_τ-metric, but it only gives efficient dynamic data structures in the planar case. So, assume that $D = 2$. We start with sketching an *exact* post-office data structure for the special case in which all query points lie on the y-axis. Consider the intersection of the Voronoi diagram of S with the y-axis. Using a lifting map and dualization (see [59]), this intersection corresponds to a planar lower convex hull. Using the algorithm of Overmars and van Leeuwen [98], we can store and maintain

43

post-office queries on the y-axis can be answered by binary search, in logarithmic time.

We use this solution to give an approximate post-office structure for the case in which all points of S are to the left of the y-axis, and all query points are to the right of the y-axis. Let $p = (p_1, p_2)$ be any query point such that $p_1 > 0$. For each i, $-\lceil \pi/(2\epsilon) - 1 \rceil \leq i \leq \lceil \pi/(2\epsilon) - 1 \rceil$, let p^i be the intersection of the y-axis with the ray from p that makes an angle of $i \cdot \epsilon$ with the negative x-axis. Also, let q^i be the exact nearest neighbor of p^i. One of these exact neighbors is a $(1 + \epsilon)$-approximate neighbor of the query point p: Let $p^* = (p_1^*, p_2^*)$ be the exact nearest neighbor of p, and assume w.l.o.g. that $p_2^* \geq p_2$. Let i be the index such that the ray with angle $i \cdot \epsilon$ is just below p^*. Then the neighbor q^i of p^i satisfies $d(p, q^i) \leq (1 + \epsilon) \, d(p, p^*)$. (For a proof, see [35].) It follows that we can solve this version of the problem with $O(\log n)$ query time, $O(\log^2 n)$ update time, and $O(n)$ space.

The general approximate post-office problem can now be solved by applying range tree techniques to the latter solution. The result is a data structure of size $O(n \log n)$, having an update time of $O(\log^3 n)$, that solves $(1 + \epsilon)$-approximate post-office queries in $O(\log^2 n)$ time. We remark that the data structure and, hence, the constants in the space and update time bounds, do *not* depend on ϵ. In fact, this data structure can be used for queries in which the input consists of a point p and an approximation bound ϵ.

5.2.4 An optimal solution for the approximate post-office problem

The $(1 + \epsilon)$-approximate post-office problem was solved optimally by Arya et al. [12]. They give a data structure of size $O(n)$, that answers queries in $O(\log n)$ time, and that can be built in $O(n \log n)$ time. This data structure and the constants in the space and building time bounds do *not* depend on ϵ. It can be used for answering queries for all degrees of precision. (Of course, the constant factor in the query time does depend on ϵ.)

The data structure is based on a hierarchical subdivision of space into cells that is similar to the ones we have seen already, e.g. in Section 2.5.2. Since we are dealing with a query problem, however, the previous techniques have to be adapted somewhat.

Define a *fat box* to be a D-dimensional axes-parallel rectangle for which the ratio of its longest side to its shortest side is bounded from above by some constant. We will construct a subdivision of $\mathbb{R}^D$ into pairwise disjoint *cells*, where a cell is either a fat box or the set theoretic difference of two fat boxes. These cells are called *box cells* and *doughnut cells*, respectively.

The construction is based on two operations on fat boxes. First, in a *split* operation, a fat box is split into two fat boxes by cutting its longest side into two equal parts. The split is called *fair* if both resulting boxes contain points of S.

Let B and B' be two fat boxes such that B' is contained in B. We say that B' is *sticky* for B if for each of the $2D$ sides of B', the distance from this side to the corresponding side of B is either zero or at least the width of B' along this dimension. The second operation is the *shrink* operation. It is only defined on a fat box B on which a fair split is not possible. This operation returns the smallest fat box $B' \neq B$ that is sticky for B and that contains all points of $B \cap S$.

Lemma 5.1 *Let B be any fat box that contains at least two points of S. Then a fair split operation or a shrink operation can be performed on B. If the shrink operation can be applied on B, then a fair split is always possible on the resulting shrunken fat box.*

Now we can describe the construction of the subdivision. We start with the smallest axes-parallel cube U containing all points of S. There are three possible cases. First assume that U contains only one point of S. Then this cube becomes a box cell of the subdivision, and the construction stops. Next, assume that U contains at least two points, but a fair split on U is not possible. Then, we perform the shrink operation on U. Let U' be the shrunken box. We create a doughnut cell $U \setminus U'$, and recursively apply the construction on U'. Finally, assume that U contains at least two points, and a fair split on U is possible. Then, we perform the fair split, yielding two fat boxes U_1 and U_2. We now recursively apply the construction on $U_1 \cap S$ and $U_2 \cap S$, respectively.

This construction defines a binary tree T in the natural way. Each node of this tree stores a cell. The cells that are stored at the leaves form the subdivision of $\mathbb{R}^D$. Lemma 5.1 implies that along any path in T there cannot be two consecutive "shrink nodes". Hence, the tree has size $O(n)$.

In order to use T for solving approximate post-office queries, we need to store some additional information. With each leaf of T, storing the cell C of the subdivision, we store a point $q(C)$ of S. If C is a box cell, then $q(C)$ is the unique point of $C \cap S$. Otherwise, if C is a doughnut cell of the form $U \setminus U'$, then $q(C)$ is an arbitrary point of $U' \cap S$.

Define the *size* of a box cell (resp. doughnut cell) of the subdivision to be the length of its longest side (resp. length of the longest side of its outer box).

Then, it is clear that for each cell C of the subdivision, and for each point x in $C \cap \mathbb{R}^D$, the distance between x and $q(C)$ is at most D times the size of C.

Before we give the query algorithm, we make some more remarks. First, we mention the following lemma, whose proof follows from packing arguments.

Lemma 5.2 *Let p be any point in $\mathbb{R}^D$ and let r and s be positive real numbers such that $r \geq s$. The number of cells of the subdivision having size at least s and intersecting the ball of radius r with center p is bounded by $O((r/s)^D)$.*

If we use a centroid decomposition to represent the tree T, see Section 3.3, then we can answer point location queries in $O(\log n)$ time. In such a query, we find the cell of the subdivision containing a given query point. Note that the size of the data structure remains linear. Using basically the same algorithm as in Section 2.5.2, the entire data structure can be built in $O(n \log n)$ time.

Let p be any point in $\mathbb{R}^D$ and let C be any cell of the subdivision. We define the distance between p and C to be the minimum distance between p and any point of $C \cap \mathbb{R}^D$. The following lemma will be important for the query algorithm.

