Covering Point Sets and Accompanying Problems

PhD Thesis

Michael Segal

Department of Mathematics and Computer Science
Ben-Gurion University

Beer-Sheva 84105, Israel

segal@cs.bgu.ac.il

1999

Acknowledgements

This thesis would not have been the same without the help of several peoples whose support I gratefully acknowledge. First of all, I would like to thank my advisor Klara Kedem for her patience, guidance, teaching and encouragement throughout my term at Ben-Gurion University. Her advice and constant suggestions, on and off the field of research made it all possible.
Special thanks go to Matya Katz with whom I had so many helpful discussions. For his support and motivation, and for the fun and excitement of doing research together I shall be grateful.
Among those to whom I am deeply indebted are Yuri Rabinovich, for being my roomate, good friend and teaching me to reason about algorithms, Avraham Melkman, who has awakened my interest for partially ordered sets, Dani Berend, for widening my horizons, for insights that led to sharper arguments and better exposition, Shlomi Dolev, for helpfull discussions at several times when I found myself stuck.
Cooperation with Sergei Bespamyatnikh, Frank Nielsen, both via email and in real life has been a pleasure for me.
My cordial thanks go out to Micha Sharir, Günter Rote, Jörg Sack, Arie Tamir for their comments and suggestions.
I would especially like to thank the members of the Mathematics and Computer Science Department who have initiated me into the computer world, and the administrative staff who supported my administrative tasks.
My parents and grandparents have encouraged and supported me during writing this thesis. Thanks for the numerous opportunities that they gave me, their love, support and a lot more.
Above all, I thank Alla, my best discovery in my research. Thanks for making our life even more exciting than the research hours. I would never have managed to finish this work without your care and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The objective of computational geometry is to design efficient algorithms for problems defined on sets of geometric data. The efficiency of the algorithm is measured in terms of its time and space complexity. One of the interesting and extensively researched subjects in computational geometry is *geometric optimization*, which has applications in transportation, station placement, facility location, statistics and other areas. During the years many optimization techniques have been developed such as: parametric searching by Megiddo [79], matrix searching by Frederickson and Johnson [54], expander graphs by Katz and Sharir [72], randomized optimization technique by Chan [26] and others.

Covering problems fall into the area of geometric optimization. Given a set S of n points in a metric space, and given an object Q, a covering problem is defined as "find the smallest object(s) congruent to Q whose union contains a subset of S with given properties". Although the common theme of most of the geometric optimization problems, particularly for covering problems, is that they can be solved using parametric searching, matrix searching, or related optimization techniques, each of them requires a problem-specific, and often fairly sophisticated, approach. The goal of this research is to find efficient algorithms for solving various covering problems and other problems related to them. For achieving our goal we use a battery of techniques, some of which are standard and will be reviewed below. We develop a new framework and a new dynamic optimization technique, that will be described in Chapters 2 and 4, respectively.

The problems that we deal with in this thesis can be divided into the following three groups.

The first group, G_1, is related to *piercing problems* which are, e.g., "Given a set $\mathcal{R}$ of n objects in metric space and some positive integer p, find whether there exists a set P of p points such that each member of $\mathcal{R}$ is intersected by at least one point of P. The members of P are called *piercing points*.

We present several algorithms for fixed values of p and rectangles or

squares serving as objects. Actually, the problems in G_1 group are dual to the following kind of problems: "Given a set S of n points in metric space, an object Q of some fixed size r and some integer p, find p objects congruent to Q whose union contains S".

The second group, G_2, belongs to the class of the classical *facility location problems* and can be described as follows: "Given a set S of n sites (points) in a region contained in a metric space (L_1, L_2 or L_∞ metric), position a point (facility), or a number of facilities, in the region such that the distance between the facility and the sites is minimized or maximized". This type of problems arises when we are asked, eg., to position supermarkets, garbage dumps, postal agences and so on for a bunch of consumers. There is a strong connection between this group of problems and the group G_1. This connection together with a new data structure enable us to provide efficient algorithms for solving several facility location problems.

The third group, G_3, is related to the following kind of problems: "Given a set S of n points in metric space and some positive integer k (which is usually between 1 and n) find some property of the set S that depends on k". For example, find the smallest axis-parallel rectangle that contains exactly k points of S, find the k farthest neighbours for each point of S, enumerate the k smallest or largest distances defined by points of S and so on. We call these problems *k-point problems*. For this kind of problems we have developed a new framework that helps in finding efficient algorithms for L_1 and L_∞ metrics.

The significance of this thesis is developing a new approach for tackling variants of these problems by connecting the three groups of problems and thus getting efficient algorithms, meanwhile coming up with and applying new frameworks and data structures, discovering a new combinatorial structure of the problems and a new variant on sorted matrix optimization technique.

1.1 The methodology of the research

Most of the algorithms that solve covering problems efficiently follow a general pattern. They solve a fixed size *decision problem* in which, given a value d of the size of the geometric object concerned, they determine whether S, or a subset of S, can be covered by objects of this size. The answer to the decision problem is Yes or No. An analysis of the optimal configuration generally leads to a set Σ of candidate values among which the optimal size is found. The set Σ is called the *feasible solution space*. The final stage consists of finding the optimal size d^*. At this stage a search over the set Σ is accomplished by applying a decision algorithm which yields the solution to the whole problem. Unfortunately, it turns out that in most of the covering

problems the size of Σ is too large to allow constructing it explicitly without loss of efficiency of the algorithm. Therefore one has to apply an implicit search technique for locating the optimal solution. Below we review some previous techniques that allow us to solve efficiently numerous optimization problems that appear in the groups G_1 and G_2. As was mentioned above, we were able to find a new framework, based on posets, that helps in finding efficient algorithms the group G_3 problems. This technique will be explained in Chapter 4.

1.2 The parametric search technique

The *parametric search* technique has been proposed by Megiddo [79] for solving efficiently a variety of optimization problems. The technique has recently been successfully applied to a number of geometric selection and optimization problems, e.g., [8, 9, 10, 30, 34, 45]. We want to emphasize that:

- Parametric search usually adds a factor of $\log^2 n$ to the complexity of the decision algorithm.
- Parametric searching requires parallelization of the decision algorithm which in some cases is not easy to implement.

The basic idea behind this method is as follows: Suppose we have a decision problem $\mathcal{P}(n, d)$ that receives as input n data items and a real parameter d, we need to find the minimal value d^* of the parameter d such that $\mathcal{P}(n, d)$ satisfies certain properties. Furthermore, assume that these properties depend on d monotonically, that is, for every real d if $\mathcal{P}(n, d) = Yes$, then $\mathcal{P}(n, d') = Yes$ for $d' > d$ (and, if $\mathcal{P}(n, d) = No$ then $\mathcal{P}(n, d') = No$ for $d' < d$). In particular, there exists a real number d^* such that

$$\mathcal{P}(n, d) = \begin{cases} Yes & \text{if } d \geq d^* \\ No & \text{if } d < d^* \end{cases}$$

Assume we have efficient sequential and parallel algorithms A_s and A_p, respectively, for solving $\mathcal{P}(n, d)$ for any given d. As a result, A_s and A_p can also determine whether the given d is equal to, smaller than, or larger than d^*. Assume that the flow of execution of A_p depends on comparisons, each of which involves testing the sign of a low-degree polynomial in d and in the input items. Megiddo's technique runs the algorithm A_p "generically", without specifying the value of the parameter d, with the intention of simulating its execution at the unknown d^*. If A_p uses P processors and runs in T_p parallel steps, then each such step involves at most P independent comparisons, that is, each can be carried out without having to know the outcome of the others. One can compute the roots $\{d_1, d_2, .., d_{O(P)}\}$ of the P polynomials associated with these comparisons, and run an implicit binary search sequentially to

find among them the interval which contains d^* using the serial algorithm A_s as an oracle. Once the location of d^* among the roots $\{d_1, d_2, .., d_{O(P)}\}$ is known, we can resolve all the P comparisons at d^* and thus resume one parallel step of the algorithm A_p. Each such step constrains the range where d^* can lie, and we thus obtain a nested sequence of progressively smaller intervals, each known to contain d^*, until we either hit d^* as one of the roots being tested, or terminate with a final interval I. Then, by construction, the outcome of A_p will be the same for any $d \in I$. Since we seek the smallest d that satisfies certain properties of the problem $\mathcal{P}(n, d)$, it follows that d^* must be the left endpoint of I. Assume that the runtime of the algorithm A_s is T_s. Then the parametric search requires $O(P + T_s \log P)$ time per parallel step of the algorithm A_p, for a total of $O(PT_p + T_s T_p \log P)$ time. Notice that, since A_p is simulated sequentially, we can assume the weak parallel model of computation of Valiant [101].

Parametric search is a very general technique, and the cost of this generality is the requirement of an efficient parallelization of the decision algorithm.

1.3 Selection and optimization via sorted matrices.

Another approach to parametric search has been proposed by Frederickson and Johnson [54]. It is based on constructing and searching in monotone matrices. We give a brief explanation of this approach. Consider a set S of arbitrary elements. *Selection* in the set S determines, for a given rank k, an element that is k^{th} in some total ordering of S. The complexity of selection in S has been shown to be proportional to the cardinality of the set [25]. Fredrickson and Johnson [54] considered selection in a set of sorted matrices. An $n \times m$ matrix M is a *sorted matrix* if each row and each colum of M is in nondecreasing order. Fredrickson and Johnson have demonstrated that selection in a set of sorted matrices, that together represent the set S, can be done in time sublinear in the size of S. They have also observed that, given certain constraints on the set S, one can construct implicitly the set of sorted matrices representing S. For instance, the sums of the pairs in a Cartesian product of two input sets, denoted by $X+Y$, can be represented by means of the sorted vectors X and Y. In [54, 55, 56], a number of selection and optimization problems on trees were considered. For example, in [56] an $O(n \log n)$ time algorithm for selecting the k^{th} longest path in a tree has been presented.

We illustrate Frederickson and Johnson optimization technique on an example of a single sorted matrix. The main ideas of this technique can be easily extended to a collection of sorted matrices.

Theorem 1.3.1 *[54] Let M be a sorted matrix of dimension $n \times m$, where*

$n \geq m$. Let T_d be the runtime of a decision algorithm which, given a value d, answers 'yes' or 'no'. Assume that the answers are monotone with respect to d. Then the total time needed to find the least element in M, for which the answer is 'yes', is $O(T_d \log n + n)$.

The algorithm performs a sequence of iterations which includes matrices of smaller size in each iteration. The matrices in any iteration are divided into submatrices called *cells*. In each iteration, two representative elements are chosen from each cell, the smallest value and the largest value. These representative elements are used to discard certain cells from further consideration. For ease of exposition it is assumed that M is a square matrix, whose dimension is a power of 2 (if not, we can extend the size of matrix). Hence every cell will be of size which is a power of 4. After a number of iterations all cells consist of single elements. Continue the iterations as before, except without cell division, until a single element remains.

The structure of the matrix induces a partition of the set of remaining cells into subsets called *chains*. Two cells belong to the same chain if and only if they are in the same diagonal of the cells obtained from the original matrix M by partitioning it into submatrices of the same dimensions as the cells. Let b_i be the ratio of the dimension of matrix M to the current cell dimension at the end of the ith iteration. Clearly, $b_i = 2^i$. The maximum possible number of chains after splitting cells on the ith iteration is $2b_i - 1$. Interesting that the number of cells does not increase too quickly as the iterations progress.

Lemma 1.3.2 *[54] Let $B_i = 4b_i - 1$. For all iterations in which the cells are divided, the number of remaining cells after the ith iteration is not greater than B_i.*

From the preceding lemma, the number of cells remaining at the end of iteration $i-1$ is no more than B_{i-1}. Hence no more than $O(B_i)$ work is done in dividing and selecting among cells on the ith iteration. Thus the total work for dividing and selecting cells is $O(\Sigma_i(B_i)) = O(\Sigma_i(b_i)) = O(n)$. Iterations with no cell division will begin when there are $O(n)$ elements. The number of remaining elements will decrease by a factor of 2 each time, yielding $O(n)$ time for the iterations, ultimately giving a least *feasible element* - the entry in matrix with the positive answer. For feasibility testing, $O(\log n)$ iterations with cell division are performed, and $O(\log n)$ iterations without cell division (cells with one element) are performed. Hence all feasibility testing requires $O(T_d \log n)$ time, yielding the total $O(T_d \log n + n)$ runtime of the algorithm.

1.4 Problems solved in this thesis and their background

In covering problems, one looks for an optimal covering of a given set S of n points in the plane by a number of congruent geometric objects (e.g., by a disk, by a strip, by two strips, by two disks, by two squares, etc.) or in d-space by d-dimensional congruent objects. The optimality is measured with respect to some size feature of the geometric object, for example, the radius of the covering disk, the width of the strip or the size of the square. We classified the covering problems we deal with in this thesis into one of the following: piercing problems, facility location problems and k-point problems. In this thesis we enumerate twenty covering problems and denote them by **p1-p20**. For many of these problems we present our algorithms and improvements.

1.4.1 Piercing Problems

In the following we will mention some previous work related to piercing problems and will present our results.

Euclidean p-center

p1: Given a set S of n demand points in d-dimensional space, find a set P of p supply points so that the maximum Euclidean distance between a demand point and its nearest supply point in P is minimized. It can be solved efficiently, when p is small, using the parametric search technique. The decision problem in this case is to determine, for a given radius r, whether S can be covered by the union of p balls of radius r. In some applications, P is required to be a subset of S, in which case the problem is refered to as the *discrete* (or *constrained*) *p-center* problem. Note that Problem **p1** belongs to G_2 while its decision variant belongs to G_1.

For the planar case Hwang et al. [62] gave an $O(n^{O(\sqrt{p})})$ algorithm for the p-center problem. Drezner [42] presented some heuristics for this problem. An algorithm with the same runtime was presented in Hwang et al. [63] for the discrete p-center problem.

Euclidean 1-center

The 1-center problem is to find the smallest ball enclosing S. The decision procedure for the 1-center problem is thus to determine whether S can be covered by a ball of radius r. In other words, we ask whether the set of n balls of radius r centered at the points of S has a non-empty intersection, i.e. is 1-pierceable. For $d = 2$, the decision problem can be solved in $O(\log n)$ parallel steps using $O(n)$ processors, e.g., by testing whether the intersection of the disks of radius r centered at the points of S is nonempty. This yields

an $O(n \log^3 n)$-time algorithm for the planar Euclidean 1-center problem. Using the prune-and-search paradigm, one can solve the 1-center problem in linear time [80], and this approach extends to higher dimensions, where, for any fixed d, the running time is $O(d^{O(d)}n)$ [11]. Chan [26] solved the three dimensional Euclidean discrete 1-center problem in $O(n \log n)$ expected time using a randomized optimization technique.

Euclidean 2-center

In this problem we want to cover a set S of n points in d-dimensional space by two balls of smallest possible common radius. Agarwal and Sharir [7] gave an $O(n^2 \log n)$-time algorithm for determining whether S can be covered by two balls of radius r. Plugging this algorithm into the parametric search machinery, one obtains an $O(n^2 \log^3 n)$-time algorithm for the Euclidean 2-center problem. The runtime of the decision algorithm was improved by Hershberger [59] to $O(n^2)$. It has been used in the algorithm of by Jaromczyk and Kowaluk [64], which runs in $O(n^2 \log n)$ time.

A major progress in this problem was recently made by Sharir [97], who gave an $O(n \log^9 n)$-time algorithm, by combining the parametric search technique with several additional techniques, including a variant of the matrix search algorithm of Frederickson and Johnson [54]. Eppstein [48] has simplified Sharir's algorithm, using randomization and different data structures, and obtained an improved solution, whose expected runtime is $O(n \log^2 n)$.

Recently Agarwal et al. [12] have developed an $O(n^{\frac{4}{3}} \log^5 n)$-time algorithm for the discrete 2-center problem.

Rectilinear p-center

In this problem the metric is L_∞. The decision problem in this case is as follows.

p2: Let $\mathcal{R}$ be a set of compact convex regions (rectangles) with nonempty interior, in the plane, where every region $r \in \mathcal{R}$ is assigned a *scaling point* c_r in its interior. We call set $\mathcal{R}$ p-pierceable if there exist a set of p points that intersects each member of $\mathcal{R}$. Our problem, thus, is to determine whether $\mathcal{R}$ is p-pierceable, and, if so, to produce a set of p piercing points. For $r \in \mathcal{R}$ and a real number $\lambda \geq 0$, let $r(\lambda)$ be the homothetic copy of r obtained by scaling r by the factor λ about c_r (i.e., $r(\lambda) = \{c_r + \lambda(a - c_r) | a \in r\}$). Finally, $\mathcal{R}(\lambda) = \{\ r(\lambda) |\ r \in \mathcal{R}\}$. The p-center problem for $\mathcal{R}$ looks for

$$\lambda_{\mathcal{R}} = min\{\ \lambda | \mathcal{R}(\lambda) \text{ is } p\text{-pierceable }\}.$$

If $\mathcal{R}$ is a set of translates of a square and the scaling points are the respective centers, then we talk about the *rectilinear p-center problem*. If the squares are still axis-parallel but of possible different sizes, then we have the *weighted rectilinear p-center problem*, and if $\mathcal{R}$ is a set of arbitrary axis-parallel rectangles (and the scaling points are also arbitrary), then we face the *general*

rectilinear p-center problem. In other words, in the *p-center problem* we are given a set S of n points in the plane, some compact convex set C, and a positive integer p. The goal is to find p isothetic copies of C of smallest possible scaling factor, whose union covers S. The paper of Sharir and Welzl [99] explains a reduction from the p-center problem to the p-piercing problem.