Lemma 5.3 *Let $p \in \mathbb{R}^D$ be any query point, and let k be a positive integer. In $O(k \log n)$ time, we can enumerate the k cells of the subdivision that are closest to p,*

moment when the enumeration starts.

Proof: We only give the algorithm for enumerating the cells. For the correctness proof and time analysis, we refer to [12]. The algorithm is based on the *priority search* technique, that was used for the first time in [10].

We say that two cells of the subdivision are neighbors if they share a common $(D-1)$-dimensional boundary. Let C be a cell of the subdivision, and let F be any of its $(D-1)$-dimensional facets. Let p' be the point on F that is closest to p. Clearly, p' can be computed in constant time. Let $N(C, F, p)$ be the cell of the subdivision that is closest to p and that is a neighbor of C along F. Given p', we can find this cell by point location, in logarithmic time.

Now we can give the algorithm. We will maintain a heap that stores cells of the subdivision, sorted by their distances from p. Initially, the heap is empty. Also, we assume that all cells are unmarked. Using point location, we find the cell that contains p, mark it, and insert it into the heap. Now, we make a sequence of k iterations.

During one iteration, we remove the cell C that is closest to p from the heap, and report it. Then, for each face F of C, compute the cell $N(C, F, p)$ and, in case it is unmarked, mark it and insert it into the heap. ∎

Now we can give the algorithm for answering approximate post-office queries. Let p be any query point. Using the algorithm of Lemma 5.3, enumerate cells of the subdivision in increasing order of their distance to p. For each cell C that is enumerated, compute the distance between p and its associated point $q(C)$, and maintain the smallest distance δ encountered so far. We stop with enumerating cells as soon as the distance between p and the current cell is larger than $\delta/(1+\epsilon)$. Then we report δ and the point whose distance to p is equal to δ.

To prove the correctness of this algorithm, let C be the cell of the subdivision that contains the exact nearest neighbor p^* of p. Note that C must be a box cell. If C has been enumerated, then $p^* = q(C)$, and we have in fact found the exact neighbor of p. Assume that C has not been enumerated. Since we visit the cells in increasing order of their distance to p, we know that the distance between C and p is at least $\delta/(1+\epsilon)$. Therefore, $\delta \leq (1+\epsilon)\, d(p, p^*)$, i.e., we have found a $(1+\epsilon)$-approximate neighbor of p.

It remains to analyze the query time. For each cell visited, we need $O(\log n)$ time. Hence, we have to show that at most a constant number of cells are enumerated before the algorithm terminates.

If C is a cell of the subdivision, then we denote its size by $size(C)$. Moreover, $d(p, C)$ denotes the distance between p and C.

Consider the query algorithm. Let C be the current cell that is being enumerated, let $\delta' = d(p, C)$ and $s = size(C)$. We claim that if $s < \delta'\epsilon/D$, then the next cell that is enumerated causes the query algorithm to terminate. To prove this, let x be a point of $C \cap \mathbb{R}^D$ such that $d(p, x) = \delta'$. We know that $d(x, q(C)) \leq Ds$. Hence, by the triangle inequality, $d(p, q(C)) \leq \delta' + Ds < (1+\epsilon)\,\delta'$. In particular, the current value of δ satisfies $\delta < (1+\epsilon)\,\delta'$. The next cell that is enumerated has distance at least $\delta' > \delta/(1+\epsilon)$ to p. This proves the claim.

cells C for which $size(C) \geq \epsilon\, d(p, C)/D$.

We define x to be the minimum value of $D\, size(C) + d(p, C)$, minimized over all cells C of the subdivision.

Lemma 5.4 *Consider the value of δ that is reported by the algorithm. We have $\delta < (1 + \epsilon)\, x$.*

Proof: Let C' be the cell for which $D\, size(C') + d(p, C') = x$. If C' is enumerated during the algorithm, then

$$\delta \leq d(p, q(C')) \leq D\, size(C') + d(p, C') = x < (1 + \epsilon)\, x.$$

Otherwise, if C' is not enumerated, then we know that $d(p, C') > \delta/(1 + \epsilon)$, which implies that

$$\delta < (1 + \epsilon)\, d(p, C') \leq (1 + \epsilon)\, d(p, q(C')) \leq (1 + \epsilon)\, x.$$

This completes the proof. ∎

Lemma 5.5 *All cells that are enumerated, except possibly the last one, intersect the ball with radius $(1 + \epsilon)\, x$ centered at p.*

Proof: Consider again the final value of δ. Let C be a cell of the subdivision such that $d(p, C) > (1 + \epsilon)\, x$. Also, let C_0 be the cell whose enumeration resulted in δ. That is, $\delta = d(p, q(C_0))$. Then, since $\delta < (1 + \epsilon)\, x$, we have $d(p, C) > \delta \geq d(p, C_0)$. It follows that if C is enumerated, then this happens after C_0 has been enumerated. At the moment when C is enumerated, however, we have $d(p, C) > \delta \geq \delta/(1 + \epsilon)$. Hence, the algorithm terminates at this moment. ∎

Let C be a cell for which $size(C) \geq \epsilon\, d(p, C)/D$. The definition of x immediately implies that $D\, size(C) + d(p, C) \geq x$. Therefore, $D\, size(C) + D\, size(C)/\epsilon \geq x$, or, equivalently, $size(C) \geq x/(D(1 + 1/\epsilon))$.

This proves that the number of cells that are enumerated is at most equal to three plus the number of cells of size at least $x/(D(1 + 1/\epsilon))$ that intersect the ball centered at p and having radius $(1 + \epsilon)\, x$. By Lemma 5.2, there are at most a constant number of such cells C.

This completes the proof of the claim that the running time of the query algorithm is logarithmic.

To conclude this section, we remark that Arya et al. extend their algorithm such that the approximate k nearest neighbors to a query point can be enumerated in $O(k \log n)$ time. (In this case, the points $q(C)$ have to be chosen more carefully.) They also present results of experiments made using a simplification of the algorithm as presented above, for 16-dimensional point sets. For details, we refer to [12].

6.1 Approximating the complete Euclidean graph: spanners

In this section, we give an overview of the algorithms that have been designed for constructing spanners. (See Definition 1.1.) Throughout, distances denote Euclidean distances.

We first remark that the problem of constructing any t-spanner, where $1 < t < 2$, has an $\Omega(n \log n)$ lower bound in the algebraic computation tree model. This follows from the simple fact that in such a spanner the closest pair must be connected by an edge. The same lower bound was in fact proved for any constant $t > 1$ by Chen, Das and Smid [49].

Spanners were introduced to computational geometry by Chew [39]. Let S be a set of n points in the plane. Chew shows that the L_1-Delaunay triangulation of S is a Euclidean $\sqrt{10}$-spanner. That is, for every pair p, q of points in S, there is a path in this triangulation between p and q whose weight is at most $\sqrt{10}$ times the Euclidean distance between p and q. Later, Dobkin, Friedman and Supowit [57] proved that the Euclidean Delaunay triangulation is a t-spanner for $t = \pi(1 + \sqrt{5})/2$. Keil and Gutwin [83] improved this to $t = \frac{2\pi}{3\cos \pi/6}$. Levcopoulos and Lingas [88] generalized the latter result in the following way. Given the Euclidean Delaunay triangulation of n planar points, and any positive real number r, a plane graph can be constructed in linear time, that is a t-spanner for $t = (1 + 1/r)\frac{2\pi}{3\cos \pi/6}$, and whose total weight is at most $2r + 1$ times the weight of a minimum spanning tree for these points.

6.1.1 Spanners based on simplicial cones

Using simplicial cones, we can give a simple construction of a t-spanner for any constant $t > 1$. The construction appeared for the first time in Clarkson [41] for the cases $D = 2$ and $D = 3$. It was discovered independently by Keil and Gutwin [83] for the case $D = 2$. Ruppert and Seidel [102] generalized the construction to an arbitrary dimension.

We describe the planar version of the construction. As in Section 4.6, we choose an integer constant k. Let $\theta = 2\pi/k$. Rotate the positive x-axis over angles $i \cdot \theta$, $0 \le i < k$, yielding k rays. Let $C_1, C_2, \ldots, C_k$ be the cones that are bounded by any two successive rays. Also, for $1 \le i \le k$, let l_i be a fixed ray that emanates from the origin and that is contained in C_i.

Define a graph in the following way. For each point p of S and each i, $1 \le i \le k$, translate the cone C_i and the corresponding ray l_i such that its apex and starting point are at p, respectively. Then take the point $q \in S$ that is contained in the translated cone and whose projection onto the translated ray is closest to p. We add the directed edge (p, q) to the graph.

The graph obtained in this way is called the Θ-*graph*. Clearly, it contains at most $kn = O(n)$ edges. To prove that this graph is a spanner, consider any two points p and q of S. We construct a path from p to q, in the following way. Let $p_0 := p$. Let $i \ge 0$, and assume we have already constructed a path $p_0, p_1, \ldots, p_i$. If $p_i = q$, then the construction stops. Otherwise, let j be the index such that the cone C_j—when

for some point r in the translated cone. We extend the path by setting $p_{i+1} := r$.

It can be shown that

$$d(p_{i+1}, q) \le d(p_i, q) - (\cos \theta - \sin \theta) \, d(p_i, p_{i+1}). \qquad (3)$$

As a result, if $k > 8$, each next point on the path is closer to q than the previous one. In particular, point q will be reached. Rewriting inequality (3) shows that the entire path from p to q has weight at most t times the distance between p and q, for $t = 1/(\cos \theta - \sin \theta)$. Note that by choosing k large enough, the value of t gets arbitrarily close to one.

The construction can easily be generalized to any dimension D, by using the simplicial cones of Yao [128]. Using the sweep technique and range trees, the Θ-graph can be constructed in $O(n \log^{D-1} n)$ time. For details, see [13, 14, 41, 83, 102].

Let the *spanner diameter* of a spanner be defined as the smallest integer ℓ such that for any pair p and q of points, there is a path from p to q of weight at most t times the distance between p and q, and that contains at most ℓ edges. The Θ-graph can have spanner diameter $n - 1$: Take n points on the real line. For these points, the Θ-graph is just a list storing the points in sorted order.

Arya, Mount and Smid [13, 14] show how to combine the Θ-graph with skip lists [100] to get a t-spanner having $O(n)$ edges and $O(\log n)$ spanner diameter, both with high probability. They also show how to maintain this spanner efficiently under insertions and deletions of points, in the model of *random updates*, as defined in Mulmuley [95].

Although the out-degree of any vertex in the Θ-graph is bounded by a constant, the maximum in-degree can be as large as $n - 1$: Take $n - 1$ points on a circle, and let the n-th point be its center. In the corresponding Θ-graph, each point on the circle has an edge towards the center vertex. Arya et al. [9] give an $O(n \log n)$-time algorithm that transforms any spanner of bounded out-degree into a spanner that has both bounded out-degree and bounded in-degree. Applying this transformation to the Θ-graph gives an $O(n \log^{D-1} n)$-time algorithm for constructing a t-spanner of bounded in- and out-degree.

6.1.2 Greedy spanners

There is a simple *greedy algorithm* for constructing spanners. Let S be a set of n points in $\mathbb{R}^D$, and let $t > 1$. Sort the $\binom{n}{2}$ point pairs in increasing order of their distances. Start with an empty graph on S, and consider all point pairs in sorted order. If p, q is the current pair, then add an edge between them if and only if the current graph does not contain a path between p and q of weight at most $t \cdot d(p, q)$.

It is clear that this algorithm constructs a t-spanner. Other properties of this spanner were analyzed in a sequence of papers. First, Chandra et al. [36] showed that each vertex in this spanner has a degree that is bounded by a constant. Recent results of Das, Heffernan and Narasimhan [46] and Das, Narasimhan and Salowe [48] prove that its weight is proportional to that of a minimum spanning tree for S.

Since the greedy algorithm looks at each pair of points explicitly, its running time is $\Omega(n^2)$. Das and Narasimhan [47] present a variant of the greedy algorithm based

of [46, 48] shows that this algorithm produces a t-spanner whose weight is proportional to that of a minimum spanning tree. Its degree, however, can be very large.

As mentioned by Vaidya [126], Feder and Nisan gave an alternative greedy algorithm for constructing a spanner of bounded degree. (See also Salowe [106].) Let $0 < \theta < \pi/4$ be a real number. Again, sort the $\binom{n}{2}$ pairs in increasing order of their distances, start with an empty graph on S, and consider the pairs in sorted order. If p, q is the current pair, then we add an edge between them if and only if there is no $r \in S$ such that (i) (p, r) is an edge and the angle rpq is less than θ or (ii) (r, q) is an edge and the angle rqp is less than θ. The resulting graph is a t-spanner for $t = 1/(\cos\theta - \sin\theta)$, and it clearly has bounded degree. The weight of this spanner, however, can be very large.