The rectilinear 1-center problem is trivially solved in linear time, and a polynomial time algorithm for the rectilinear 2-center problem is given in [81]. A linear time algorithm for the planar rectilinear 2-center problem is given by Drezner [43]. There are several papers in which the p-piercing problem for axis-parallel rectangles is investigated; we mention only the very recent papers. The 1-piercing problem was solved in linear time using the observation that 1-piercing problem for rectangles is equivalent to finding whether the intersection of rectangles is empty or not. In Sharir and Welzl [99] 2- and 3-piercing problems in the plane are solved in linear time, while they achieve an $O(n \log^3 n)$ bound for the 4-piercing problem and $O(n \log^4 n)$ bound for the 5-piercing problem. Katz and Nielsen [71] present a linear time algorithm for d-dimensional boxes ($d \geq 2$) for 2-piercing problem. Sharir and Welzl [99] have developed a linear expected time algorithm for the rectilinear 3-center problem, by showing that it is an LP-type problem. They have also obtained an $O(n \log n)$-time algorithm for computing a rectilinear 4-center and an $O(n \log^5 n)$-time algorithm for computing a rectilinear 5-center. The algorithms for the 4-center and 5-center employ the Frederickson-Johnson matrix searching technique. Recently, Chan [26] has developed $O(n \log^4 n)$ expected time algorithm for rectilinear 5-center problem. In Chapter 2 we present efficient algorithms for finding a piercing set (i.e., a set of p points as above) for values of $p = 1, 2, 3, 4, 5$ (Problem **p2**). Our algorithms for 4 and 5-piercing improve the existing result of $O(n \log^3 n)$ and $O(n \log^4 n)$ to $O(n \log n)$ time. The result for 5-piercing can be applied as an $O(n \log^2 n)$ time algorithm for the planar version of Problem **p1**, L_∞ metric and $p = 5$. Applying the technique of Chan [26] immediately leads to the $O(n \log n)$ expected time algorithm for this problem. We improve the existing $O(n^{p-4} \log^5 n)$ time algorithm [99] for a general (but fixed) p to $O(n^{p-4} \log n)$ running time, and we also extend our algorithms to higher dimensional space. Recently, Nussbaum [86] and Makris and Tsakalidis [76] presented algorithms with similar runtimes for various piercing problems.

Euclidean p-line-center

p3: Let S be a set of n points in d-dimensional space and δ be the Euclidean distance function. We wish to compute the smallest real value w^* so that S can be covered by a union of p strips of width w^*. Problem **p3** does not belong (directly) to any of the groups G_1, G_2, G_3, but satisfies the definition of covering problem.

The 1-line center is the classical *width* problem. For $d = 2$, an $O(n \log n)$-

time algorithm was given by Houle and Toussaint [61]. For the 2-line center problem in the plane, Agarwal and Sharir [7] present an $O(n^2 \log^5 n)$-time algorithm, using parametric search. This algorithm is very similar to their 2-center algorithm, i.e., the decision algorithm finds all subsets of S that can be covered by a strip of width w and for each such subset S_1, it determines whether $S - S_1$ can be covered by another strip of width w. The runtime for the optimization problem was improved to $O(n^2 \log^4 n)$ by Katz and Sharir [72] who use expander graphs, and by Glozman et al. [57] who apply Frederickson-Johnson matrix search technique. The best known algorithm, by Jaromczyk and Kowaluk [66], runs in $O(n^2 \log^2 n)$ time and does not use any of the mentioned above optimization techniques. It is an open problem whether a near-linear (or just subquadratic) time algorithm exists for computing a 2-line center problem.

Two-covering

p4: Given a set S of n points in d-dimensional space, $d \geq 2$, find two axis-parallel boxes b_1 and b_2 that together cover the set S and minimize the maximum of measures $\mu(b_1)$ and $\mu(b_2)$, where μ is a monotone function of the box, i.e. $b_1 \subseteq b_2$ implies $\mu(b_1) \leq \mu(b_2)$. Examples of the box measure μ are the volume of the box, the perimeter of the box, the length of the diagonal etc. The min-max two box problem is a classical "covering problem". On the other hand it belongs to "partition problems" where we are interested in partitioning a set of points into two subsets (not necessarily disjoint) in order to optimize some given function of the "sizes" of two subsets [13, 60, 84, 85].

This problem is closely related to the rectilinear p-center problem (and in particular to the 2-center problem) - **p1** for L_∞. In a very recent paper Sharir and Welzl [99] using LP-type framework and Helly-type results obtained an $O(n)$ expected time algorithm for the general rectilinear 2-center problem in the plane. Hershberger and Suri [60] solve the following problem: Given a planar set of points S, a *rectangular measure* μ acting on S and a pair of values μ_1 and μ_2, does there exist a bipartition $S = S_1 \cup S_2$ satisfying $\mu(S_i) \leq \mu_i$ for $i = (1, 2)$? They present an algorithm which solves this problem in $O(n \log n)$ time. Based on this algorithm and using the sorted matrix technique of Frederickson and Johnson [54], Glozman et al. [57] obtained an $O(n \log n)$ time algorithm that solves min-max box problem in the plane.

In Chapter 2 we present an efficient algorithm for solving the min-max two box (problem **p4**) for fixed arbitrary dimension $d \geq 2$. The runtime of the algorithm is $O(n \log n + n^{d-1})$. Our algorithm is simpler than that in [57] for the planar case.

Center problems

We consider yet another version of rectilinear p-center problem. We are given a set S of n demand points and a set C of m supply points in the plane. Call a square (rectangle) *discrete* or *constrained* if its center lies on some point of C. One can define the following problems:
p5: Find two constrained axis-parallel squares whose union covers S, so as to minimize the size of the larger square.
p6: Find two constrained parallel squares whose union covers S, so as to minimize the size of the larger square. The squares are allowed to rotate but must remain parallel to each other.
p7: Find two constrained squares whose union covers S, so as to minimize the size of the larger square, where each square is allowed to rotate independently.

The three problems above continue a list of optimization problems that deal with covering a set of points in the plane by two geometric objects of the same shape. We mention some of them: The two center problem, solved in time $O(n \log^9 n)$ by Sharir [99], and recently in time $O(n \log^2 n)$ by Eppstein [49] (by a randomized algorithm); the constrained two center problem, solved in time $O(n^{\frac{4}{3}} \log^5 n)$ by Agarwal et al. [12]; the two line-center problem, solved in time $O(n^2 \log^2 n)$ by Jaromczyk and Kowaluk [66] (see also [57, 72]); the two square-center problem, where the squares are with mutually parallel sides (the unconstrained version of Problem **p6**), solved in time $O(n^2)$ by Jaromczyk and Kowaluk [65]. The algorithm in [65] is based on a new data structure called $\alpha - silhouette$ which allows efficient maintenance of the point dominances.

In Chapter 2 we describe an $O(n \log^2 n)$-time and $O(n \log n)$ space algorithm for Problem **p5** when $C = S$, $O(\max(n \log n,\ m \log n(\log n + \log m)))$-time algorithm for general C and $O(mn \log m \log n)$-time algorithm for the case of rectangles and general C. We also consider the dynamic versions of this problem where the points of S are allowed to be inserted or deleted.

For Problem **p6** ($C = S$) our algorithm runs in $O(n^2 \log^4 n)$ time and uses $O(n^2)$ space. Finally, we solve Problem **p7** ($C = S$) by an $O(n^3 \log^2 n)$-time and $O(n^2)$-space algorithm.

1.4.2 Facility Location Problems

There is a very strong relationship between the problems that belong to groups G_1 and G_2. In some sense the decision version of a problem that belongs to G_2 group usually belongs to G_1 as well. The facility location is a classical problem of operations research that has also been examined in the computational geometry community. The task is to position a point in the plane (a *facility*) such that a distance between the facility and given points (*sites*) is minimized or maximized. Most of the problems described in the facility location literature are concerned with finding a "desirable" facility

location: the goal is to *minimize* a distance function between the facility (*e.g.*, a service) and the sites (*e.g.*, the customers). Just as important is the case of locating an "undesirable" or obnoxious facility. In this case instead of minimizing the largest distance between the facility and the destinations, we maximize the smallest distance. Applications for the latter version are, *e.g.*, locating garbage dumps, dangerous chemical factories or nuclear power plants. The latter problem is unconstrained if the domain of possible locations for the facility is the entire plane. Practically the location of the facility should be in a bounded region R, whose boundary may or may not have a constant complexity description.

Undesirable location

p8: Let S be a set of n points in the plane, enclosed in a rectangular region R. Let each point p of S have two positive weights $w_1(p)$ and $w_2(p)$. Find a point $c \in R$ which maximizes

$$\min_{p \in S}\{\max\{w_1(p) \cdot d_x(c,p), w_2(p) \cdot d_y(c,p)\}\},$$

where $d_x(c,p)$ defines the distance between the x coordinates of c and p, and $d_y(c,p)$ defines the distance between the y coordinates of c and p.

Problem **p8** is concerned with locating an obnoxious facility in a rectangular region R under the weighted L_∞ metric, where each site has two weights, one for each of the axes. An application for two-weighted distance is, *e.g.*, an air pollutant which is carried further by south-north winds than by east-west winds. For the *unweighted* case of this problem, where R is a simple polygon with up to n vertices and under the Euclidean metric, Bhattacharya and Elgindy [23] present an $O(n \log n)$ time algorithm. For weighted sites one can construct the Voronoi diagram and look for the optimal location either on a vertex of this diagram or on the boundary of the region R. For weighted sites, the Voronoi diagram is known to have quadratic complexity in the worst case, and it can be constructed in optimal $\Theta(n^2)$ time [16]. Thus, the optimal location, using the Voronoi diagram, can be found in $O(n^2)$ time [51]. The first subquadratic algorithm for the weighted problem under L_∞ metric and a rectangular region R was presented by Follert et al. [52]. Their algorithm runs in $O(n \log^4 n)$ time.

Desirable location

p9: Given a set S of n points and a number $1 \leq k \leq n-1$, find a point p such that the sum of the $L_1(L_\infty)$ distances from p to its k nearest neighbors in S is minimized.

Problem **p9** deals with locating a desirable facility under the *min-sum* criterion. Some applications for this problem are locating a component in a VLSI chip or locating a welding robot in an automobile manufacturing plant.

Elgindy and Keil [47] consider a slight variation of the problem under the L_1 metric: Given a positive constant D, locate a facility c that maximizes the number of sites whose sum of distances from c is not greater than D. They consider the discrete and continuous cases. The runtimes of their algorithms are $O(n \log^4 n)$ for the discrete case and $O(n^2 \log n)$ for the continuous case, respectively.

Facilities in region

Another variant of the facility location problem is to place k obnoxious facilities, with respect to n given demand sites and m given regions, where the goal is to *maximize* the minimal distance between the demand sites and the facility under the constraint that each of the regions must contain at least one facility. More specifically,
p10: Let S be a set of n points in the plane (called *demand* points), and let $\mathcal{R}$ be a set of m, $m \leq n$, regions in the plane (called *neighborhoods*). Let k be a positive integer (k is the number of facilities, e.g., garbage dumps, to be placed). Find k sites $c_1, \ldots, c_k$ for the k facilities, such that (i) $C = \{c_1, \ldots, c_k\}$ is a *piercing set* for $\mathcal{R}$, that is, each of the neighborhoods in $\mathcal{R}$ is served by at least one facility that is located in the neighborhood. (ii) The minimal distance between a demand point in S and a site in C is maximized. Brimberg and Mehrez [24] solve the following problem: Find k locations in the rectangle R (for k facilities), such that (i) the distance between any two locations is at least some given value d, and (ii) the minimal distance between a demand point and a facility is at least some given value r. The running time of their algorithms is $O(n^{2k})$.

We solve problems **p8**, **p9**, **p10** in Chapter 3. For Problem **p8** we present $O(n \log^2 n)$ runtime algorithm. Our algorithm for Problem **p9** runs in time $O(n \log^2 n)$ for the discrete case, and for the continuous case in $O((n-k)^2 \log^2 n + n \log n)$ time. We also solve several variants of problem **p10**. For example, for L_∞ metric and for $k = 2$, Problem **p10** can be solved in $O(n \log^2 n)$ time.

1.4.3 k-point Problems

Below we list several problems that belong to G_3. Some of them appear as natural extensions of problems from G_1 and G_2. By applying our posets framework, we were able to obtain efficient algorithms for this set of problems.

k-point rectangle

A closely related problem to the rectilinear 1-center problem is the following:
p11: Given a set S of n points in the plane and an integer k, find the smallest axis-parallel rectangle (smallest in terms of perimeter or area) that encloses

exactly k points of S. One can also think about the query version of the problem above:
p12: Given a number k decide whether a query rectangle contains k points or less.

Problem **p11** has been investigated by many researchers, some of whose results we cite below. Aggarwal et al. [3] present an algorithm which runs in time $O(k^2 n \log n)$ and uses $O(kn)$ space. Eppstein et al. [49] and Datta et al. [36] show that this problem can be solved in $O(n \log n + k^2 n)$ time; the algorithm in [49] uses $O(kn)$ space, while the algorithm in [36] uses $O(n)$ space. These algorithms are efficient for small k values, but become inefficient for large k values. The paper of Chan [26] presents an $O(n \log n)$ expected time algorithm for finding the minimum L_∞-diameter k-point subset of a planar n-point set, i.e. for finding the smallest square that encloses exactly k points.

For Problem **p11** we present an efficient algorithm [95] for k values in the range $\frac{n}{2} < k \leq n$. It is based on *posets* [1] and runs in time $O(n + k(n-k)^2)$ and $O(n)$ space.

Problem **p12** is a variant of orthogonal range searching where we are given a set S of n points and want to find how many points are enclosed in the query rectangle. This problem was efficiently solved by Bentley [19] in $O(\log n)$ query time, using the range search tree and with preprocessing time and space $O(n \log n)$. For this problem we obtain an algorithm with $O(n + (n-k)\log n)$ preprocessing time and space and $O(\log (n-k))$ query time (for $k \geq \frac{n}{2}$). We also show how to extend the algorithms of both **p11** and **p12** for higher dimensional space.

k-point circle

A problem related to the ones we tackle in this thesis is: **p13:** Given a set S of n points in the plane and an integer k, find a disk of the smallest radius that contains k of the n input points. The best known deterministic algorithm runs in time $O(n \log n + nk \log k)$ using $O(n + k^2 \log k)$ space [46]. Matoušek [78] also showed that the smallest disk covering all but k points can be computed in time $O(n \log n + k^3 n^\varepsilon)$ for any $\varepsilon > 0$.

Selecting distances

p14: Let S be a set of n points in the plane, and let $1 \leq k \leq \frac{n(n-1)}{2}$. We wish to compute the k-th smallest distance between a pair of points of S. The solution can be obtained using a parametric searching. The decision problem is to compute, for a given real r, the sum $\Sigma_{p \in S} |D_r(p) \cap (S - \{p\})|$, where $D_r(p)$ is the closed disk of radius r centered at p. Agarwal et al. [6] gave an $O(n^{\frac{4}{3}} \log^{\frac{4}{3}} n)$ expected-time randomized algorithm for the deci-

sion problem, which yields an $O(n^{\frac{4}{3}}\log^{\frac{8}{3}} n)$ expected-time algorithm for the distance-selection problem. Goodrich [58] derandomized this algorithm, at a cost of an additional polylogarithmic factor in the runtime. Katz and Sharir [72] obtained an expander-based $O(n^{4/3}\log^{3+\varepsilon} n)$-time deterministic algorithm for this problem. By applying a randomized approach Chan [27] was able to obtain an $O(n\log n + n^{2/3}k^{1/3}\log^{5/3} n)$ expected time algorithm for this problem.

The problem of selecting distances is closely related to the problem of finding the nearest (farthest) neighbors. The problem is defined as follows. **p15:** Find the k nearest (farthest) neighbors for each point of S. Dickerson et al. [39] present an algorithm for this problem which runs in time $O(n\log n + nk\log k)$, and works for any convex distance function. Eppstein and Erickson [49] solve the problem on a random access machine model in time $O(n\log n + kn)$ and $O(n\log n)$ space. In the algebraic desicion tree model their time bound increases by a factor of $O(\log\log n)$. Flatland and Stewart [50] present an algorithm which runs in time $O(n\log n + kn)$ in the algebraic decision tree model. Finally, a recent paper of Dickerson and Eppstein [41] describes an $O(n\log n + kn)$ time and $O(n)$ space algorithm for this problem. This algorithm works for any metric and is extendable to higher dimensions. Our algorithm [93] for L_∞ metric runs in time $O((n-k)n)$ (assuming $k \geq \frac{n}{2}$), uses linear space and has the same runtime for any fixed, high dimension.