Arya and Smid [15] gave an efficient implementation of a variant of Feder and Nisan's greedy algorithm that uses the data structure of Section 4.3. The result is an $O(n \log^D n)$-time algorithm that constructs a t-spanner of bounded degree whose weight is bounded by $O(\log n)$ times the weight of a minimum spanning tree. If we apply the results of [47] to this spanner, then we get an $O(n \log^D n)$-time algorithm for constructing a t-spanner of bounded degree with weight within a constant factor of that of a minimum spanning tree.

6.1.3   Spanners based on well-separated pairs

The first optimal $O(n \log n)$-time algorithms for constructing a t-spanner on any set of n points in $\mathbb{R}^D$, for any constant $t > 1$, were given by Vaidya [126] and Salowe [104]. Their algorithms are related and use hierarchical subdivisions. It should not be a surprise that we can also use the well-separated pair decomposition (WSPD) of Section 2.5.2 for giving a similar construction. (This is due to Callahan and Kosaraju [32].)

To prove this, let $\{\{A_1, B_1\}, \{A_2, B_2\}, \ldots, \{A_m, B_m\}\}$ be a WSPD for S with $m = O(n)$. For each i, $1 \leq i \leq m$, choose arbitrary *representative* points $a_i \in A_i$ and $b_i \in B_i$, and add an edge between these two points.

This defines the graph. It is clear that it can be built in $O(n \log n)$ time. To prove that it is a spanner, consider any two points p and q of S. We know that there is an index i such that $p \in A_i$ and $q \in B_i$. Consider the corresponding representatives $a_i \in A_i$ and $b_i \in B_i$. We recursively construct paths between p and a_i, and between b_i and q. The final path between p and q consists of the path between p and a_i, followed by the edge (a_i, b_i), followed by the path between b_i and q.

The definition of WSPD implies that the edge (a_i, b_i) is long compared to the distances $d(p, a_i)$ and $d(q, b_i)$. Using this observation, it can easily be shown that by choosing the separation constant large enough, the graph is a t-spanner.

It has been shown that by modifying the WSPD-spanner, other spanners can be constructed that have a variety of properties.

First, if the representatives are chosen carefully, then the edges of the graph can be directed such that the spanner has bounded out-degree. (See [32].) Hence, applying the transformation of Arya et al. [9] that was mentioned in Section 6.1.1 shows that for each $t > 1$, we can build in $O(n \log n)$ time a t-spanner of bounded degree.

another way, the spanner based on the WSPD has $O(\log n)$ spanner diameter. Arya et al. [9] show that the weight of this spanner is bounded by $O(\log n)$ times the weight of a minimum spanning tree. We note, however, that this spanner does not have bounded degree.

Arya et al. [9] use simplicial cones to prune the WSPD-spanner. They analyze the pruned graph using a theorem of Das, Narasimhan and Salowe [48]. The result is an algorithm that constructs a t-spanner of bounded degree whose weight is proportional to the weight of a minimum spanning tree. This is the first algorithm that constructs such a spanner in $O(n \log n)$ time.

One of the main results in [9] is a technique that decomposes the WSPD into a constant number of hierarchically organized sets of well-separated pairs. This decomposition is used to give a class of spanners that can be viewed as the union of a constant number of trees. Moreover, each of the $\binom{n}{2}$ spanner paths arises as the unique path between two leaves in one of these trees. In Alon and Schieber [6] and Bodlaender, Tel and Santoro [29], algorithms are given for adding edges to a tree such that its diameter is reduced. Applying this gives a t-spanner of spanner diameter 2 with $O(n \log n)$ edges, a t-spanner of spanner diameter 3 with $O(n \log \log n)$ edges, a t-spanner of spanner diameter 4 with $O(n \log^* n)$ edges, and so on. In fact, it is even possible to obtain a t-spanner of spanner diameter $\alpha(n)+2$ with only $O(n)$ edges, where $\alpha(n)$ is the inverse of Ackermann's function. All these spanners can be constructed in $O(n \log n)$ time. Moreover, the trade-offs between the spanner diameter and the number of edges are optimal. This is true even for spanners for one-dimensional point sets. (The optimality follows from the results in [6, 29]. These papers do not consider spanners explicitly, but their results can easily be "translated" to spanners.)

We finally mention the other results in [9]. The decomposition mentioned in the preceding paragraph can be combined with topology trees [65] in order to get a t-spanner of bounded degree and $O(\log n)$ spanner diameter. This spanner can be constructed in $O(n \log n)$ time. Moreover, it is shown that the weight of this spanner is bounded by $O(\log^2 n)$ times the weight of a minimum spanning tree.

6.1.4 Spanners with small degree

We have seen that for each $t > 1$, a t-spanner of bounded degree can be constructed. The upper bound on the degree, however, depends on t and the dimension D. Typically, it is of the form $(c/(t-1))^D$, for some constant c. Dobkin, Friedman and Supowit [57] posed the problem of determining the smallest integer v^* such that for every set of n points in $\mathbb{R}^D$, a t-spanner of degree v^* can be constructed, for some constant t. (Of course, t will depend on D.) They prove that $3 \leq v^* \leq 7$.

Salowe [106] proves that $v^* \leq 4$. He gives an algorithm that transforms any t-spanner of degree v into a t'-spanner of degree $\lfloor v/2 \rfloor + 2$, where $t' \leq 117 t + 32 \cdot 3^{D+1}$. Starting with any bounded degree spanner, and applying this transformation repeatedly, proves Salowe's result.

The problem of determining v^* was solved by Das and Heffernan [45]. They prove that $v^* = 3$. To be more precise, they give a polynomial time algorithm that, given any set of n points in $\mathbb{R}^D$ and any $\delta > 1$, constructs a t-spanner, for some constant t,

6.2 More results on closest point problems

We have given algorithms and data structures for closest point problems whose complexity does not depend on the geometry of the points. That is, all complexity bounds only depend on the number n of points, and on constants like the dimension D. Several authors have developed algorithms whose running time also depends on measures such as the ratio of the maximum distance to the minimum distance. We mention the algorithm of Clarkson [40] for solving the all-nearest-neighbors problem, and the data structures of Clarkson [43] and Chen and Snoeyink [35] for solving the approximate post-office problem.

There has been much work on designing simple algorithms that are especially efficient if the input points come from some probability distribution. For example, Friedman, Bentley and Finkel [67] and Bentley, Weide and Yao [24] give algorithms for the post-office problem and the closest pair problem. Bartling and Hinrichs [18] analyze the expected running time of a simple algorithm for the all-nearest-neighbors problem for point sets that are uniformly distributed in the unit-cube. Finally, Golin [69] gives a simple data structure that maintains the closest pair in logarithmic expected time per update, when the points are drawn from a wide class of distributions.

We considered the problem of computing the minimum distance, or the k smallest distances, in a point set, and gave optimal algorithms for these problems. The problem of selecting only the k-th smallest distance turns out to be much more difficult. Salowe [103] shows how to select the k-th smallest L_∞-distance in a set of n points in $\mathbb{R}^D$, in $O(n \log^D n)$ time. Agarwal et al. [1] give an efficient randomized algorithm for selecting the k-th Euclidean distance in a planar set of points. Both algorithms use the parametric search technique.

The problem of selecting the largest distance, i.e., the *diameter*, in a set of n points in $\mathbb{R}^D$, can be solved in $O(n \log n)$ time if $D = 2$. See [111, 99]. If $D = 3$, the best result known is due to Amato, Goodrich and Ramos [7]. They show how the diameter can be computed in $O(n \log^3 n)$ time.

In the *bichromatic closest pair problem*, we are given sets R and B of red and blue points in $\mathbb{R}^D$, respectively, and we want to find a pair (r, b), where $r \in R$ and $b \in B$, whose distance is minimal. Let n denote the total number of red and blue points. Using Voronoi diagrams, the planar version of this problem can easily be solved in $O(n \log n)$ time. For $D > 2$, the problem is much more difficult. Agarwal et al. [2] show that there is a relationship between the bichromatic closest pair problem and the minimum spanning tree problem. They use it to give an efficient algorithm to compute the bichromatic closest pair. Callahan and Kosaraju [32] used their well-separated pair decomposition to improve the running time. Katoh and Iwano [82] give a general technique that can be used to compute the k closest bichromatic pairs, or the k furthest bichromatic pairs.

Agarwal et al. [3] introduced a variation of the bichromatic closest pair problem. Here, we are given a collection of point sets, and for each point p we have to find a nearest neighbor that does not belong to p's set. If n denotes the total number of

52

Hinrichs [73].

We finally mention that research has been on proximity problems on sets of objects other than points. We mention Sharir [113], who solves the closest pair problem for a set of planar disks, and Graf and Hinrichs [72], who give a plane sweep algorithm for solving the all-nearest-neighbors problem on a set of convex planar objects.

7 Open problems

In the preceding sections, we have seen optimal solutions to several closest point problems. In particular, the complexity of the static and dynamic versions of the closest pair problem have been completely settled. There remain, however, many challenging open problems in this field. We mention some of these.

The static post-office problem has been solved optimally in the planar case. Does there exist a dynamic data structure for the planar post-office problem having poly-logarithmic query and update times and using $O(n \log^{O(1)} n)$ space?

We mentioned already that in dimension $D > 2$, the complexity of the static post-office problem has not been determined. It has been conjectured that a data structure of size $O(n \log^{O(1)} n)$ with polylogarithmic query time does not exist. An important open problem is to prove or disprove this conjecture.

We have seen that the all-nearest-neighbors problem can be solved optimally, in $O(n \log n)$ time. No non-trivial results are known, however, for maintaining the all-nearest-neighbors under insertions and deletions, even in the planar case. Note that when a point is inserted or deleted, at most a constant number of nearest neighbor pairs are changed. The problem is, of course, finding these pairs.

We mentioned in Section 6.2 that the problem of selecting the k-th smallest L_∞-distance in a set of n points in $\mathbb{R}^D$ can be solved in $O(n \log^D n)$ time. Can this problem be solved in $O(n \log n)$ time? Also, what is the complexity of selecting the k-th Euclidean distance, even in a planar set of points?

In Section 6.1, several optimal algorithms were mentioned for constructing spanners that have one or more properties. The most general result is an $O(n \log n)$-time algorithm for constructing a t-spanner of bounded degree, $O(\log n)$ spanner diameter, and weight bounded by $O(\log^2 n)$ times the weight of a minimum spanning tree. Is there an $O(n \log n)$-time algorithm that constructs a spanner of bounded degree, $O(\log n)$ spanner diameter, and weight bounded by $O(\log n)$ times the weight of a minimum spanning tree? Such an algorithm would be optimal: We saw already that $\Omega(n \log n)$ is a lower bound on the time to construct any spanner. Also, in [9] it is shown that any spanner of $O(\log n)$ spanner diameter must have weight $\Omega(\log n)$ times the weight of a minimum spanning tree.

Finally, it is clear that all $\binom{n}{2}$ distances in a set of n points in $\mathbb{R}^D$ can be sorted in $O(n^2 \log n)$ time. Can this be improved to $O(n^2)$?

References

[1] P.K. Agarwal, B. Aronov, M. Sharir and S. Suri. *Selecting distances in the plane.* Algorithmica **9** (1993), pp. 495–514.

[2] P.K. Agarwal, H. Edelsbrunner, O. Schwarzkopf and E. Welzl. *Euclidean minimum spanning trees and bichromatic closest pairs.* Discrete & Computational Geometry **6** (1991), pp. 407–422.

[3] A. Aggarwal, H. Edelsbrunner, P. Raghavan and P. Tiwari. *Optimal time bounds for some proximity problems in the plane.* Information Processing Letters **42** (1992), pp. 55–60.

[4] 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 & Computational Geometry **4** (1989), pp. 591-604.

[5] A. Aggarwal, H. Imai, N. Katoh and S. Suri. *Finding k points with minimum diameter and related problems.* Journal of Algorithms **12** (1991), pp. 38–56.

[6] N. Alon and B. Schieber. *Optimal preprocessing for answering on-line product queries.* Report 71/87, Tel-Aviv University, 1987.

[7] N.M. Amato, M.T. Goodrich and E.A. Ramos. *Parallel algorithms for higher-dimensional convex hulls.* Proceedings 35th Annual IEEE Symposium on Foundations of Computer Science, 1994, pp. 683–694.

[8] H.C. Andrews. *Introduction to Mathematical Techniques in Pattern Recognition.* Wiley-Interscience, New York, 1972.

[9] S. Arya, G. Das, D.M. Mount, J.S. Salowe and M. Smid. *Euclidean spanners: short, thin, and lanky.* Proceedings 27th Annual ACM Symposium on the Theory of Computing, 1995, pp. 489–498.

[10] S. Arya and D.M. Mount. *Algorithms for fast vector quantization.* Proceedings of the Data Compression Conference (DCC '93), IEEE Press, 1993, pp. 381–390.

[11] S. Arya and D.M. Mount. *Approximate nearest neighbor queries in fixed dimensions.* Proceedings 4th Annual ACM-SIAM Symposium on Discrete Algorithms, 1993, pp. 271–280.

gorithm for approximate nearest neighbor searching. Proceedings 5th Annual ACM-SIAM Symposium on Discrete Algorithms, 1994, pp. 573–582.