The following enumeration problems is tightly connected to the previous problem.
p16: Enumerate the k largest (smallest) rectilinear distances in decreasing (increasing) order and,
p17: given a distance $\delta > 0$, find all the pairs of points of S which are of rectilinear distance δ or less (more). For our best knowledge only two papers, one by Dickerson and Shugart [40] and one by Katoh and Iwano [67] present an algorithm for the second problem (for the *largest* k distances). The algorithm in [40] works for any metric, and requires $O(n+k)$ space with expected runtime of $O(n\log n + \frac{k\log k\log n}{\log\log n})$. The paper of Katoh and Iwano [67] presents an algorithm for the second problem for the L_2 metric with running time $O(\min(n^2, n\log n + k^{4/3}\log n/(\log k)^{1/3}))$ and space $O(n + k^{4/3}/(\log k)^{1/3} + k\log n)$. Their algorithm is based on the k nearest neighbor Voronoi diagrams. Dickerson et al. [39] present an algorithm for the problem: enumerate all the k *smallest* distances in S in *increasing* order. Their algorithm works in time $O(n\log n + k\log k)$ and uses $O(n+k)$ space. Lenhof et al. [75], Salowe [90], Dickerson and Eppstein [41] solve this problem too but they just report the k closest pairs of points without sorting the distances, spending $O(n\log n + k)$ time and $O(n+k)$ space. An algorithm for solving the enumerating problem (for the smallest k distances) is also presented in [41], spending $O(n\log n + k\log k)$ time and using $O(n+k)$ space. Chan [27] present $O(n\log n + k)$ expected time simlple algorithm for reporting the k closest pairs of points that is based on the Lenhof et al. [75]

algorithm. He also proposed an $O(n\text{polylog}\, n + k)$ expected time algorithm for enumeration the k farthest pairs. Dickerson and Eppstein [41] considered the following problem: find all pairs of points of S separated by distance δ or less. They give an $O(n \log n + k)$ time and $O(n)$ space algorithm, where k is the number of distances not exceeding δ. For the enumeration Problem **p16** we present two algorithms: one for enumerating the largest and the other for the smallest distances. The first runs in time $O(k \log n + n)$, and uses $O(n)$ space. The second runs in time $O(n \log n + k \log n)$, and uses $O(n)$ space. These algorithms can be easily extended to high dimensional space without affecting the runtime and space requirements. For Problem **p17** we give algorithms with the similar running times as in [41] for the rectilinear distances.

Circle fitting

p18: Given a set S of n points in the plane, fit a circle C through S so that the maximum distance between the points of S and the circle C is minimized. This is equivalent to finding an annulus of minimum width containing S. Ebara et al. [44] observed that the center of a minimum-width annulus is either a vertex of the closest-point Voronoi diagram of S, or a vertex of the farthest-point Voronoi diagram, or an intersection point of a pair of edges of the two diagrams. Based on this observation, they obtained a quadratic time algorithm. Using parametric search, Agarwal et al. [10] have shown that the center of the minimum-width annulus can be found without checking all of the $O(n^2)$ candidate intersection points explicitly; their algorithm runs in $O(n^{\frac{8}{5}+\varepsilon})$ time. Using randomization and an improved analysis, the expected runtime has been improved to $O(n^{\frac{3}{2}+\varepsilon})$ by Agarwal and Sharir [5]. Finding an annulus of minimum area that contains S is a simpler problem, since it can be formulated as an instance of linear programming in 4-dimensional space, and can thus be solved in $O(n)$ time.

Some variations of Problem **p18** have been considered in previous papers. Efrat et al.[46] consider the problem from the group G_3 of enclosing k points within a minimal area circle and pose an open problem of covering k points by a ring. They gave two solutions for the smallest k-enclosing circle. When using $O(nk)$ storage, the problem can be solved in time $O(nk \log^2 n)$. When only $O(n \log n)$ storage is allowed, the running time is $O(nk \log^2 n \log \frac{n}{k})$. The problem of computing the roundness of a set of points, which is defined as the minimum width concentric annulus that contains all points of the set was solved in [5, 73, 100]. The best known running time is $O(n^{\frac{3}{2}+\epsilon})$, given in [5], where $\epsilon > 0$ is an arbitrary small constant. The paper of Barequet et al. [18] presents algorithms for several variants of the polygon annulus placement problem: given an input polygon P and a set S of points, find an optimal placement of P that maximizes the number of points in S that fall in a certain annulus region defined by P and some offset distance $\delta > 0$.

Other variants on the problem of circle fitting are :
p19: find the smallest ring that contains k ($k \geq \frac{n}{2}$) points of S,
p20: find the smallest area sector in a constrained circular ring that covers $k \geq \frac{n}{2}$ points. We can consider Problem **p19** for circular or rectangular ring and for both constrained and unconstrained case (recall that by *constrained* we mean that the center of the ring is one of the points of S). A *rectangular ring* consists of two concentric rectangles, the internal rectangle fully contained in the external one. As a measure we take the maximum width or area of the ring. We solve Problem **p19** in $O(n(n-k)\log(n-k))$ time and $O(n)$ space for rectangular ring, constrained case, while for the non constrained case we present an algorithm with runtime $O(n(n-k)^4 \log n)$ and $O(n)$ space. For a circular ring that covers k ($k \geq \frac{n}{2}$) points (Problem **p19**) we present an algorithm that runs in $O(n^2 + n(n-k)\log n)$ time and uses $O(n)$ space, and we find a *sector* of a constrained circular ring (Problem **p20**) that covers k points ($k \geq \frac{n}{2}$) in $O(n^2 + nk(n-k)^2)$ time and $O(n^2)$ space. We also show how to extend all of the above algorithms to higher dimensional space.

A possible motivation for this kind of problems (group G_3) is to cover all but a small number of points by one or more objects comes from statistics. In the analysis of statistical data one would like to get rid of outliers in the data. Assuming $n-k$ data points are outlyers, one way to find the "good" k data points is to enclose them in a small given shape (or shapes). Chapter 4 is dedicated to these problems.

On the next page we present a table in which we summarize our and previous results for the problems described above.

Pbm	*Previous results*	*Our results*
2	$p = 4$: $O(n \log^3 n)$[99] $p = 5$: $O(n \log^4 n)$[99]	$O(n \log n)$
4	$O(n \log n)$, $d = 2$,[57]	$O(n \log n + n^{d-1})$, $d \geq 2$
5	no result	$O(n \log^2 n)$
6	no result	$O(n^2 \log^4 n)$
7	no result	$O(n^3 \log^2 n)$
8	$O(n \log^4 n)$[52]	$O(n \log^3 n)$
9	no results, discrete no results, continuous	$O(n \log^2 n)$ $O((n-k)^2 \log^2 n + n \log n)$
10	no result	$O(n \log^2 n)$, L_∞, $k = 2$
11	$O(n \log n + kn)$[36]	$O(n + (n-k)^2 n)$
12	Preprocess: $O(n \log n)$ Query: $O(\log n)$ [19]	$O(n + (n-k) \log n)$ $O(\log (n-k))$
15	$O(n \log n + kn)$ [41]	$O(n \log n + (n-k)n)$
16	$O(n \log n + \frac{k \log k \log n}{\log \log n})$ (expected) [40]	$O(n + k \log n)$
17	$O(n \log n + k)$ [41]	$O(n \log n + k)$
19	no results, constrained no results, non constrained	$O(n(n-k) \log (n-k)$ $O(n(n-k)^4 \log n)$
20	no results	$O(n^2 + nk(n-k)^2)$

Table 1.1: Summary of best previous results and our results.

Chapter 2

Piercing Problems

All the problems in this section are defined as follows. "Given a set $\mathcal{R}$ of n objects in metric space and some positive integer p, find whether there exists a set of p points that intersects each member of $\mathcal{R}$. Each member of the set of p points is called a *piercing point*. Determining whether a set of some n objects is 1-pierceable means that we look for a set with one point p contained in each member of $\mathcal{R}$. This problem is equivalent to determining whether this set has a non-empty intersection. More specifically, we consider the following problems mentioned in Introduction:

p2: Given a collection of axis-parallel rectangles in the plane, determine whether there exists a set of $p = 1, 2, 3, 4, 5$ points whose union intersects all the given of p points as above) for values of p. We also consider this problem in higher dimensional space.

p4: Given a set S of n points in d-dimensional space, $d \geq 2$, find two axis-parallel boxes that together cover the set S and minimize the maximum of measures of boxes, where the measure is a monotone function of the box.

p5, p6, p7: Given a set S of n points in the plane, we seek two squares whose center points belong to S, their union contains S, and the area of the larger square is minimal. We consider three variants of this problem: In the first (**p5**) the squares are axis parallel, in the second (**p6**) they are free to rotate but must remain parallel to each other, and in the third (**p7**) they are free to rotate independently.

2.1 Rectilinear piercing (p2)

In this section we consider the rectilinear piercing problems. In the following subsections we present efficient algorithms for determining whether s set of n rectangles in the plane is p-pierceable, for a small values of p and also solve the d-dimensional case.

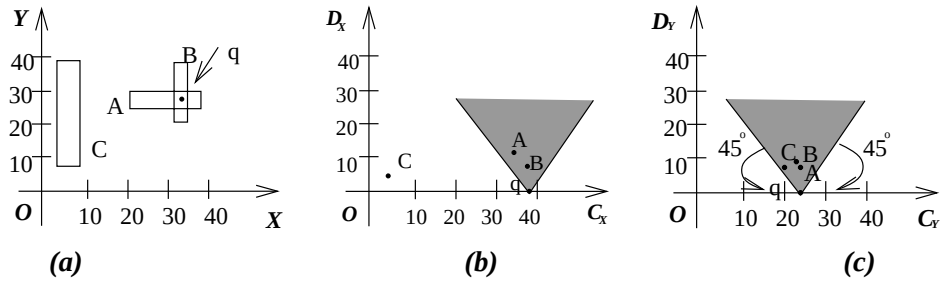

Figure 2.1: (a) There are 3 rectangles, and q is a query point. All intervals containing q are in the shaded regions. Intervals appearing in the shaded regions of both (b) and (c) correspond to rectangles that contain P.

2.1.1 Rectilinear 1-piercing

We are given a set $\mathcal{R}$ of n axis-parallel rectangles in the plane; The goal is to decide whether their intersection is empty or not. We assume that there is no axis-parallel line that traverses $\mathcal{R}$. The case of the existence of such a line was considered in [99]. They show that this case can be easily solved in linear time for any fixed p.

We begin with an observation due to Samet [92]: If a shape M is described by k parameters, then this set of parameter values defines a point in a k-dimensional space assigned to the class of shapes. Such a point is termed a *representative point*. Note, that a representative point and the class to which it belongs completely define all of the topological and geometric properties of the corresponding shape.

The class of two-dimensional axis-parallel rectangles in the plane is described by a representative point in four dimensional space. One choice for the parameters is the x and y coordinates of the centroid of the rectangle, denoted by c_x, c_y, together with its horizontal and vertical extents (i.e. the horizontal and vertical distances from the centroid to the relevant sides), denoted by d_x, d_y. In this case a rectangle is represented by the four-tuple (c_x, d_x, c_y, d_y) interpreted as the Cartesian product of a horizontal and a vertical one-dimensional *interval* : (c_x, d_x) and (c_y, d_y), respectively.

Let P be the set of 4 dimensional points representing the parameters of $\mathcal{R}$. Let $P_x = \{p_1^x, \ldots, p_n^x\}$ be the projections of the x-intervals of $\mathcal{R}$ into the plane (c_x, d_x), and let $P_y = \{p_1^y, \ldots, p_n^y\}$ be the projections of the y-intervals of $\mathcal{R}$ into the plane (c_y, d_y). A query that asks which rectangles contain a given point is easy to implement (see Figure 2.1).

A query point q is represented by a four-tuple $(q_x, 0, q_y, 0)$. We transform the rectangles (A,B,C) in Figure 2.1(a) into the points in two 2-dimensional spaces ((c_x, d_x) and (c_y, d_y)) (Figure 2.1(b) and 2.1(c)). There are two points representing q in these 2-dimensional spaces. $(q_x, 0)$ in (c_x, d_x)-space, and $(q_y, 0)$ in (c_y, d_y)-space. It is easy to see that all the rectangles that contain Q must be transformed into two cones in these spaces respectively (the

shaded cones in Figure 2.1). These cones have apexes on $(q_x, 0)$ and $(q_y, 0)$ respectively and are of slopes 45° and 135°. In Figure 2.1, A and B are in both cones and thus q is in these rectangles.

In order to find whether the set $\mathcal{R}$ is 1-pierceable, we check whether there exist two cones C_1 and C_2, C_1 apex is in the plane (c_x, d_x) and C_2 apex is in the plane (c_y, d_y), each cone is of slope 45° and 135°, such that C_1 and C_2 cover all the points in each one of the planes. Formally, we find in each plane the rightmost intersection point R_x (R_y) of the 45° lines through the points of P_x (P_y) with the c_x (c_y) axis. The point R_x defines the right boundary of C_1, while the point R_y defines the right boundary of C_2. Then we find the leftmost intersection point (L_x and L_y respectively) of the 135° lines with through P_x (P_y) with the c_x (c_y) axis. These latter points define the left boundaries of the cones C_1 and C_2, respectively.

The existence of C_1 and C_2 is equivalent to the existence (not emptiness) of non-empty intervals $[R_x, L_x]$ and $[R_y, L_y]$. If the above is true, then any point Q whose x and y projections are in these intervals, respectively, is a piercing point.

Thus, we can conclude by the following theorem:

Theorem 2.1.1 *We can find whether a set of n axis-parallel rectangles is 1-pierceable in $O(n)$ time, and compute a piercing point, if it exists, in the same runtime.*

2.1.2 Rectilinear 2- and 3-piercing

We begin with the 2-piercing problem. Similarly to the previous section, we have to find whether there exist four cones $C_1, C_2 \in (c_x, d_x)$ and $C_3, C_4 \in (c_y, d_y)$ such that:

1. $C_1 \cup C_2$ covers P_x.
2. $C_3 \cup C_4$ covers P_y.
3. Denote by $[C_i]$ the set of the points from P_x (or P_y) that are covered by C_i. At least one of the following two conditions is true:
   (i) $([C_1] \cap [C_3]) \cup ([C_2] \cap [C_4])$ contains all the points of P. This will imply that the apexes of C_1, C_3 define one piercing point and apexes of C_2, C_4 define the other piercing point.
   (ii) $([C_1] \cap [C_4]) \cup ([C_2] \cap [C_3])$ contains all the points of P. This will imply that the apexes of C_1, C_4 define one piercing point and apexes of C_2, C_3 define the other piercing point.

We can *constrain* the locations of the cones C_1, C_2, C_3, C_4. They are defined by minimal and maximal points of intersection of the 45° and 135° lines

with the horizontal axes in the two planes (c_x, d_x) and (c_y, d_y) respectively. It is easy to see that in order for the rectangles to be 2-pierceable, we put, without loss of generality, the apex of C_1 on R_x, C_2 on L_x, C_3 on R_y and C_4 on L_y. Clearly, if these cones cover all the points then the set $\mathcal{R}$ is 2-pierceable.

In the case of 3-piercing, we have to find six cones, $C_i, 1 \leq i \leq 6$, which will define three piercing points with the following properties:

1. $C_1 \cup C_2 \cup C_3$ covers P_x.
2. $C_4 \cup C_5 \cup C_6$ covers P_y.
3. For $i, k, z \in \{1, 2, 3\}$, pairwise disjoint and $j, l, h \in \{4, 5, 6\}$, pairwise disjoint

$$|([C_i] \cap [C_j]) \cup ([C_k] \cap [C_l]) \cup ([C_z] \cap [C_h])| = n$$

for at least one combination of i, k, z (there are at most 6 combinations), where the union is taken without repetitions.

Without loss of generality, we can find the constrained cones C_1, C_3, C_4, C_6 as in the algorithm for 2-piercing. Namely, the left boundary of C_1 (C_4) is constrained by the leftmost 135° line through the points of P_x (P_y), and the right boundary of C_3 (C_6) is constrained by the rightmost 45° line through the points of P_x (P_y).

To fulfill condition (3) we look at each combination: $[C_1] \cap [C_4]$ or $[C_1] \cap [C_6]$ or $[C_3] \cap [C_4]$ or $[C_3] \cap [C_6]$ and for these four possibilities we check, in linear time, whether the rest of the points is 2-pierceable. Thus we conclude:

Theorem 2.1.2 *We can check in linear time whether a set of n axis-parallel rectangles is 2- or 3-pierceable and find a solution, if exists, in the same running time.*

Our method is, in a sense, dual to that of Sharir and Welzl paper [99], as we will show below. Sharir and Welzl define ℓ^L to be the vertical line containing the leftmost right edge of a rectangle in $\mathcal{R}$, ℓ^R is the vertical line containing the rightmost left edge of a rectangle in $\mathcal{R}$, let ℓ^T is the horizontal line containing the highest bottom edge of a rectangle in $\mathcal{R}$, and let ℓ^B s the horizontal line containing the lowest top edge of a rectangle in $\mathcal{R}$. They further consider the closed left halfplane H^L bounded by ℓ^L, the closed right halfplane H^R bounded by ℓ^R, the closed top halfplane H^T bounded by ℓ^T, and the closed bottom halfplane H^B bounded by ℓ^B. Let $\tilde{H}^X$ denote the closure of the complement of H^X, for $X = L, R, T, B$. They note that if no axis-parallel line traverses $\mathcal{R}$, then H^L is disjoint from H^R, and H^T is disjoint from H^B, and thus $R_0 := \bigcap_{X \in \{L,R,T,B\}} \tilde{H}^X$ is nonempty (with nonempty interior). R_0 is called the *location domain for* $\mathcal{R}$.

Sharir and Welzl show that for 2 or 3-pierceability of $\mathcal{R}$ the piercing points should be located at the vertices of the location domain. In our setting the

piercing points correspond to the apexes of cones. More precisely, two paired constrained cones (in two different planes) correspond to a vertex in the location domain.

2.1.3 Rectilinear 4-piercing

Now we have to find eight cones $C_i, 1 \leq i \leq 8$ with the following properties:

1. $C_1 \cup C_2 \cup C_3 \cup C_4$ covers P_x.
2. $C_5 \cup C_6 \cup C_7 \cup C_8$ covers P_y.
3. For some pair of cones $C_i, C_j, i \in \{1,2,3,4\}, j \in \{5,6,7,8\}$ the set of all rectangles without those covered by $[C_i] \cap [C_j]$ is 3-pierceable.

As before, assume without loss of generality that C_1, C_4, C_5, C_8 are constrained, so condition (3) when we choose $i \in \{1,4\}$ and $j \in \{5,8\}$ is easily checked in linear time, because we can find the location of C_1, C_4, C_5, C_8 in linear time and then answer the 3-piercing problem in linear time. If $i \in \{2,3\}$ and $j \in \{6,7\}$ then there exist $i' \in \{1,4\}$ and $j' \in \{5,8\}$ such that if the set of rectangles is 4-pierceable then one piercing point must be determined by the cones $C_{i'}$ and $C_{j'}$. Thus, this case is also computed in $O(n)$ time. The more interesting and difficult case is when each constrained cone in one plane corresponds to a non-constrained cone in the other plane. There is a constant number of such pairs, namely eight. Without loss of generality, let C_3 be an unconstrained cone in (c_x, d_x) and C_5 a constrained cone in (c_y, d_y). The analysis for all other such pairs is almost identical.

We sort all the 45° (135°) lines determined by P_x in the (c_x, d_x) plane from left to right, and do the same to the lines determined by P_y in the (c_y, d_y) plane. Clearly, the apex of C_3 is between the apexes of C_1 and C_4. So, we fix the apex of C_3 to coincide with the apex of C_1 and begin to move it (to the right) towards C_4.

The main idea in [99] is similar : they take a point at some vertex of the location domain and begin to move it along the edge of the location domain, at each step maintaining the set of rectangles that is pierced by this point and checking whether the rest of rectangles is 3-pierceable. The checking phase was done using 3-level structure for $p = 4$. Actually, using this structure leads to additional time in total running time of their algorithm. In contrast, in our algorithm, we consider so called *combinations of cones* defined below and use simple data structures, like balanced binary trees. It seems that the same thing can not be done in the original paper of Sharir and Welzl [99] because it becomes quite complex to explain the behavior of piercing points that move on the edges of the location domain.

We define an *event* when a point of P_x is inserted or deleted from C_3. Initially, we compute the set of points $A \subseteq P$ covered by $[C_3] \cap [C_5]$ (when

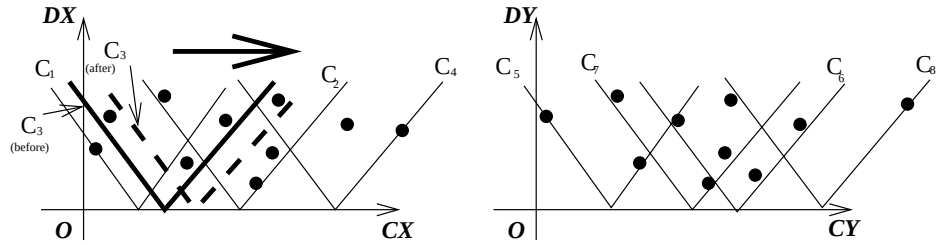

Figure 2.2: Moving the apex of C_3 from the apex of C_1 towards the apex of C_4.

the apex of C_3 is determined by the leftmost 135° line through the points of P_x) and apply the 3-piercing algorithm for the rest of the points $S = P \setminus A$, allowing, only this time, C_1 to move freely. If we have a positive answer, we are done; otherwise we continue.

We next move C_3 to the right, till the next event occurs, and change S accordingly (as in Figure 2.2). The first encountered event as C_3 moves from its initial position is when the leftmost point of P_x is deleted from C_3. We run again the 3-piercing algorithm for S. Here, too, if S is 3-pierceable then we are done. Clearly, from now on the location of the apexes of C_1, C_4 and C_8 will not change during the rest of the algorithm because these cones are defined by the extreme points of P_x and P_y that will never appear in both C_3 and C_5. Let C_7 be the leftmost cone covering S in (c_y, d_y). The location of C_7 will change since C_7 will move towards C_8 and back to cover points. But once C_7 moves back from C_8 it will never move towards C_8 again. This is because C_5 is constrained and C_7, the second cone from the left, moves back to cover points that were removed from A. Since the leftmost point has to be covered in order to have 4-piercing, once C_7 got back to its leftmost position, it will never move to the right again. Thus, the number of changes that we perform on C_7 is $O(n)$. Our goal is to determine the location of the cones C_2 in (c_x, d_x) and C_6 in (c_y, d_y). We will check the possible combinations of pairing the cones to create piercing points. Assuming the cones C_3 and C_5 describe a piercing point, we have the following combinations for the rest of the piercing points:

(a) $(C_1, C_7), (C_2, C_6), (C_4, C_8)$, (b) $(C_1, C_7), (C_2, C_8), (C_4, C_6)$,
(c) $(C_1, C_8), (C_2, C_7), (C_4, C_6)$, (d) $(C_1, C_8), (C_2, C_6), (C_4, C_7)$,
(e) $(C_4, C_7), (C_1, C_6), (C_2, C_8)$, (f) $(C_4, C_8), (C_1, C_6), (C_2, C_7)$.

Observation 2.1.3 *Every solution to the problem belongs to one of the combinations, that is, the combinations of the cones at each step of the 4-piercing algorithm are* **independent**, *meaning that we check 3-pierceability for each fixed combination of the cones throughout all the steps of the 4-piercing algorithm. If we get a negative answer for a combination, we check the other*

combinations. If there is a solution it will be found by the 3*-piercing algorithm for one of the combinations.*

This observation (which has no analogue in [99]) allows us to design an efficient algorithm for our problem. The independency and the finite number of combinations allow to perform the 4-piercing algorithm for each of the combinations separately. For each combination the 4-piercing algorithm is slightly different. Denote by $C_{ij} = [C_i] \cap [C_j] \cap S$. Recall that $S = P \setminus A$, A being the set of points covered by (C_3, C_5). For every combination of cones the following events happen during the algorithm.

The 4-piercing algorithm exemplified by (C_3, C_5)

1. Initially the left boundary point q of C_3 is getting out of C_3. If q was in A then we re-run the 3-piercing algorithm. Otherwise no update is needed, since q did not belong to A and in the current situation nothing was changed.

2. If, when we move the apex of C_3 towards C_4, a point q' is inserted to C_3, we first check if the corresponding point in (c_y, d_y) is covered by C_5. If it is not covered, then we continue moving C_3 to the next event; otherwise we have the following cases:

   2.1 If in a previous stage of the algorithm q' defined the left boundary of the middle cone C_2 in (c_x, d_x), or q' defined the left boundary of the left cone C_7, or q' defined the left boundary of the middle cone C_6 in (c_y, d_y) for the combinations (a)-(d) (similarly, the right boundary for the combinations (e)-(f)), then, for the given combination we perform the following updating scheme: we first check if q' defines the left boundary of C_7. If yes then we have to find, by binary search over S, the new left boundary for C_7 and:

      i. For combination (a). Find the new boundaries of the middle cones C_2 and C_6 in both planes and check whether they cover the rest of the points by computing and examining the set $S^{(1)} = S \setminus C_{17} \setminus C_{48}$. Note that the cones C_4 and C_8 are both constrained and do not move during the whole algorithm. Technically, we compute C_{48} at the beginning of the 4-piercing algorithm. To determine whether C_2 and C_6 can cover $S^{(1)}$ we are only interested in the maxima and minima of the 45° and 135° lines through the points of $S^{(1)}$ in both planes respectively. Note that the total number of updates on C_{48} and on C_7 is at most $O(n)$, thus if we maintain the points of the dynamically changing set $S^{(1)}$ sorted according to the 45° and 135° lines we can update $S^{(1)}$ and find the maxima and minima in both planes by a simple binary search. Consequently, in $O(1)$ time we check whether there exist two cones

C_2 and C_6 with boundaries on these maximal and minimal lines that pass through these points that cover $S^{(1)}$.

ii. For combination (b) (similarly (e)). These combinations are similar in the sense that C_7 (that has $O(n)$ updates) is paired with a constrained cone, either C_1 in (b) or C_4 in (e), and the non constrained cones C_2 and C_6 are each paired with a constrained cone. By computing and examining the set C_{17} (C_{14}) we found the new left boundaries of C_2 and C_6. What is remained is to check whether the pairs (C_2, C_8) and (C_4, C_6) ((C_1, C_6) for combination (e)) cover the set of all points of P which are not already covered by (C_3, C_5) or by (C_1, C_7) ((C_4, C_7) for combination (e)). This can be done as follows. For combination (b) (similarly (e)) compute the set C_{48} (C_{18}) at the beginning of the 4-piercing algorithm. In each step of the 4-piercing algorithm compute the set $S^{(2)} = C_{48} \setminus C_{17}$ ($C_{18} \setminus C_{47}$). Observe the set of all points of P not pierced by (C_3, C_5) and (C_1, C_7) ((C_4, C_7)). They will have to be pierced by (C_2, C_8) and (C_4, C_6) ((C_1, C_6)). Now the points in $S^{(2)}$ should be covered by either C_2 or C_6, whereas the points of $C'_4 = [C_4] \setminus S^{(2)}$ ($C'_1 = [C_1] \setminus S^{(2)}$) must be covered by C_6, and the points of $C'_8 = [C_8] \setminus S^{(2)}$ must be covered by C_2. C_1 and C_7 (C_4 and C_7) determine the left (right) boundary of C_2 and C_6, which are found by a binary search over the points of $S \setminus C_{17}$ ($S \setminus C_{47}$). As for combination (a) the number of updates on C_{48} (C_{18}), and, C_7 is at most $O(n)$. The sets C'_4 (C'_1) and C'_8 are maintained sorted according to the lines throughout the whole algorithm. To check how $S^{(2)}$ is pierced, we maintain two balanced binary trees T_1, T_2. The leaves of $T_1(T_2)$ contain the set $S^{(2)}$ sorted according to the 45° (135°) lines in the plane (c_x, d_x). Let T be T_1 or T_2. Initially, the leaves of T contain the sorted points of C_{48} (C_{18}) in the plane (c_x, d_x). After we compute C_{17} (C_{47}) for the first time we empty the leaves that contain the points that belong to $C_{17}(C_{47})$. Now T contains the sorted lines through the points of $S^{(2)}$. Let p be a point of $S^{(2)}$. A leaf corresponding to p contains the x value of the point of intersection of the 45° (135°) line through p with the c_x axis in (c_x, d_x). It will also contain the y value of the point of intersection of a 45° (135°) line through p with the c_y axis in (c_y, d_y). An inner node $v \in T$ will contain the maximum of the y values corresponding to 135° lines of the leaves of the subtree rooted at v, and the minimum of the 45° lines. During the algorithm we perform a sequence of updates, namely insertions and deletions, in the tree T. When a point q is add to $S^{(2)}$, then we insert it into T in a sorted x-order and update the minimum and maximum y values on the nodes

of path from the leaf q to the root of T. If a point q is deleted from T, then we find the leaf of q, delete it and update the y values of the nodes on the path from the leaf to the root of T. Each update of T takes $O(\log n)$ time. We can check, using the tree T, whether C_2 together with C_6 cover all the points in $S^{(2)}$.

iii. For combination (c) (similarly (f)). At the beginning of the 4-piercing algorithm we compute C_{18}. The cones C_1 and C_8 are constrained and do not move during the whole algorithm. At each step of the 4-piercing algorithm we work with the set $S^{(3)} = S \setminus (C_{18} \cup C_{27})$ and find the leftmost and rightmost points in this set that should be covered in both planes by C_6 and C_4 respectively. We maintain $S^{(3)}$ by incremental updates according to the motion of C_3. Note that the number of updates on C_2 in the whole algorithm is $O(n)$. This is because the left boundary of C_2 is defined by the leftmost point (of S) in (c_x, d_x) not covered by C_{18} and thus C_2 moves towards and back from C_4, but when it moves back it will never move (to the right) again.

iv. For combination (d). We perform a scheme almost identical to that of (c), but with the difference that at each step of the 4-piercing algorithm we work with the set $S^{(4)} = S \setminus (C_{18} \cup C_{47})$ and find the leftmost and rightmost points that should be covered in both planes by C_6 and C_2. Again, we update $S^{(4)}$ at each motion of C_3 in logarithmic time. We find the new boundaries of C_2 and C_6 and check whether C_6 and C_2 cover the leftmost and rightmost points in both plane that we just found.

2.2 If q' does not define a left boundary of a cones as above, then for each combination we perform an identical updating scheme as in 2.1 but without computing a new left boundary of the middle cones C_2 and C_6.

3. If, when we move apex of C_3, a point q'' is deleted from C_3, then

3.1 If $q'' \notin C_5$ we proceed to the next event.

3.2 If in a previous stage of the algorithm q'' defined the left boundary of the cone C_2 in (c_x, d_x), or q'' defined the left boundary of the left cone C_7, or q'' defined the left boundary of the middle cone C_6 in (c_y, d_y) for the combinations (a)-(d) (similarly, right boundary for the combinations (e)-(f)), then, for the given combination we perform the following updating scheme: If q'' defines a new left boundary C_7, then we compute a new location of C_7 and:

i. For combination (a), find the new boundaries of the middle

cones C_2 and C_6 in both planes and compute the rightmost and leftmost points of the set $S^{(1)}$ in both planes.

ii. For combination (b), by examining the set C_{17}, find the new left boundaries of C_2 and C_6, compute the sets $S^{(2)}, C_4'$ and C_8' and update T_1 and T_2.

iii. For combination (c), find the new boundaries of C_2 and C_6. By examining the set $S^{(3)}$ find the leftmost and rightmost points of this set in both planes.

iv. For combination (d), find the new boundaries of C_2 and C_6. By examining the set $S^{(4)}$ we find the leftmost and rightmost points of this set in both planes.

3.3 If q' does not define a left boundary of a cones as above, then for each combination we perform an identical updating scheme as in 3.2 but without computing a new left boundary of the middle cones C_2 and C_6. Notice that in this case (when q'' is deleted from C_3) the 4-piercing of P is not possible, because it wasn't possible in previous step of the algorithm.

Thus,

Theorem 2.1.4 *We can determine whether a set of n axis-parallel rectangles is 4-pierceable or not in $O(n \log n)$ time, and give a solution (if it exists) in the same running time.*

Comparing our algorithm to that in [99]. Our main observation is the "independency" of the cone combinations which has no analogue in [99]. In addition, our data structure is a balanced binary tree as opposed to the 3-level structure of [99]. Actually, each level of the 3-level structure adds additional $\log n$ factor to the running time of the algorithm in [99].

2.1.4 Rectilinear 5-piercing

Now we have to find ten cones $C_i, 1 \leq i \leq 10$ with the following properties:

1. $C_1 \cup C_2 \cup C_3 \cup C_4 \cup C_5$ covers P_x.
2. $C_6 \cup C_7 \cup C_8 \cup C_9 \cup C_{10}$ covers P_y.
3. For some pair of cones $C_i, C_j, i \in \{1,2,3,4,5\}, j \in \{6,7,8,9,10\}$ the set of all rectangles without those covered by $[C_i] \cap [C_j]$ is 4-pierceable.

Due to the duality relation between our analysis and that in [99] we follow the case analysis in [99]. Assume, without loss of generality, that C_1, C_5, C_6, C_{10} are constrained and the order of the cones is from left to right. We may also assume that one of the following situations occurs:

(a) There is one pair of constrained cones C_i, C_j, $i \in \{1,5\}$ and $j \in \{6,10\}$. We try all of these possibilities, find the set of rectangles not covered by the given pair of cones, and test whether this set is 4-pierceable, using the preceding algorithm. This takes $O(n \log n)$ time.

(b) Every constrained cone is paired with a non-constrained cone. Since there are four constrained cones there are two pairs with the same constrained cone. We proceed as follows. First, we guess a *unique* constrained cone, say C_1, which is paired with a non-constrained, say C_7. Then we proceed as in the 4-piercing algorithm, i.e. slide C_7 from left to right, starting at the apex of C_6 and stopping when we reach the apex of C_{10}. In each move, we check whether the set of the rest of the rectangles is 4-pierceable using the following observation by Sharir and Welzl [99]. They observe that the 4-piercing problem has always a pair of two constrained cones in its solution. In our case they are either C_2 and C_6, or C_2 and C_{10} (C_2 becomes constrained after computing $S \setminus C_{17}$). We process with each of these cases separately. Assume, without loss of generality, that we process C_2 and C_6. Then at each move of C_7 we update C_{26} and check whether the rest of rectangles is 3-pierceable as in the update step in the 4-piercing algorithm. Omitting some details, we obtain a procedure that runs in $O(n \log n)$ time.