[13] S. Arya, D.M. Mount and M. Smid. *Randomized and deterministic algorithms for geometric spanners of small diameter.* Proceedings 35th Annual IEEE Symposium on Foundations of Computer Science, 1994, pp. 703–712.

[14] S. Arya, D.M. Mount and M. Smid. *Dynamic algorithms for geometric spanners of small diameter: randomized solutions.* Report MPI-I-94-156, Max-Planck-Institut für Informatik, Saarbrücken, 1994.

[15] S. Arya and M. Smid. *Efficient construction of a bounded-degree spanner with low weight.* Algorithmica **17** (1997), pp. 33–54.

[16] F. Aurenhammer. *Voronoi diagrams: a survey of a fundamental geometric data structure.* ACM Computing Surveys **23** (1991), pp. 345–405.

[17] F. Bartling and K. Hinrichs. *A plane-sweep algorithm for finding a closest pair among convex planar objects.* Proceedings 9th Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science, Vol. 577, Springer-Verlag, Berlin, 1992, pp. 221–232.

[18] F. Bartling and K. Hinrichs. *Probabilistic analysis of an algorithm for solving the k-dimensional all-nearest-neighbors problem by projection.* BIT **31** (1991), pp. 558–565.

[19] M. Ben-Or. *Lower bounds for algebraic computation trees.* Proceedings 15th Annual ACM Symposium on the Theory of Computing, 1983, pp. 80–86.

[20] J.L. Bentley. *Divide and conquer algorithms for closest point problems in multidimensional space.* Ph.D. Thesis, University of North Carolina, Chapel Hill, N.C., 1976.

[21] J.L. Bentley. *Decomposable searching problems.* Information Processing Letters **8** (1979), pp. 244–251.

[22] J.L. Bentley. *Multidimensional divide-and-conquer.* Communications of the ACM **23** (1980), pp. 214–229.

[23] J.L. Bentley and M.I. Shamos. *Divide-and-conquer in multidimensional space.* Proceedings 8th Annual ACM Symposium on the Theory of Computing, 1976, pp. 220–230.

[24] J.L. Bentley, B.W. Weide and A.C. Yao. *Optimal expected time algorithms for closest point problems.* ACM Transactions on Mathematical Software **6** (1980), pp. 563–580.

[25] M. Bern. *Approximate closest-point queries in high dimensions.* Information Processing Letters **45** (1993), pp. 95–99.

Computer and Systems Sciences **48** (1994), pp. 384–409.

[27] S.N. Bespamyatnikh. *The region approach for some dynamic closest-point problems.* Proceedings 6th Canadian Conference on Computational Geometry, 1994, pp. 75–80.

[28] S.N. Bespamyatnikh. *An optimal algorithm for closest pair maintenance.* Proceedings 11th Annual ACM Symposium on Computational Geometry, 1995, pp. 152–161.

[29] H.L. Bodlaender, G. Tel and N. Santoro. *Trade-offs in non-reversing diameter.* Nordic Journal of Computing **1** (1994), pp. 111–134.

[30] P.B. Callahan. *Dealing with higher dimensions: the well-separated pair decomposition and its applications.* Ph.D. Thesis, Johns Hopkins University, Baltimore, Maryland, 1995.

[31] P.B. Callahan and S.R. Kosaraju. *A decomposition of multi-dimensional point-sets with applications to k-nearest-neighbors and n-body potential fields.* Proceedings 24th Annual ACM Symposium on the Theory of Computing, 1992, pp. 546–556.

[32] P.B. Callahan and S.R. Kosaraju. *Faster algorithms for some geometric graph problems in higher dimensions.* Proceedings 4th Annual Symposium on Discrete Algorithms, 1993, pp. 291–300.

[33] P.B. Callahan and S.R. Kosaraju. *Algorithms for dynamic closest pair and n-body potential fields.* Proceedings 6th Annual Symposium on Discrete Algorithms, 1995, pp. 263–272.

[34] P.B. Callahan and S.R. Kosaraju. *A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields.* Journal of the ACM **42** (1995), pp. 67–90.

[35] T.M. Chan and J. Snoeyink. *Algorithms for approximate nearest-neighbor queries.* Manuscript, 1995.

[36] B. Chandra, G. Das, G. Narasimhan and J. Soares. *New sparseness results on graph spanners.* International Journal of Computational Geometry and Applications **5** (1995), pp. 125–144.

[37] B. Chazelle. *A functional approach to data structures and its use in multidimensional searching.* SIAM Journal on Computing **17** (1988), pp. 427–462.

[38] B. Chazelle and L.J. Guibas. *Fractional cascading I: A data structuring technique.* Algorithmica **1** (1986), pp. 133–162.

[39] P. Chew. *There is a planar graph almost as good as the complete graph.* Proceedings 2nd Annual ACM Symposum on Computational Geometry, 1986, pp. 169–177.

24th Annual Symposium on Foundations of Computer Science, 1983, pp. 226–232.

[41] K.L. Clarkson. *Approximation algorithms for shortest path motion planning.* Proceedings 19th Annual ACM Symposium on the Theory of Computing, 1987, pp. 56–65.

[42] K.L. Clarkson. *A randomized algorithm for closest-point queries.* SIAM Journal on Computing **17** (1988), pp. 830–847.

[43] K.L. Clarkson. *An algorithm for approximate closest-point queries.* Proceedings 10th Annual ACM Symposum on Computational Geometry, 1994, pp. 160–164.

[44] R.F. Cohen and R. Tamassia. *Combine and conquer: a general technique for dynamic algorithms.* Proceedings 1st Annual European Symposium on Algorithms, Lecture Notes in Computer Science, Vol. 726, Springer-Verlag, Berlin, 1993, pp. 97–108.

[45] G. Das and P.J. Heffernan. *Constructing degree-3 spanners with other sparseness properties.* Proceedings 4th International Symposium on Algorithms and Computations, Lecture Notes in Computer Science, Vol. 762, Springer-Verlag, Berlin, 1993, pp. 11–20.

[46] G. Das, P. Heffernan and G. Narasimhan. *Optimally sparse spanners in 3-dimensional Euclidean space.* Proceedings 9th Annual ACM Symposum on Computational Geometry, 1993, pp. 53–62.

[47] G. Das and G. Narasimhan. *A fast algorithm for constructing sparse Euclidean spanners.* Proceedings 10th Annual ACM Symposum on Computational Geometry, 1994, pp. 132–139.

[48] G. Das, G. Narasimhan and J.S. Salowe. *A new way to weigh malnourished Euclidean graphs.* Proceedings 6th Annual ACM-SIAM Symposium on Discrete Algorithms, 1995, pp. 215–222.

[49] D. Z. Chen, G. Das and M. Smid. *Lower bounds for computing geometric spanners and approximate shortest paths.* Proceedings 8th Canadian Conference on Computational Geometry, 1996, pp. 155–160.

[50] A. Datta, H.-P. Lenhof, C. Schwarz and M. Smid. *Static and dynamic algorithms for k-point clustering problems.* Journal of Algorithms **19** (1995), pp. 474–503.

[51] W.H.E. Day and H. Edelsbrunner. *Efficient algorithms for agglomerative hierarchical clustering methods.* Journal of Classification **1** (1984), pp. 7–24.

[52] M.T. Dickerson, R.L. Scot Drysdale and J.R. Sack. *Simple algorithms for enumerating interpoint distances and finding k nearest neighbors.* International Journal of Computational Geometry & Applications **2** (1993), pp. 221–239.

dimensions. Computational Geometry, Theory and Applications **5** (1996), pp. 277-291.

[54] M. Dietzfelbinger, T. Hagerup, J. Katajainen and M. Penttonen. *A reliable randomized algorithm for the closest-pair problem.* Report 93/25, Department of Computer Science, University of Copenhagen, 1993.

[55] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert and R.E. Tarjan. *Dynamic perfect hashing: upper and lower bounds.* SIAM Journal on Computing **23** (1994), pp. 738–761.

[56] D.P. Dobkin, R.L. Drysdale, III, and L.J. Guibas. *Finding smallest polygons.* In: Computational Geometry, Advances in Computing Research, Vol. 1, JAI Press, London, 1983, pp. 181–214.

[57] D. Dobkin, S.J. Friedman and K.J. Supowit. *Delaunay graphs are almost as good as complete graphs.* Discrete & Computational Geometry **5** (1990), pp. 399–407.

[58] D. Dobkin and S. Suri. *Maintenance of geometric extrema.* Journal of the ACM **38** (1991), pp. 275–298.

[59] H. Edelsbrunner. *Algorithms in Combinatorial Geometry.* Springer-Verlag, Berlin, 1987.

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

[61] A. Efrat, M. Sharir and A. Ziv. *Computing the smallest k-enclosing circle and related problems.* Computational Geometry, Theory and Applications **4** (1994), pp. 119–136.

[62] D. Eppstein and J. Erickson. *Iterated nearest neighbors and finding minimal polytopes.* Discrete & Computational Geometry **11** (1994), pp. 321–350.

[63] R.A. Finkel and J.L. Bentley. *Quad trees — a data structure for retrieval on composite keys.* Acta Informatica **4** (1974), pp. 1–9.

[64] S. Fortune and J. Hopcroft. *A note on Rabin's nearest-neighbor algorithm.* Information Processing Letters **8** (1979), pp. 20–23.

[65] G.N. Frederickson. *A data structure for dynamically maintaining rooted trees.* Proceedings 4th Annual ACM-SIAM Symposium on Discrete Algorithms, 1993, pp. 175–184.

[66] M. Fredman, F. Komlós and E. Szemerédi. *Storing a sparse table with $O(1)$ worst case access time.* Journal of the ACM **31** (1984), pp. 538–544.

[67] J.H. Friedman, J.L. Bentley and R.A. Finkel. *An algorithm for finding best matches in logarithmic expected time.* ACM Transactions on Mathematical Software **3** (1977), pp. 209–226.

geometry problems. Proceedings 16th Annual ACM Symposium on the Theory of Computing, 1984, pp. 135–143.

[69] M.J. Golin. *Dynamic closest pairs—a probabilistic approach.* Proceedings 3rd Scandinavian Workshop on Algorithm Theory, Lecture Notes in Computer Science, Vol. 621, Springer-Verlag, Berlin, 1992, pp. 340–351.

[70] M. Golin, R. Raman, C. Schwarz and M. Smid. *Randomized data structures for the dynamic closest-pair problem.* Proceedings 4th Annual ACM-SIAM Symposium on Discrete Algorithms, 1993, pp. 301–310.

[71] M. Golin, R. Raman, C. Schwarz and M. Smid. *Simple randomized algorithms for closest pair problems.* Nordic Journal of Computing **2** (1995), pp. 3–27.

[72] T. Graf and K. Hinrichs. *A plane-sweep algorithm for the all-nearest-neighbors problem for a set of convex planar objects.* Proceedings 3rd Workshop on Algorithms and Data Structures, Lecture Notes in Computer Science, Vol. 709, Springer-Verlag, Berlin, 1993, pp. 349–360.

[73] T. Graf and K. Hinrichs. *Algorithms for proximity problems on colored point sets.* Proceedings 5th Canadian Conference on Computational Geometry, 1993, pp. 420–425.

[74] L. Guibas, J. Hershberger, D. Leven, M. Sharir and R.E. Tarjan. *Linear time algorithms for visibility and shortest path problems inside triangulated simple polygons.* Algorithmica **2** (1987), pp. 209–233.

[75] J.A. Hartigan. *Clustering Algorithms.* Wiley, New York, 1975.

[76] W. Heiden, M. Schlenkrich and J. Brickmann. *Triangulation algorithms for the representation of molecular surface properties.* J. Comp. Aided Mol. Des. **4** (1990), pp. 255–269.

[77] K. Hinrichs, J. Nievergelt and P. Schorn. *Plane-sweep solves the closest pair problem elegantly.* Information Processing Letters **26** (1987/88), pp. 255–261.

[78] K. Hinrichs, J. Nievergelt and P. Schorn. *An all-round sweep algorithm for 2-dimensional nearest-neighbors problems.* Acta Informatica **29** (1992), pp. 383–394.

[79] E. Hintz. *Ein optimaler Algorithmus für das on-line closest pair Problem im k-dim. Raum.* Master's Thesis, Universität Münster, 1993.

[80] D.B. Johnson and T. Mizoguchi. *Selecting the Kth element in $X + Y$ and $X_1 + X_2 + \ldots + X_m$.* SIAM Journal on Computing **7** (1978), pp. 147-153.

[81] S. Kapoor and M. Smid. *New techniques for exact and approximate dynamic closest-point problems.* SIAM Journal on Computing **25** (1996), pp. 775–796.

59

matic pairs for points in the plane. International Journal of Computational Geometry and Applications **5** (1995), pp. 37–51.

[83] J.M. Keil and C.A. Gutwin. *Classes of graphs which approximate the complete Euclidean graph.* Discrete & Computational Geometry **7** (1992), pp. 13–28.