(c) There a pair of two unconstrained cones. Assume, without loss of generality, the cones are C_4 and C_8. We also assume without loss of generality, that we have paired C_6 and C_3, C_{10} and C_2, C_1 and C_7, C_5 and C_9 (a constrained cone with an unconstrained cone). Then, as was observed in [99], either at least one of the c_x-coordinates of the apexes of C_2 and C_3 is smaller then the the c_x-coordinate of the apex of C_4, or at least one of the c_x-coordinates of the apexes of C_2 and C_3 is larger than the c_x-coordinate of the apex of C_4. Suppose one of them is smaller than C_4. Then we slide C_7 (that is paired with C_1) from left to right, starting at the apex of C_6 and stopping when we reach the apex of C_{10}. At each event, we check whether a set of the rest of rectangles is 4-pierceable. As was claimed in [99] at each move of C_7, the cone C_2 is paired either with C_6 or with C_{10}. Thus we have a situation identical to case (b). This can be computed as in case (b) above, implying that case (c) can also be computed in $O(n \log n)$ time. Hence we obtain:

Theorem 2.1.5 *We can determine whether a set of n axis-parallel rectangles is 5-pierceable or not in $O(n \log n)$ time, and give a solution (if it exists) in the same running time.*

The result for 5-piercing can be applied to find a better solution for the rectilinear 5-center problem with $O(n \log^2 n)$ running time compared to one in [99].

2.1.5 Extending to high dimensional space and to $p > 4$

Our technique immediately implies a linear time algorithm for 2-pierceability of a set of axis-parallel rectangles for arbitrary (fixed) dimension $d, d \geq 2$ (there are only constrained cones) and an $O(n \log n)$ time algorithm for 3-pierceability of a set of axis-parallel rectangles for dimension d, $3 \leq d \leq 5$ (the same result was obtained by [14] independently). In the latter problem there is always a combination where $d-1$ cones are constrained and (at most) one is a non-constrained cone. At each step of the algorithm there is a finite number of the d-coupling combinations of the cones.

Returning to the planar p-piercing problem we apply an algorithm similar to the 4-piercing algorithm. Using our approach we obtain an efficient algorithm for general (but fixed) $p \geq 6$ improving [99]. The general observation is that a constrained cone is always paired with a constrained or an unconstrained cone. Thus for solving a $(p+1)$-piercing problem we have to consider the two cases. In the first case there are two constrained cones paired together, we can determine the rest of the (uncovered) boxes in linear time and apply the p-piercing algorithm for the rest of the boxes. In the second case, a constrained cone is paired with a non-constrained one. We move the apex of the non-constrained cone between the apexes of the constrained cones in its plane. Thus we have $O(n)$ steps (when a point is either inserted or deleted from the non-constrained cone). In each step we run the $(p-1)$-piercing algorithm for the rest of the points. Thus our algorithm for general, but fixed $p \geq 6$ in the plane runs in time $O(n^{p-4} \log n)$, while the algorithm in [99] in time $O(n^{p-4} \log^5 n)$.

Conclusions

There is some duality between the analysis of [99] and ours. A pair of constrained cone with nonconstrained cone in our algorithms corresponds to an edge on the boundary of the location domain in [99], and two paired constrained cones in our algorithms correspond to a corner in the location domain of [99]. We are looking into applying a similar technique for sets of triangles, rhombi, etc. Recently, Nussbaum [86] and Makris and Tsakalidis [76] present a algorithm with a similar runtimes for a various piercing problems. Still, the most intriguing question is whether we can improve the running time of algorithm for p-piercing problems where $p > 5$. We hope that our approach can help in obtaining a better solution to these problems.

2.2 Two-Covering (p4)

Given a set S of n points in d-dimensional space, $d \geq 2$, find two axis-parallel boxes b_1 and b_2 that together cover the set S and minimize the maximum of measures $\mu(b_1)$ and $\mu(b_2)$, where μ is a monotone function of the box, i.e. $b_1 \subseteq b_2$ implies $\mu(b_1) \leq \mu(b_2)$. Examples of the box measure μ are the volume of the box, the perimeter of the box (in higher dimensions it can be defined as the sum of 1-dimensional edges or as the area of the boundary of the box), the length of the diagonal etc. We assume that the dimension d is fixed and the measure of the box can be computed in $O(1)$ time. For simplicity we consider the general case of the distinct coordinates, i.e. the projection of S onto any coordinate axis is a set of n distinct points. Initially we sort all the points of S according to each of the coordinates.

Given a set of points S, the *bounding box* of S, denoted by $bb(S)$, is the smallest axis-parallel box that contains S. The bounding box of S is determined by $2d$ points, two from each axis i, $i = 1, \ldots, d$: the leftmost point $l_i(S)$ of S, and the rightmost point $r_i(S)$. We call these points the *determinators* of S. For a box $b = [l_1, r_1] \times \ldots \times [l_d, r_d]$, the points l_i and r_i are also called the *determinators of* b. For a point p, let $x(p)$ and $y(p)$ denote the coordinates of p in the first and the second axes, respectively.

2.2.1 The algorithm for the plane

Glozman et al. [57] consider three different ways to partition the determinators between two subsets of S

1. One of the subsets gets three determinators and the other gets one determinator.
2. Each subset gets two determinators lying on opposite sides of $bb(S)$.
3. Each subset gets two determinators lying on adjacent sides of $bb(S)$.

We distinguish only 2 cases:

1. One subset gets two determinators lying on adjacent sides of $bb(S)$.
2. Each subset gets two determinators lying on opposite sides of $bb(S)$.

The algorithm finds the solutions for both cases and then returns the pair with minimum measure μ. The algorithm for the first case is essentially the same as the algorithm for the problem **Measure2** which is described in Section 2.2.2 (in the problem **Measure2** one subset gets the determinators of S from two different axes). Thus, it is sufficient to explain the algorithm for the second case.

Consider the box, say b_1, that contains the uppermost and bottommost points of S. The second box b_2 contains the set S_2 of the points outside of b_1. We can assume that b_2 is the bounding box of these points, i.e. $b_2 = bb(S \backslash b_1)$. In other words, two determinators (left and right) of the box b_1 define both boxes. Let l and r be the left and right determinators of the box b_1. They can be selected among the x-coordinates of the points of S. The naive approach is to compute the measures of the boxes b_1 and b_2 for each pair of x-coordinates. It takes $\Omega(n^2)$ time in the worst case. We can reduce the number of pairs l and r that are candidates for the solution. Let $L = (p_1, \ldots, p_n)$ be the list of points S sorted by their x-coordinate. For each left determinator $l' = x(p_i)$ of b_1, the algorithm computes only two candidates for the right determinator of b_1 : c_1 and c_2. The value c_1 is the largest x-coordinate such that the box b_1 with the right determinator $r = c_1$ has the measure not greater than the box b_2. The value c_2 is the smallest x-coordinate (if any) such that the box b_1 with the right determinator $r = c_2$ has the measure greater than the box b_2. Clearly, $c_1 < c_2$ and c_2 is the next element of L list after c_1 if it exists. If we will think about c_1 and c_2 as functions that depend on the choice of l', we can see that c_1 and c_2 are both monotonic, i.e. $c_1(b) \geq c_1(a)$ and $c_2(b) \geq c_2(a)$, if $b > a$.

The monotonicity of c_1 and c_2 allows us to find them using at most $4n - 2$ computations of the measure μ. To find c_1 and c_2 for $l' = x(p_i)$ the algorithm starts with the value $r' = c_2(x(p_{i-1}))$ that plays role of the right determinator of b_1. If the box b_1 with the determinators l' and r' has a measure greater than the measure of the corresponding box b_2, then c_1 and c_2 coincide with the previous values. Otherwise, we do the following. Let A be a matrix with n rows and n columns. Its element $A(i, j)$ is equal to 1 if the algorithm computes the measure of the box b_1 with the determinators p_i and p_j; otherwise $A(i, j) = 0$. The unit elements of A correspond to the path from $(1, 1)$ to (n_0, n) where $p_n = c_2(x(p_{n_0}))$ or $n = n_0$ (if $c_2(x(p_{n_0}))$ does not exist). The value of r' is changed accordingly to the unit elements in A. On each step we move in left to right or top to down direction on the matrix A. The algorithm visits at most $2n - 1$ cells. At each cell it computes the measures of the boxes b_1 and b_2. Thus, it computes at most $4n - 2$ boxes. Below we give the formal description of the algorithm.

It remains to show how to compute the bounding box of S_2. The set S_2 undergoes both insertions and deletions of the points throughout the algorithm, but each point can be deleted from and inserted to S_2 only once. Initially $S_2 = S$. We partition S_2 into two subsets S_2' and S_2'' as defined below:

- S_2' contains the points that were not deleted from S_2
- $S_2'' = S_2 \setminus S_2'$.

Actually S_2' (S_2'') has the points to the right (resp. left) of the box containing points of S_1 but the definition above can also be used for higher dimensions. The set S_2' is updated only by deletions of points. The set

S_2'' is updated only by insertions of points. The box $bb(S_2'')$ can be easily maintained during the algorithm. Moreover, using a presorting step we can maintain the box $bb(S_2')$ in $O(1)$ time. The box $bb(S_2)$ can be computed in $O(1)$ time using the computed boxes $bb(S_2')$ and $bb(S_2'')$.

Algorithm TwoBoxes ($*$ Given points $p_1, \ldots, p_n$ sorted by x-coordinate. Find the minimum of maximum measure of two boxes covering points such that one box gets the uppermost $y_{\max}$ and bottommost $y_{\min}$ determinators $*$)
$j := 1$
$S_2 := S$
solution:= $\mu_2 := \mu(bb(S_2))$
for $i := 1$ to n **do**
  $\mu_1 = \mu([p_i, p_j] \times [y_{\min}, y_{\max}])$
  if solution$<$ $\min(\mu_1, \mu_2)$ **then** solution:= $\min(\mu_1, \mu_2)$
  while $\mu_1 \leq \mu_2$ **do**
    if $j \geq n$
    then return solution
    $S_2 := S_2 \setminus \{p_j\}$
    $\mu_2 := \mu(bb(S_2))$
    $j := j + 1$
    $\mu_1 = \mu([p_i, p_j] \times [y_{\min}, y_{\max}])$
    if solution$<$ $\min(\mu_1, \mu_2)$ **then** solution:= $\min(\mu_1, \mu_2)$
  $S_2 := S_2 \cup \{p_j\}$
  $\mu_2 := \mu(bb(S_2))$
return solution

Clearly, the algorithm described above takes only linear time. We conclude by theorem.

Theorem 2.2.1 *The min-max two box problem for n presorted points in the plane can be solved in $O(n)$ time and $O(n)$ space.*

2.2.2 The algorithm in higher dimensions

The main idea of the algorithm is the reduction of the dimension. We assume that the dimension d is greater than two and reduce it to two. Let the boxes b_1 and b_2 be the solution of the min-max two box problem. Then there is a box which has at least d determinators coinciding with the corresponding determinators of S. Assume, without loss of generality, that b_1 is such a box. The box b_1 defines the smallest box $bb(S \setminus b_1)$ containing all points outside b_1. For simplicity we assume that the box b_2 is equal to $bb(S \setminus b_1)$ throughout this Section. The boxes b_1 and b_2 are the solution of the following problem.

Problem Measure1. Given a set S of n points and d determinators of the box b_1, find the remaining d determinators of b_1 such that the expression $\max(\mu(b_1), \mu(b_2))$ is minimized, where b_2 is equal to $bb(S \setminus b_1)$.

For a set S, there are $\binom{2d}{d}$ ways to fix d determinators. So, we have $\binom{2d}{d}$ problems **Measure1**. We solve all these problems. The solution of the min-max two box problem can be chosen from the $\binom{2d}{d}$ pairs of boxes b_1 and b_2. (Of course we do not need to store all these pairs.)

Now we show how to solve the problem **Measure1**. The box b_1 has d fixed determinators and d free determinators. Since the dimension d is greater than 2, there are two free determinators on the different axes. Assume, without loss of generality, that they are right determinators on x and y axes, i.e. $r_1(b_1)$ and $r_2(b_1)$. We want to find these determinators and the remaining $d-2$ free determinators. At first step the algorithm defines the remaining determinators. They can be determined by the points of S (for example, the i-th left determinator $l_i(b_1)$ is determined by the point $q = (q_1, \ldots, q_d) \in S$ if $l_i(b_1) = q_i$). Let us consider all the $(d-2)$-tuples of the points of S. The algorithm goes through all these tuples and finds the combination that attains the required minimum.

It is clear that the number of tuples is n^{d-2}. Note that some tuples cannot give the solution since the determinators of b_1 have to satisfy the inequality $l_i < r_i, i = 1, \ldots, d$.

The 2-dimensional problem can be now formulated as

Problem Measure2. Given a set S of n points and $2d-2$ determinators $l_1, \ldots, l_d$, $r_3, \ldots, r_d$ of the box b_1. Find two determinators r_1 and r_2 of the box b_1 in order to minimize the expression $\max(\mu(b_1), \mu(bb(S \setminus b_1)))$.

We show that the problem **Measure2** can be solved in $O(n)$ time. The determinators r_1 and r_2 can be chosen from the sets $\{x(p_1), \ldots, x(p_n)\}$ and $\{y(p_1), \ldots, y(p_n)\}$ respectively. Let $b(x, y)$ denote the box $[l_1, x] \times [l_2, y] \times [l_3, r_3] \times \ldots \times [l_d, r_d]$, where $x \in \{x(p_1), \ldots, x(p_n)\}$ and $y \in \{y(p_1), \ldots, y(p_n)\}$. Let S_1 denote the set of points of S in the box $b(x, y)$ and $S_2 = S \setminus S_1$. For each $x \in \{x(p_1), \ldots, x(p_n)\}$, our algorithm finds the largest $y \in \{y(p_1), \ldots, y(p_n)\}$ such that $\mu(b(x, y)) \leq \mu(bb(S \setminus b(x, y)))$. Denote it by $Y_1(x)$. We observe that $Y_1(x)$ is *monotone*.

Observation 2.2.2 *$Y_1(x)$ is non-increasing function.*

Proof. Consider two x-coordinates $x' < x''$. It is clear that the box $B = b(x'', Y_1(x''))$ contains the box $C = b(x', Y_1(x''))$. Therefore the box $B' = bb(S \setminus B)$ is contained in the box $C' = bb(S \setminus C)$. This implies $\mu(B) \geq \mu(C)$ and $\mu(B') \leq \mu(C')$. $\mu(B) \leq \mu(B')$ by the definition Y_1. Hence $\mu(C) \leq \mu(B) \leq \mu(B') \leq \mu(C')$. It means that $Y_1(x'') \leq Y_1(x')$. ∎

For each $x \in \{x(p_1), \ldots, x(p_n)\}$ our algorithm (for the problem **Measure2**) finds the smallest $y \in \{y(p_1), \ldots, y(p_n), \infty\}$ such that $\mu(b(x, y)) > \mu(bb(S \setminus b(x, y)))$ (if it exists). Denote it by $Y_2(x)$.

Observation 2.2.3 *There exists a solution of problem* **Measure2** *such that $r_2 = Y_1(r_1)$ or $r_2 = Y_2(r_1)$.*

Proof. Let the boxes $b_1 = b(r_1, r_2)$ and $b_2 = bb(S \setminus b_1)$ be the solution of the problem **Measure2**. If $\mu(b_1) \leq \mu(b_2)$ then $r_2 \leq Y_1(r_1)$. Using arguments like one of Observation 2.2.2 we can show

$$\mu(b_1) \leq \mu(b(r_1, Y_1(r_1))) \leq \mu(bb(S \setminus b(r_1, Y_1(r_1)))) \leq \mu(b_2).$$

We can enlarge the box b_1 to the box $b(r_1, Y_1(r_1))$. This gives a solution with $r_2 = Y_1(r_1)$.

In the case $\mu(b_1) > \mu(b_2)$ we can take $r_2 = Y_2(r_1)$. ∎

For each $x \in \{x(p_1), \ldots, x(p_n)\}$ and $y \in \{Y_1(x), Y_2(x)\}$, we compute $\max(\mu(b(x,y)), \mu(bb(S \setminus b(x,y))))$. Then we compute the minimum value of these numbers. It gives the solution of the problem **Measure2**.

Now we explain how to achieve $O(n)$ running time. Let b_1^*, b_2^* be the pair of boxes that are candidates for the solution. Let us consider the moment when we have computed $Y_1(x)$ and $Y_2(x)$ for some x. Using the fact that the points of S are sorted separately in each of the coordinate axes, we get the next value $x' > x$ in this order. For the point p of S with x-coordinate x', we perform the following operations.

First we check whether p can lie in b_1. If p cannot lie in b_1 (the i-th coordinate of p is less than l_i for some i, or the i-th coordinate of p is greater than r_i, for some $i > 2$) then p remains in S_2 and we pass it. Otherwise we compare $y(p)$ and $Y_1(x)$. If $y(p) > Y_1(x)$ then p remains in S_2 (by monotonicity of Y_1) and we pass it. Else we delete p from S_2 and insert it into S_1. For the determinators $r_1 = x'$ and $r_2 = Y_1(x)$ of the box b_1, compute $\mu_1 = \mu(b_1)$ and $\mu_2 = \mu(bb(S_2))$. If $\max(\mu(b_1), \mu(b_2)) < \max(\mu(b_1^*), \mu(b_2^*))$, then set $b_1^* = b_1$ and $b_2^* = b_2$. There are two possible cases.

Case 1: $\mu_1 \leq \mu_2$. In this case $Y_1(x') = Y_1(x)$ and $Y_2(x') = Y_2(x)$ by monotonicity of Y_1. It should be noted that we do not need to compute the rectangular measure μ for the pair x' and $Y_2(x)$ since it is greater than or equal to the measure of the boxes b_1 and b_2 determined by the pair x and $Y_2(x)$. We only have to compute the rectangular measure $\max(\mu_1, \mu_2)$ for the pair x' and $Y_1(x)$ and update the current solution if its measure is greater than $\max(\mu_1, \mu_2)$.

Case 2: $\mu_1 > \mu_2$. In this case $Y_1(x') < Y_1(x)$ and $Y_2(x') < Y_2(x)$. We remove a few points from the box b_1 in decreasing order of y-coordinate to achieve $\mu_1 \leq \mu_2$. Let $y = Y_1(x)$. In order to find $Y_1(x')$ and $Y_2(x')$ we perform the following operations as long as $\mu_1 > \mu_2$.

- For each point $p \in S_1$ with $y(p) = y$, move the point p from S_1 to S_2.
- Using the sorted order of S according to the y-coordinate we decrease y to the next point whose y-coordinate is less than current y.
- Compute $\mu_1 = \mu(b(x', y))$ and $\mu_2 = \mu(bb(S_2))$. If $\max(\mu(b_1), \mu(b_2)) < \max(\mu(b_1^*), \mu(b_2^*))$, then set $b_1^* = b_1$ and $b_2^* = b_2$.

After these operations we have $Y_1(x') = y$ and $Y_2(x')$ is the previous value of y. Now the processing of x' is finished.

We start the algorithm at the point x that is less than the x-coordinate of all the points of S, for example $x = \min\{x(p_1), \ldots, x(p_n)\} - 1$. We set $Y_1(x) = \min\{y(p_1), \ldots, y(p_n)\}$. Initially we have $S_1 = \emptyset$, $\mu_1 = 0$, $S_2 = S$ and $\mu_2 = \mu(bb(S))$, Y_2 can have any value because it is used only after S_1 becomes non-empty.

Ignoring the time spent on computing μ_1 and μ_2 the algorithm takes $O(n)$ time. It remains to show how to compute μ_2. Recall $\mu_2 = \mu(bb(S_2))$. The maintenance of the bounding box of S_2 is described in Section 2.2.1.

We summarize by the following theorem.

Theorem 2.2.4 *The min-max two box problem in d-dimensional space, $d \geq 3$, can be solved in time $O(n \log n + n^{d-1})$ using $O(n)$ space.*

Proof. We spend $O(n \log n)$ time in sorting S according to all its coordinates. As it was pointed above, we solve $\binom{2d}{d}$ problems **Measure1** in order to obtain a solution for the min-max two box problem. Each problem **Measure1** is solved by considering all the $(d-2)$-tuples of points S. There are n^{d-2} such tuples. For each such a tuple related to the problem **Measure2** we spend $O(n)$ time. ∎

Conclusions

We present an efficient algorithm for solving min-max two box problem. The efficiency of the algorithm is based on the monotonicity of the evaluated function in the problem. It would be interesting to find some connection between this problem and the problems considered by Jaromczyk and Kowaluk [64] and Segal [69]. It is still an open question to prove some nontrivial lower bounds for the problems appeared in [64, 69] and this section.

2.3 Center Problems

In order to solve Problems **p5, p6, p7** we employ a variety of techniques to solve these optimization problems. The decision algorithm of Problem **p5** searches for the centers of a solution pair (of squares) in an implicit special matrix, using a technique that has recently been used in [38, 97]. To find an optimal solution, a search in a collection of sorted matrices [54] is performed.

For the second algorithm for Problem **p5** we present an implicit use of Frederickson and Johnson technique of sorted matrices [54] i.e. we embed this technique into the decision algorithm in order to speed up the running time. This is crucial for the dynamic version of our algorithm, because standard use of this technique may lead to additional factors of $O(n)$, in the case of squares, and $O(n^2)$, in the case of rectangles, to the running time. We obtain an $O(\max(n \log n,\ m \log n(\log n + \log m)))$ runtime algorithm for the squares case and an $O(mn \log m \log n)$ runtime algorithm for the rectangle case. As for the dynamic versions the runtimes for update operations for both algorithms are polylogarithmic in n for any values of m.

The decision algorithm of Problem **p6** involves maintenance of dynamically changing convex hulls, and maintenance of an orthogonal range search tree that must adapt to a rotating axes system. For the optimization, we apply Megiddo's [79] parametric search. However, since our decision algorithm is not parallelizable, we had to find an algorithm that solves a completely different problem, but is both parallelizable and enables to generate the optimal square size when the parametric search technique is applied to it.

In Problem **p7** we describe the sizes of candidate solution squares as a collection of curves. For a dynamically changing set of such curves, we transform the problem of determining whether their upper envelope has a point below some horizontal line, into the problem of stabbing a dynamically changing set of segments. The latter problem is solved using a (dynamic) segment tree.

2.3.1 Two constrained axis-parallel squares (p5)

The first algorithm

We are given a set S of n points in the plane, and wish to find two axis-parallel squares, centered at points of S, whose union covers (contains) S, such that the area of the larger square is minimal. We first transform the corresponding decision problem into a constrained 2-piercing problem, which we solve in $O(n \log n)$ time. We then apply the algorithm of Frederickson and Johnson [54] to find an optimal solution.

The decision algorithm

The decision problem is stated as follows: Given a set S of n points, are there two constrained axis-parallel squares, each of a given area $\mathcal{A}$, whose union covers S. We present an $O(n \log n)$ algorithm for solving the decision problem.

We adopt the notation of [99] (see also [71, 94] and section 2.1). Denote by $\mathcal{R}$ the set of axis-parallel squares of area $\mathcal{A}$ centered at the points of S. Recall that $\mathcal{R}$ is *p-pierceable* if there exists a set $\mathcal{X}$ of p points which intersects each of the squares in $\mathcal{R}$. The set $\mathcal{X}$ is called a *piercing set* for $\mathcal{R}$. Notice that $\mathcal{X}$ is a piercing set for $\mathcal{R}$ if and only if the union of the axis-parallel squares of area $\mathcal{A}$ centered at the points of $\mathcal{X}$ covers S. $\mathcal{R}$ is *p-constrained pierceable* if there exists a piercing set of p points which is contained in S. Thus, solving the decision problem is equivalent to determining whether $\mathcal{R}$ is 2-constrained pierceable.

We first compute the rectangle $R = \cap\mathcal{R}$. If R is not empty then $\mathcal{R}$ is 1-pierceable, and we check whether it is also 1-constrained pierceable by checking whether S has a point in R. If $\mathcal{R}$ is 1-constrained pierceable then we are done, so assume that it is not. If $\mathcal{R}$ was not found to be 1-pierceable, then we apply the linear time algorithm from section 2.1.2 (see also [99, 37]) to check whether $\mathcal{R}$ is 2-pierceable. If $\mathcal{R}$ is neither 1-pierceable nor 2-pierceable, then obviously $\mathcal{R}$ is not 2-constrained pierceable and we are done. Assume therefore that $\mathcal{R}$ is 2-pierceable (or 1-pierceable).

Assume $\mathcal{R}$ is 2-constrained pierceable, and let $p_1, p_2 \in S$ be a pair of piercing points for $\mathcal{R}$. We assume that p_1 lies to the left of and below p_2. (The case where p_1 lies to the left of and above p_2 is treated analogously.) We next show that $\mathcal{R}$ can be divided into two subsets $\mathcal{R}_1, \mathcal{R}_2$, such that (i) $p_1 \in \cap\mathcal{R}_1$, $p_2 \in \cap\mathcal{R}_2$, and (ii) $\mathcal{R}_1$ (alternatively $\mathcal{R}_2$) can be represented in a way that will assist us in the search for p_1 and p_2.

Denote by $X_{\mathcal{R}}$ the centers of the squares in $\mathcal{R}$ (the points in P) sorted by their x-coordinate (left to right), and by $Y_{\mathcal{R}}$ the centers of the squares in $\mathcal{R}$ sorted by their y-coordinate (low to high). We now claim:

Claim 2.3.1 *If p_1 and p_2 are as above, then $\mathcal{R}$ can be divided into two subsets $\mathcal{R}_1$ and $\mathcal{R}_2$, $p_1 \in \cap\mathcal{R}_1$, $p_2 \in \cap\mathcal{R}_2$, such that $\mathcal{R}_1$ can be represented as the union of two subsets $\mathcal{R}_1^x$ and $\mathcal{R}_1^y$ (not necessarily disjoint, and one of them might be empty), where the centers of squares of $\mathcal{R}_1^x$ form a consecutive subsequence of the list $X_{\mathcal{R}}$, starting from its beginning, and the centers of squares of $\mathcal{R}_1^y$ form a consecutive subsequence of $Y_{\mathcal{R}}$, starting from the list's beginning.*

Proof. We prove by constructing the sets $\mathcal{R}_1^x$ and $\mathcal{R}_1^y$, and then putting $\mathcal{R}_1 = \mathcal{R}_1^x \cup \mathcal{R}_1^y$ and $\mathcal{R}_2 = \mathcal{R} - \mathcal{R}_1$. We next show that indeed $p_1 \in \cap\mathcal{R}_1$ and $p_2 \in \cap\mathcal{R}_2$.

We consider the centers in $Y_{\mathcal{R}}$, one by one, in increasing order, until a center is encountered whose corresponding square A is not pierced by p_1. $\mathcal{R}_1^y$ consists of all squares in $Y_{\mathcal{R}}$ below A (i.e., preceding A in $Y_{\mathcal{R}}$). A might be the first square in $Y_{\mathcal{R}}$, in which case $\mathcal{R}_1^y$ is empty. We now find the location of the x-coordinate of the center of A in $X_{\mathcal{R}}$, and start moving from this point leftwards, i.e., in decreasing order. Thus moving, we either encounter a square, call it B, that is **higher** than A and is not pierced by p_2, or we do not.

If we do not encounter such a square B (which is clearly the case if the bottom edge of A lies above p_1), then put $\mathcal{R}_1^x = \emptyset$, otherwise $\mathcal{R}_1^x$ consists of all squares in $X_{\mathcal{R}}$ to the left of B including B.

It remains to show that $p_1 \in \cap\mathcal{R}_1$ and that $p_2 \in \cap\mathcal{R}_2$. We assume that the square B exists, which is the slightly more difficult case. We first show the former assertion, i.e., $p_1 \in \cap\mathcal{R}_1$. The fact that p_2 is not in B implies that p_2 lies to the right of the right edge of B, because B cannot lie below p_2 since it is higher than A which is already pierced by p_2. Therefore none of the squares in $\mathcal{R}_1^x$ is pierced by p_2 thus $p_1 \in \cap\mathcal{R}_1^x$. By our construction, $p_1 \in \cap\mathcal{R}_1^y$, so together we have $p_1 \in \cap\mathcal{R}_1$. Now consider a square $C \in \mathcal{R}_2$, $C \neq A$. C is higher than A, because it is not in $\mathcal{R}_1^y$. Therefore if C is not pierced by p_2, then C must lie to the left of A. But if so, it is in $\mathcal{R}_1^x$ and thus not in $\mathcal{R}_2$. ∎

The claim above reveals a monotonicity property that allows us to design an efficient algorithm for the decision problem. We employ a technique, due to Sharir [97], that resembles searching in monotone matrices; for a recent application and refinement of this technique, see [38]. Let M be an $n \times n$ matrix whose rows correspond to $X_{\mathcal{R}}$ and whose columns correspond to $Y_{\mathcal{R}}$. An entry M_{xy} in the matrix is defined as follows. Let D_x be the set of squares in $\mathcal{R}$ such that the x-coordinate of their centers is smaller or equal to x, and let D_y be the set of squares in $\mathcal{R}$ such that the y-coordinate of their centers is smaller or equal to y. Let $D_{xy}^l = D_x \cup D_y$ and $D_{xy}^r = (\mathcal{R} - D_{xy}^l)$.

$$M_{xy} = \begin{cases} `YY' & \text{if both } D_{xy}^r \text{ and } D_{xy}^l \text{ are 1-constrained pierceable} \\ `YN' & \text{if } D_{xy}^r \text{ is 1-constrained pierceable but } D_{xy}^l \text{ is not} \\ `NY' & \text{if } D_{xy}^r \text{ is not 1-constrained pierceable but } D_{xy}^l \text{ is} \\ `NN' & \text{if neither } D_{xy}^r \text{ nor } D_{xy}^l \text{ is 1-constrained pierceable} \end{cases}$$

Sharir's technique enables us to determine whether M contains an entry of the form 'YY' without having to construct the entire matrix. In order to apply his technique the lines and columns of M^1 must be non-decreasing (assuming 'Y' > 'N'), and the lines and columns of M^2 must be non-increasing, where M^i is the matrix obtained from M by picking from each entry only the i'th letter, $i = 1, 2$. In our case this property clearly holds, since, for example, if for some x_0 and y_0, $M^1_{x_0,y_0}$ ='Y', then for any $x' \geq x_0$ and $y' \geq y_0$, $M^1_{x',y'}$='Y'. Thus we can determine whether M contains an entry 'YY' by

inspecting only $O(n)$ entries in M, advancing along a *connected* path within M [38]. For each entry along this path, we need to determine whether D^z_{xy} is 1-constrained pierceable, $z \in \{l, r\}$. This can be done easily in $O(\log n)$ time by maintaining dynamically the intersection $\cap D^z_{xy}$, and utilizing a standard orthogonal range searching data structure of size $O(n \log n)$ [20]. Thus in $O(n \log n)$ time we can determine whether M contains a 'YY' entry.

Theorem 2.3.2 *Given a set S of n input points and area $\mathcal{A}$, one can find two constrained axis-parallel squares of area $\mathcal{A}$ each that cover S in time $O(n \log n)$ using $O(n \log n)$ space.*

We have just found whether a set of equal-sized squares is 2-pierceable by two of their centers. For the optimization, we shrink these squares as much as possible, so that they remain 2-constrained pierceable.

Optimization

For solving the optimization problem we observe that each L_∞ distance (multiplied by 2 and squared) can be a potential area solution. We can represent all L_∞ distances as in [57] by sorted matrices. We sort all the points of P in x and y directions. Entry (i, j) in the matrix M_1 stores the value $4(x_j - x_i)^2$, where x_i, x_j are the x-coordinates of the points with indices i, j in the sorted x-order, and, similarly, entry (i, j) in the matrix M_2 stores the value $4(y_j - y_i)^2$, where y_i, y_j are the y-coordinates of the points with indices i, j in the sorted y-order. We then apply the Frederickson and Johnson algorithm [54] to M_1 and M_2 and obtain the smallest value in the matrices for which the decision algorithm answers "Yes" and thus obtain the optimal solution. We have shown:

Theorem 2.3.3 *Given a set S of n input points, one can find two constrained axis-parallel squares that cover all the input points such that the size of the larger square is minimized in $O(n \log^2 n)$ time using $O(n \log n)$ space.*

The second algorithm

We solve here a more general problem which is defined as follows. Given a set S of n demand points and a set C of m center points, find two axis-parallel squares (or rectangles) that cover all the points of S and centered at the points of C such that that size of largest square (rectangle) is minimized.

Recall the notations from the algorithm for Problem **p4** from section 2.2. Given a set of points S, the *bounding box* of S, denoted by $B(S)$, is the smallest axis-parallel rectangle that contains S. The bounding box of S is determined by the four points, two from each axis : leftmost (smallest coordinate) and rightmost (largest coordinate) points in each of the axes,

which we denote by l_x, l_y, r_x, r_y. We call these points the *determinators* of $B(S)$. Denote by X_S (Y_S) the sorted list of the points in S according to x (y) axis.

The decision algorithm

Let s_1 be a square of area $\mathcal{A}$. In the decision algorithm we go over all the points of C as a center of s_1. At each step check whether we can cover the rest of the points of S (which are not covered by s_1) by a second constrained square s_2 of size $\mathcal{A}$. Denote by K the set of points which is not covered by s_1. Denote by s_{v_1} and s_{v_2} two vertical lines that go through the left and right side of s_1, respectively. Similarly, s_{h_1} and s_{h_2} are two horizontal lines that go through the bottom and the top sides of s_1, respectively. For $s_{v_1}(s_{v_2})$ we compute (by a binary search) the nearest point p (q) in X_S from the left (right) of $s_{v_1}(s_{v_2})$. For $s_{h_1}(s_{h_2})$ we compute the nearest point p' (q') in Y_S that is below (above) of $s_{h_1}(s_{h_2})$.

Let S_i^l (S_j^r) be the set that contains all the points of S with the x-coordinate that less or equal (equal or larger) to the x-coordinate of ith point (jth point) in the list X_S. Similarly, let S_k^b (S_m^t) be the set that contains all the points of S with the y-coordinate that less or equal (equal or larger) to the y-coordinate of kth point (mth point) in the list Y_S.