[84] S. Khuller and Y. Matias. *A simple randomized sieve algorithm for the closest-pair problem.* Information and Computation **118** (1995), pp. 34–37.

[85] D.E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison-Wesley, Reading, MA, 1973.

[86] H.-P. Lenhof and M. Smid. *An animation of a fixed-radius all-nearest-neighbors algorithm.* 3rd Annual Video Review of Computational Geometry, 1994. See Proceedings 10th Annual ACM Symposium on Computational Geometry, 1994, page 387.

[87] H.-P. Lenhof and M. Smid. *Sequential and parallel algorithms for the k closest pairs problem.* International Journal of Computational Geometry & Applications **5** (1995), pp. 273–288.

[88] C. Levcopoulos and A. Lingas. *There are planar graphs almost as good as the complete graphs and as short as minimum spanning trees.* Proceedings International Symposium on Optimal Algorithms, Lecture Notes in Computer Science, Vol. 401, Springer-Verlag, Berlin, 1989, pp. 9–13.

[89] G.S. Lueker. *A data structure for orthogonal range queries.* Proceedings 19th Annual IEEE Symposium on the Foundations of Computer Science, 1978, pp. 28–34.

[90] J. Matoušek. *On enclosing k points by a circle.* Information Processing Letters **53** (1995), pp. 217–221.

[91] J. Matoušek. *On geometric optimization with few violated constraints.* Discrete & Computational Geometry **14** (1995), pp. 365–384.

[92] K. Mehlhorn. *Data Structures and Algorithms, Volume 1: Sorting and Searching.* Springer-Verlag, Berlin, 1984.

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

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

[95] K. Mulmuley. *Computational Geometry, an Introduction through Randomized Algorithms.* Prentice Hall, Englewood Cliffs, 1994.

[96] M.H. Overmars. *Dynamization of order decomposable set problems.* Journal of Algorithms **2** (1981), pp. 245–260.

puter Science, Vol. 156, Springer-Verlag, Berlin, 1983.

[98] M.H. Overmars and J. van Leeuwen. *Maintenance of configurations in the plane.* Journal of Computer and Systems Sciences **23** (1981), pp. 166-204.

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

[100] W. Pugh. *Skip lists: a probabilistic alternative to balanced search trees.* Communications of the ACM **33** (1990), pp. 668–676.

[101] M.O. Rabin. *Probabilistic algorithms.* In *Algorithms and Complexity: New Directions and Recent Results.* (J.F. Traub, Ed.) Academic Press, 1976, pp. 21–39.

[102] J. Ruppert and R. Seidel. *Approximating the d-dimensional complete Euclidean graph.* Proceedings 3rd Canadian Conference on Computational Geometry, 1991, pp. 207–210.

[103] J.S. Salowe. *L-infinity interdistance selection by parametric search.* Information Processing Letters **30** (1989), pp. 9–14.

[104] J.S. Salowe. *Constructing multidimensional spanner graphs.* International Journal of Computational Geometry & Applications **1** (1991), pp. 99–107.

[105] J.S. Salowe. *Enumerating interdistances in space.* International Journal of Computational Geometry & Applications **2** (1992), pp. 49–59.

[106] J.S. Salowe. *Euclidean spanner graphs with degree four.* Discrete Applied Mathematics **54** (1994), pp. 55–66.

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

[108] C. Schwarz and M. Smid. *An $O(n \log n \log \log n)$ algorithm for the on-line closest pair problem.* Proceedings 3rd Annual ACM-SIAM Symposium on Discrete Algorithms, 1992, pp. 280–285.

[109] C. Schwarz, M. Smid and J. Snoeyink. *An optimal algorithm for the on-line closest-pair problem.* Algorithmica **12** (1994), pp. 18–29.

[110] R. Seidel. *Backwards analysis of randomized geometric algorithms.* In *New Trends in Discrete and Computational Geometry.* (J. Pach, Ed.) Springer-Verlag, Berlin, 1993, pp. 37–67.

[111] M.I. Shamos. *Geometric complexity.* Proceedings 7th Annual ACM Symposium on the Theory of Computing, 1975, pp. 224–233.

[112] M.I. Shamos and D. Hoey. *Closest-point problems.* Proceedings 16th Annual IEEE Symposium on the Foundations of Computer Science, 1975, pp. 151–162.

61

Journal on Computing **14** (1985), pp. 448–468.

[114] M. Sharir and E. Welzl. *A combinatorial bound for linear programming and related problems.* Proceedings 9th Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science, Vol. 577, Springer-Verlag, Berlin, 1992, pp. 569–579.

[115] D.D. Sleator and R.E. Tarjan. *A data structure for dynamic trees.* Journal of Computer and System Sciences **26** (1983), pp. 362–391.

[116] M. Smid. *Algorithms for semi-online updates on decomposable problems.* Proceedings 2nd Canadian Conference on Computational Geometry, 1990, pp. 347–350.

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

[118] M. Smid. *Maintaining the minimal distance of a point set in less than linear time.* Algorithms Review **2** (1991), pp. 33–44.

[119] M. Smid. *Range trees with slack parameter.* Algorithms Review **2** (1991), pp. 77–87.

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

[121] M. Smid. *Maintaining the minimal distance of a point set in polylogarithmic time.* Discrete & Computational Geometry **7** (1992), pp. 415–431.

[122] M. Smid. *Finding k points with a smallest enclosing square.* Report MPI-I-92-152, Max-Planck-Institut für Informatik, Saarbrücken, 1992.

[123] M. Smid. *Dynamic rectangular point location, with an application to the closest pair problem.* Information and Computation **116** (1995), pp. 1–9.

[124] K.J. Supowit. *New techniques for some dynamic closest-point and farthest-point problems.* Proceedings 1st Annual ACM-SIAM Symposium on Discrete Algorithms, 1990, pp. 84–90.

[125] P.M. Vaidya. *An $O(n \log n)$ algorithm for the all-nearest-neighbors problem.* Discrete & Computational Geometry **4** (1989), pp. 101–115.

[126] P.M. Vaidya. *A sparse graph almost as good as the complete graph on points in K dimensions.* Discrete & Computational Geometry **6** (1991), pp. 369–381.

[127] D.E. Willard and G.S. Lueker. *Adding range restriction capability to dynamic data structures.* Journal of the ACM **32** (1985), pp. 597–617.

related problems. SIAM Journal on Computing **11** (1982), pp. 721–736.

[129] M.H. von Zülow. *Das k-Closest-Pair-Problem.* Master's Thesis, Universität des Saarlandes, Saarbrücken, 1993.