Observation 2.3.4 *The determinators of $B(K)$ are defined by the determinators of $B(S_i^l)$, $B(S_j^r)$, $B(S_k^b)$, $B(S_m^t)$. More precisely, the determinators of $B(K)$ are the leftmost, rightmost, lowest bottom and highest top points of the set $S_i^l \cup S_j^r \cup S_k^b \cup S_m^t$.*

This observation provides a way to solve the decision problem. For each point in C as the center for the first square s_1 we do the following:

1. Find $B(K)$. If $B(K)$ has a side of length greater than $\sqrt{\mathcal{A}}$, then the answer to the decision problem is "no".

2. Otherwise define the search region R' which is the locus of all points of L_∞ distance at most $\frac{\sqrt{\mathcal{A}}}{2}$ from all four sides of $B(K)$ and search for a point of C in R'. As was pointed above R' is an axis-parallel rectangle.

As before we perform orthogonal range searching [20] to determine whether there is a point of C in R'. If there is at least one point the answer is "yes"; otherwise it is "no". It remains to explain how we compute efficiently the determinators of $B(S_i^l), B(S_j^r), B(S_k^b), B(S_m^t)$. A bounding box might be empty or degenerate, in which case we compute the rest of determinators for this bounding box. We explain the algorithm for $B(S_i^l)$.

The rightmost point p of S_i^l has been computed. The leftmost point of S_i^l is the leftmost point of S. Thus, it remains to find the lowest and highest

points of the set S_i^l. These values can be precomputed for $i = 1, \ldots, n$. For the dynamic version of the problem computing these values will be too costly. Therefore we maintain a balanced binary search tree T as follows. The nodes of T contain the x-coordinates of the points of S. As we create the tree we maintain at each inner node the maximum of the y-coordinates of the points in the subtree rooted at this node. Thus, given the point p, the highest and lowest points of $B(S_i^l)$ can be found in $O(\log n)$ time. Similarly we do for $B(S_j^r)$, $B(S_k^b)$, $B(S_m^t)$.

Considering the time complexity of the whole algorithm. We spend $O(n \log n)$ to sort all the points of S and build T. For each point in C as a center for s_1 we compute the determinators of $B(S_i^l), B(S_j^r), B(S_k^b), B(S_m^t)$ in total $O(\log n)$ time. Checking the search region R' for a point of C takes $O(\log m)$ time using a standard orthogonal range tree with fractional cascading [20]. We have shown:

Theorem 2.3.5 *Given a set S of n demand points and a set C of m center points in the plane, one can find whether there exist two axis-parallel squares of area $\mathcal{A}$, centered at points of C, that cover all the points of S in time $O(\max(n \log n, m(\log n + \log m)))$ using $O(n + m \log m)$ space.*

Optimization

If we generalize the observation from previous section (see also [69]), we obtain that each rectilinear (x or y) distance between the points of C and the points of S (multiplied by 2 and squared) can be a potential area solution. Thus there are $O(mn)$ potential areas. One possibility for the optimization step is to apply the Frederickson and Johnson algorithm for sorted matrices [54]. For example all the potential size solutions defined by x distances can by represented as shown below. Define a matrix M as following : consider X_S the sorted x order of points of S and also X_C the sorted x order of points of C. Entry $(i, j), 1 \leq i \leq m, 1 \leq j \leq n$ in the matrix M stores the value $x_i^S - x_j^C$ where x_i^S is the x coordinate of the point with index i in X_S and x_j^C is the x coordinate of the point with index j in X_C. The matrix M is sorted, but some of the potential area values appear in matrix with negative sign. To overcome this difficulty, we split M into two matrices M^1 and M^2. The positive entries of M^1 are equal to M except that the negative entries are switched to be 0. In M_2 the negative entries of M become positive and the positive entries of M are switched to 0. Clearly, M^1 and M^2 are sorted matrices and they represent the set of all possible areas according to x-coordinates. Similar procedure works for the y-coordinates, and thus, we obtain four sorted matrices that represent all the possible solutions. This technique works fine in our case, but still has two disadvantages. First disadvantage is that it leads to some additive factor to the runtime of the optimization scheme ($O(m \log(2n/m))$) and second is that we need to maintain these matrices under deletions and insertions for the dynamic version of our problem.

Denote by T_d the runtime of the decision algorithm after the preprocessing step (which is $O(n \log n + m \log m)$). Instead of representing all the distances by sorted matrices, we perform a search of the square size for each point $c \in C$ as a center for s_1. The search is for each axis and in each direction (left, right, up, down). Below we describe the algorithm for axis x, center c of s_1 and the right direction. The size of s_1 (and also s_2) is defined as follows:

1. Let the number of points of S that lie to the right of c be $0 \leq k \leq n$. We denote the x-sorted set of these points by $S^r_{n-k+1} = \{p_{n-k+1}, \ldots, p_n\}$.

2. Perform a binary search on the size of s_1. This size is defined by c and some of S^r_{n-k+1}. Namely we perform the following actions.

   (i) Find a median point $p_{n-\frac{k}{2}+1}$ in the set S^r_{n-k+1}.

   (ii) Compute the x-distance between c and $p_{n-\frac{k}{2}+1}$.

   (iii) This distance multiplied by 2 and squared defines the size $\mathcal{A}$.

   (iv) Run the decision algorithm for $\mathcal{A}$. If the answer to the decision problem is "yes", then set $k = \frac{k}{2}$ and return to step (ii). If the answer to the decision problem is "no", then set $k = k + \frac{k}{2}$ and return to step (ii).

3. Repeat the above procedure for the remaining directions.

The smallest size for which the decision algorithm answers "yes", after running it for each axis and in each direction, is the solution to the optimization problem. Clearly, the described algorithm takes $O(n \log n + T_d \log n)$ time. Thus, we have

Theorem 2.3.6 *Given a set S of n demand points and a set C of m center points in the plane, one can find a solution in time $O(\max(n \log n, m \log n(\log n + \log m)))$ using $O(n + m \log m)$ space.*

A related lower bound: We prove a lower bound to the following (closely related to our) problem: Given an integer A and a set S of n demand points and a set C of m center points on the line, find two segments of length A centered at points of C that cover the largest possible number of points of S. An $\Omega(n \log n)$ lower bound under the linear decision tree model is achieved by a reduction from the set element uniqueness problem as in [17]. We set $C = S$ and asking the question for a limit $A = 0$. The answer is 2 if and only if the elements of the set are disjoint.

The dynamic version

In the dynamic version of problem **p5** points may be inserted to or deleted from S. Our algorithm for the static version can be extended to support

updates and queries in which we ask what are the two smallest enclosing constrained squares that contain the current set S.

The sorted order of the points of S according to x and y coordinates is maintained in the tree T as following. When we delete from or insert to T some point we should update all the maximum y-values stored at the inner nodes on the updating path from the corresponding leaf to the root. In addition, for each node v in T we store the information about the number of nodes that are in the left and right subtrees of the tree rooted at v. This information is useful to compute the median for optimization step (2.i) and to find the set S^r_{n-k+1} by a binary search in T in $O(\log n)$ time. Storing this information does not affect the running time of the insertion or deletion, since we can update while walking on the same updating path. The update of the tree T takes $O(\log n)$ time [35]. When we have a query, we can run our decision algorithm together with the embeded optimization scheme using T in order to get the answer. Using our previous result we can conclude by theorem.

Theorem 2.3.7 *Given a set S of n demand points and a set C of m center points in the plane, where the points of S are allowed to be inserted or deleted, we can answer the query in $O(m \log n(\log n + \log m))$ time. The update time is $O(\log n)$ for the points of S. The preprocessing time is $O(n \log n + m \log m)$.*

As one can see the running time of the query is similar the running time of the algorithm for the static version. However, in the dynamic version of the problem, the query runs without precomputing all the data structures that have been used in the algorithm for a static version. Thus, if $m = o(n)$, we have a polylogarithmic running time query.

Higher dimensions

Our algorithm can be generalized to work in any (fixed) d-dimensional space, $d \geq 3$. The changes we need to perform in order to allow this are following:

1. For the points of C we use d-dimensional orthogonal range tree [29] with a query time $O(\log^{d-1} m)$ for the static version.

2. We maintain d balanced binary search trees T_i, $i = 1, \ldots, d$ for the points of S, one for each axis. But now each node contains the $d-1$ maximal and minimal values of the other coordinates. The update scheme of T_i is done in time $O(d \log n)$.

The rest follows immediately.

Theorem 2.3.8 *Given a set S of n demand points and a set C of m center points in the d-dimensional space, $d \geq 3$, one can find a solution (d-dimensional) in*

$$O(\max(n \log n, m \log n(\log n + \log^{d-1} m)))$$

time.

Theorem 2.3.9 *Given a set S of n demand points and a set C of m center points in the d-dimensional space, $d \geq 3$, where the points of S are allowed to be inserted or deleted, we can answer the query in $O(m \log n(\log n + \log^{d-1} m))$ time. The update time is $O(\log n)$ for the points of S. The preprocessing time is $O(n \log n + m \log^{d-1} m)$.*

Rectangles

We consider first the planar version: Given a set S of n demand points and set C of m center points in d-dimensional space ($d \geq 2$), find two axis-parallel rectangles that cover all the points of S and are centered at the points of C and size of the larger rectangle is minimized. Let us call the solution of this problem *minimal rectangular cover*. Here we consider the size as a perimeter but it could be the area, diagonal length or some other rectangular measure. Hershberger and Suri [60], Glozman et al. [57] and also this thesis (section 2.2) consider a similar two-covering problem (Problem **p4**), but without constraining the centers of the rectangles to be in C. They present an algorithm which runs in time $O(n \log n)$. Our algorithm runs in time $O(mn \log m \log n)$.

The decision algorithm

Assume we are given a rectangle perimeter $\mathcal{A}$. The general idea is very similar to the one used for the squares: we go over all the points in C as a center for the first constrained rectangle r_1, and at each step we check whether the rest of the points can be covered by a second discrete rectangle r_2. The difference is that we do not know the form of r_1 and r_2. In order to solve this problem our decision algorithm tries all possible placements of r_1 on points of C and checks whether the set of points not covered by r_1 can be covered by a constrained rectangle r_2. We demonstrate our algorithm for a point $c \in C$. Four lines l_1, l_2, l_3, l_4 with slopes $-1, 1, -1, 1$ in quadrants in clockwise direction, starting with a positive x and y quadrant, respectively, define the locus of all rectangles with a given perimeter $\mathcal{A}$, centered at O. The lines have to construct a 45° tilted square Q. Assume for a moment that $c = O$. Consider the $S' \subseteq S$ that contains all the points of S which are inside of intersection Q of the halfplanes defined by lines l_1, l_2, l_3, l_4 and containing c. Each point $s \in S'$ defines two rectangles with center c and the given perimeter: where s either determines the *width* of the rectangle, or its *height*. For the time being we look at the rectangle whose width is determined by s. Let s be the point that determines the widest rectangle r_1 and assume w.l.o.g. that s is to the left of c.

We shrink the width of the rectangle, keeping its corners on the corresponding lines until an *event* happens. An event is when a point of S is added

to or deleted from the rectangle during the width shrinking. We check if the rest of points of S is covered by r_2. If it does then we are done; otherwise we continue to shrink the rectangle until the next event. We perform the same actions for the height as well.

In order to speed up this algorithm we define four dynamic subsets U, D, R, L of S' corresponding to the halfplanes that bound r_1. R is the set of all the points of S' that contained in the halfplane to the right of the left side of r_1. Similarly, L (U, D) is the set of points of S' that contained in the halfplane to the left (up, down) of the right (upper, lower) side of the rectangle r_1. We define $p_r(p_l)$ to be the point x-closest to r_1 in R (L) and $p_u(p_d)$ to be the point y-closest to r_1 in U (D). Assume that we are shrinking r_1 in x direction until the next event. Assume that the x-closest neighbor of $p_r(p_l)$ in $R(L)$ is $p_r^h(p_l^h)$ and the y-closest neighbor of $p_u(p_d)$ in $U(D)$ is $p_u^v(p_d^v)$. Thus, our event is when one of p_r^h, p_l^h or p_u^v, p_d^v enters or leaves the rectangle r_1. If the next event is a point from R or L, then the number of points uncovered by r_1 increases by 1, otherwise decreases by 1. We update p_r, p_l, p_u, p_d (and also the subsets U, D, R, L). We check whether r_2 can cover the rest of points $K \subset S$ that are uncovered by r_1 by following algorithm.

We first find the determinators of the bounding box $B(K)$. For the static version of this problem, we can precompute for each set S_i^l, S_j^r, S_k^b, S_m^t the minimal and maximal values. If the length of some side of $B(K)$ is larger than $\mathcal{A}$ then the answer to the decision problem is "no". Otherwise we find a search region R' for the center of r_2. It can be done as following. We make a rectangle r_2 with a perimeter $\mathcal{A}$ and a minimal height such that r_2 covers $B(K)$ and its left lower corner of r_2 coincides with the left lower corner of $B(K)$. We slide r_2 up keeping in touch the left sides of r_2 and $B(K)$ till the left upper corners of r_1 and $B(K)$ coincide. Then we continue sliding r_2 to the right keeping in touch the upper sides of r_2 and $B(K)$, then up while touching right sides and finally to the left while touching down sides till we reach the initial position of r_2. We look onto segments on which the center of the r_2 lies during the sliding motion of the square. This defines a rectangular search region R' where can be found the center of the r_2 that covers $B(K)$, but only for this form of r_2. Generally, r_2 can have an infinite number of forms. But, as was observed in [60], all the rectangles r_2, with the same perimeter and the same lower left corner, have their upper right corner on particular curve Γ. In this case of perimeter Γ is a segment with slope -1. Thus we should compute R' as before for all the forms of r_2 and then take their union, thus obtaining the final search region R''. The region R'' has a form of axis-parallel rectangle rotated to 90°. In order to find whether R'' contains any point of C we perform a standard orthogonal range searching algorithm but only for coordinate axes rotated to 90°.

After preprocessing in $O(n \log n)$ time, the algorithm above runs in $O(n \log m)$ time for one point $c_i \in C$ if the values of $B(S_i^l), B(S_j^r), B(S_k^b), B(S_m^t)$ are precomputed before. This is because we can carry each step of the algorithm

in constant time (except of orthogonal range searching) after computing the first time boundaries of the rectangle r_1.

Thus, we have

Theorem 2.3.10 *Given a set S of n demand points and a set C of m center points in the plane, one can find whether exist two axis-parallel rectangles of perimeter $\mathcal{A}$ centered at the points of C that cover all the points of S in $O(\max(n \log n, mn \log m))$ time.*

Optimization

As in the case of squares we embed the optimization step into the decision algorithm. Similar to the squares algorithm, the explicit use of sorted matrix may lead to the additional additive factor $O(n^2)$ to the runtime for the optimization algorithm. We would like to avoid the explicit use of sorted matrices for the dynamic version of this problem by embedding the search into the decision algorithm. In our case we obtain that each pair containing one rectilinear x-distance and one y-distance between the points of S and the same point in C (multiplied by 4 and summarized) can be a potential perimeter solution. The optimization scheme is very similar to previous one, but instead of performing a binary search for each one of the directions, we define a sorted matrix M whose rows contain the sorted x-distances from $c_i \in C$ to the points of S and whose columns contain the sorted y-distances from $c_i \in C$ to the points of S. Note that the number of elements in M is n^2. Denote by T_d^i the running time of the rectangles decision algorithm for point c_i as a center of r_1. (Thus the total number of potential perimeter solution is mn^2.) Then we can perform a binary search on the elements of the matrix M, making only a constant number of calls to the decision algorithm for point c_i per iteration. As was shown in [54] the overall runtime consumed by the algorithm is $O(\Sigma_{i=1}^{m} T_d^i \log n + n)$. We obtain

Theorem 2.3.11 *Given a set S of n demand points and a set C of m center points in the plane, one can find a minimal rectangular cover in $O(mn \log m \log n)$ time.*

The dynamic version

For dynamization of the decision algorithm for rectangles we use the same updating scheme as for the decision algorithm for squares. The update and query operations the points of S remain the same. We use the same data structures as in the dynamic version of the algorithm for squares. For the optimization step we also have to take care of maintaining the sorted matrix for every point of C. It can be easily done while maintaining dynamically the sorted order of the points of S according to their x and y-coordinates. The difference form the static version is using a balanced binary search trees

in the decision algorithm, but in optimization step we first perform inorder traversal, obtain a sorted list of points and then apply our optimization scheme. As before, the query time remains the same as for the static version, but we need not to recompute again all the data structures that we have used before. Thus, we have

Theorem 2.3.12 *Given a set S of n demand points and a set C of m center points in the plane, where the points of S are allowed to be inserted or deleted, we can answer the query "What is the minimal rectangular cover?" in $O(mn \log n \log m)$ time. The update time is $O(\log n)$ for the points of S. The preprocessing time is $O(n \log n + m \log m)$.*

Higher Dimensions

Similarly to the case of squares, our algorithm can be generalized to work in any (fixed) d-dimensional space, $d \geq 3$. The changes are exactly as in the d-dimensional algorithm for the squares, which include maintaining d-dimensional orthogonal range tree for the points of C, d balanced binary search trees, d sorted orders of points. In addition, we perform the d-dimensional decision algorithm by fixing one dimension and applying recursively $d-1$-dimensional decision algorithm. For the optimization step the number of potential perimeters is mn^d. We can represent them as m sorted matrices, each one of the dimension d. Each sorted matrix is obtained by cartesian product of d 1-dimensional arrays, identically to the plane case. If we denote by T^d running time of the optimization algorithm (static or dynamic) in d-dimensional space, $d \geq 3$, then we can be easily verify that $T^d = O(nT^{d-1})$.

Theorem 2.3.13 *Given a set S of n demand points and a set C of m center points in the d-dimensional space, $d \geq 3$, one can find a minimal rectangular cover in $O(mn^{d-1} \log^{d-1} m \log n)$ time.*

Theorem 2.3.14 *Given a set S of n demand points and a set C of m center points in the d-dimensional space, $d \geq 3$, where the points of S are allowed to be inserted or deleted, we can answer the query "What is the minimal rectangular cover?" in $O(mn^{d-1} \log n \log^{d-1} m)$ time. The update time is $O(\log n)$ for the points of S. The preprocessing time is $O(n \log n + m \log^{d-1} m)$.*

2.3.2 Two constrained parallel squares (p6)

Our problem is: Given a set S of n points in the plane, find a pair of parallel constrained squares whose union contains S, so as to minimize the area (equivalently, the side length) of the larger square. The problem where the squares are not constrained was recently solved by Jaromczyk and Kowaluk [65] in $O(n^2)$ time using $O(n^2)$ space.

We first solve the decision problem for squares with a given area $\mathcal{A}$ in time $O(n^2 \log^2 n)$ and $O(n^2)$ space. For the optimization, we present a parallel version of another algorithm (solving a different problem), to which we apply Megiddo's parametric search [79] to obtain an $O(n^2 \log^4 n)$ time and $O(n^2)$ space optimization algorithm.

The decision algorithm

For each of the input points, $p_i \in S$, draw an axis-aligned square Q_i of area $\mathcal{A}$, centered at p_i. For each p_i denote by U_i the set of points in S that are not covered by Q_i. If, for some i, there is a constrained axis-aligned square of area $\mathcal{A}$ which covers U_i, then we are done. Otherwise, we rotate the squares $\{Q_i \mid i = 1, \ldots, n\}$ simultaneously about their centers, stopping at certain *rotation events* to check if any of the corresponding U_i's can be covered by a parallel square of area $\mathcal{A}$, and halting when the answer is "yes".

A *rotation event* occurs whenever a point of S enters or leaves a square Q_i, $i = 1 \ldots n$. When a square Q_i rotates by $\frac{\pi}{2}$ from its initial axis-aligned position, every point of S enters and leaves Q_i at most once. Thus, the number of rotation events for Q_i is $O(n)$. For all the points in P we can precompute all the $O(n^2)$ rotation events in $O(n^2)$ time with $O(n^2)$ space. We sort the rotation events according to their corresponding angles.

We compute the initial convex hulls for each U_i, $i = 1, \ldots, n$ (i.e., at orientation $\theta = 0$), and start rotating the squares till we get to the next rotation event. Assume that at the current rotation event a point p_j enters Q_i. (The case where a point p_j leaves Q_i is treated similarly.) The set U_i and its convex hull are updated as p_j leaves U_i, and we check whether there exists a constrained cover of S involving Q_i and another constrained square (that covers U_i).

We explain how this is done for one square Q_i at orientation $\theta = 0$. First we find the tangents of the convex hull of U_i that are parallel to the sides of Q_i. They define a rectangle R which is the bounding box of U_i. If R has a side of length greater than $\sqrt{\mathcal{A}}$, then none of the other $n-1$ constrained squares covers U_i. Otherwise we define a *search region* R' which is the locus of all points of L_∞ distance at most $\frac{\sqrt{\mathcal{A}}}{2}$ from all four sides of R, and search for a point of S in R'. (Clearly C is a rectangle whose sides are parallel to the sides of Q_i.) We perform orthogonal range searching to determine whether there is a point of S in R'. If there exists such a point then the answer to the decision problem is "yes".

Assume we have computed all the rotation events and have $O(n^2)$ rectangular search regions associated with them. (Assume the coordinate system rotates together with the rotating squares $\{Q_i\}$, thus, at any rotation event, the corresponding rectangular search region is parallel to the current axes.) In order to perform orthogonal range search on the rectangular regions we use a dynamic orthogonal range search tree which is updated at certain rotation

events as follows.

Denote by L the list of all $O(n^2)$ lines passing through pairs of points in S. Let P consist of all the slopes of lines in L that lie in the range $[0, \pi/2)$, and of all the slopes in the range $[0, \pi/2)$ of lines that are perpendicular to the lines in L. We sort P, obtaining the sorted sequence $\{\alpha_1, \alpha_2, \ldots\}$. We rotate the axes so that the x-axis has slope α_1, and compute an orthogonal range search tree for S with respect to the rotated axes, storing just the labels of the points of S in the tree. For each search region whose side slope is between α_1 and α_2 we perform a usual range search with this tree. Before considering the next search regions, we rotate the axes some more until the x-axis has slope α_2. Notice that just one pair of points in S has swapped in x or y order in this angle range. We update the range search tree accordingly: Assuming the leaves of the main structure in the range tree are sorted by x-coordinate, and the leaves in the secondary trees are sorted by y-coordinate. If, when moving from α_1 to α_2, the swap occurred in the x-order of the pair of points, then we swap the (labeling of the) points in the main structure and in the secondary structures affected by that swap; if the swap occurred in the y-order, then we swap the labeling in the affected secondary structures. Now we can proceed with the search ranges whose sides have slopes between α_2 and α_3. And so on.

We analyze the time and space required for the decision algorithm. The total number of rotation events is $O(n^2)$. They can be precomputed and sorted in $O(n^2 \log n)$ time with $O(n^2)$ space. Similarly P can be obtained and sorted within the same bounds. Merging the two sets of slopes (rotation events and P) is done in time $O(n^2)$. Initially computing the convex hulls for all sets U_i takes $O(n^2 \log n)$ time with $O(n^2)$ space. Applying the data structure and algorithm of Overmars and van Leeuwen [88], each update of a convex hull takes $O(\log^2 n)$ time, totaling in $O(n^2 \log^2 n)$ time and $O(n^2)$ space for all rotation events. Our range searching algorithm takes $O(\log^2 n)$ time per query and per update, after spending $O(n \log n)$ preprocessing time and using $O(n \log n)$ space (notice that this is the total space requirement for the range searching), and we perform $O(n^2)$ queries and updates. Thus we have shown:

Theorem 2.3.15 *Given a set S of n points and an area $\mathcal{A}$, one can decide whether S can be covered by two constrained parallel squares, each of area $\mathcal{A}$, in $O(n^2 \log^2 n)$ time and $O(n^2)$ space.*

Optimization

Having provided a solution to the decision problem, we now return to the minimization problem. The number of candidate square sizes is $O(n^4)$ (see below and Figure 2.3). The candidate sizes are determined by either

- A point of S as a center of a square (see Figure 2.3(i)–(iv)) and either

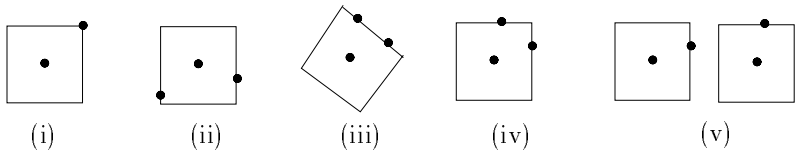

Figure 2.3: Critical events that determine candidate square sizes. Cases (i) – (iv) involve a single square, and case (v) two squares.

(i) another point of S on a corner of this square, or (ii) two points of S on parallel sides of the square, or (iii) two points of S on one side of the square, or (iv) two points of S on adjacent sides of the square, or

- Two points of S as centers of two squares and another point of S on the boundary of each of the squares (Figure 2.3(v)).

In order to apply the Megiddo optimization scheme we have to parallelize our decision algorithm. However, the range searching part of the decision algorithm is not parallelizable, so, as in [7], we come up with an auxiliary problem whose parallel version will generate the optimal solution to our problem.

The auxiliary problem is described as follows. Assume we have a set S of $n > 2$ points and a fixed size d. Assume we have produced the set of strips such that each strip is of width d and contains at least one point of S on each of its boundaries. In this situation a point on one boundary might stand for the square center and the point on the other boundary is the one on the side of the square. Maintain the set of strips by storing their slopes and the corresponding pairs of points that define them in P. Let $\bar{P}$ be the set of *slopes* obtained by the slopes of P by adding $\pi/2 \pmod{\pi}$. With each slope in $\bar{P}$ we store the pair of points associated with the corresponding slope in P.

A slope $\bar{s} \in \bar{P}$ stands for a pair of square sides perpendicular to the ones defined by its corresponding slope $s \in P$. So that if two perpendicular slopes, s_1 and s_2 (in P) define a square (as in Figure 2.3 (i),(iv) and (v)), then s_1 and $\bar{s_2}$ are equal. The set of squares thus defined is a superset of the candidate solution squares as defined above. Let $\mathcal{P} = P \cup \bar{P}$ be a set of slopes with their associated point pairs. The **auxiliary problem** is to sort the slopes in $\mathcal{P}$.

Clearly not all pairs of points in S define strips, and thus slopes, in $\mathcal{P}$. A pair of points in S whose distance is smaller than d will not generate the required width strip. For every pair of points in S whose distance from each other is larger than d, there are exactly two slopes for which the width of the strip, with a point of this pair on each of its boundaries, is d. We add these slopes (and their $\bar{P}$ corresponding slopes) to $\mathcal{P}$. Reporting the sorted order of $\mathcal{P}$ can be done in $O(n^2 \log n)$ time, and a parallel algorithm with $O(n^2)$

processors will sort the list in $O(\log n)$ time [33].

We now want to (generically) apply this parallel sort algorithm for finding the optimal square size d^*. For this we first augment our algorithm, as in [7], and get an initial interval where d^* resides. We perform a preliminary stage that disposes of the cases in which the width of the strip is exactly the distance between two points of S, and those in which the width is the distance between two points multiplied by $\sqrt{2}/2$. We call these distances *special distances*. We can afford to list all these $O(n^2)$ strip widths, sort them, and perform a binary search for d^* over them, applying our decision algorithm of the previous subsection at each of the comparisons. This results in an initial closed interval of real numbers, I_0, that contains the optimal square size d^*, and none of the just computed special sizes is contained in its interior.

Consider now a single step in the parallel sort (the auxiliary problem). In this step we perform $O(n^2)$ slope comparisons, each comparison involving two pairs of points. There are two cases: (a) the two compared slopes are from P (or both are in $\bar{P}$), and (b) one slope is in P and the other in $\bar{P}$. Let one such comparison involve the pairs (p_1, p_2) and (p_3, p_4). In order to resolve this comparison, we must compute for the point pair (p_1, p_2) the slopes of the two strips of width d^* that have p_1 on one boundary of the strip and p_2 on the other. Similarly, we compute the slopes of the two strips of width d^* through (p_3, p_4). Then we sort the four strips by their slopes. Of course, we do not know d^*, so we compute the (at most two) *critical values* d where the sorted order of the four strips changes, namely, for case (a) above, where the two strips are parallel, and for case (b), when the two strips are perpendicular to each other. We do this for all $O(n^2)$ critical value comparisons. Now we apply the decision algorithm of the subsection above to perform a binary search over the $O(n^2)$ critical values that were computed. Thus we find an interval $I \subseteq I_0$ where d^* resides, resolve all the comparisons of this parallel stage, and proceed to the next parallel stage.

What does resolving mean here? See Figure 2.4 which depicts case (a). If the comparison was made for two pairs of points (p_1, p_2) and (p_3, p_4) then, if the distance between a pair of points, $d_1 = (p_1, p_2)$ or $d_2 = (p_3, p_4)$, is smaller than the smaller endpoint of the current interval I then this pair will not have a strip of width d^* and it is omitted from the rest of the sort. If the distance is larger than the smaller endpoint of I then the slope ordering of the four strips at d^* is uniquely determined as follows. In Figure 2.4 (a) the strips s_1 and s_2 are parallel at some width d', and in Figure 2.4 (b) we plot the strips of width d^* for the two pairs of points. In Figure 2.4 (c) we graph d as a function of $\theta \in [0, \pi)$ for the two pairs of points. The graph of $d = d_1 \cos(\theta - \theta_1)$ achieves its maximum at (θ_1, d_1), and similarly the graph of $d = d_2 \cos(\theta - \theta_2)$ achieves its maximum at (θ_2, d_2), where θ_1 (θ_2) is the angle that the line perpendicular to the line through (p_1, p_2) ((p_3, p_4)) makes with the positive x-axis. It is easy to see that for every d each pair of points has

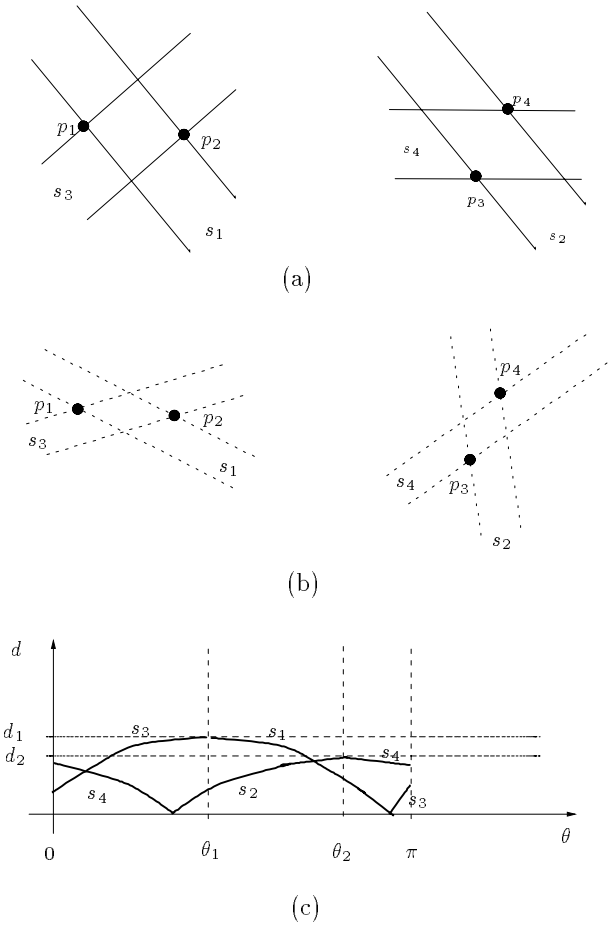

Figure 2.4: Slope ordering for the comparison of (p_1, p_2) and (p_3, p_4): (a) strips s_1 and s_2 are parallel for some d, (b) the ordering of the slopes at d^*, (c) d as a function of θ

two strips and that the two functions intersect at two points. We split the domain of definition of each function to two parts, one in which the function strictly increases and one in which it strictly decreases. In Figure 2.4 (a) and 2(b) the strip s_1 corresponds to the decreasing half of the function in Figure 2.4 (c) and s_3 to the increasing half. Similarly with the strips of (p_3, p_4), s_2 corresponds to the increasing half and s_4 to the decreasing half. Thus the order of the strips at d^* is the order in which the line $d = d^*$ intersects their functions, and the width values at the intersection points of the two functions consist of the critical values for these two pairs of points.

For case (b) assume the pair (p_1, p_2) belongs to a strip of $\bar{P}$. We simply cyclically shift the function of (p_1, p_2) (of P) by $\pi/2$. The intersection points of the functions are now at two values of d' where the two strips are perpendicular to each other, and all the rest of the argument is analogous.

Note: We have to be a little more careful here about the notion of the domain of definition of the functions, and we might want to break the domain of definition of the functions also at $\theta = 0$. This is a slight formality that we neglect since it does not change anything in the analysis.

The closed interval I is always guaranteed to contain d^* but we need to show that a comparison is made where $d = d^*$.

Claim 2.3.16 *If d^* is not one of the special distances then the slope order of the strips changes as d changes from values slightly smaller than d^* to values slightly larger than d^*.*

Proof. Observe again Figure 2.3. Clearly if d^* is not one of the special distances then it involves two pairs of points. In Figure 2.3 (ii), (iii), (iv), the pairs are the center point of the square paired with each of the two points on the boundary of this square, and in Figure 2.3 (v) the pairs are the center point of each square paired with the point on the side of its square. None of these cases represents a special distance, and hence the slopes of the strips are monotone functions of their widths. These two monotone functions intersect at d^* thus in a small neighborhood of d^* one function is above the other for $d < d^*$ and below for $d > d^*$. ∎

Note that at some stage the optimal solution will appear on the boundary of the interval I computed at that stage (it could even appear on the boundary of I_0). However, once it appears, it will remain one of the endpoints of all subsequently computed intervals. At the end, we run the decision algorithm for the left endpoint of the final interval. If the answer is positive, then this endpoint is d^*, otherwise d^* is the right endpoint of the final interval.

Theorem 2.3.17 *Let S be a set of n points, we can find a pair of parallel constrained squares whose union covers S and such that the area of the larger square is minimized in $O(n^2 \log^4 n)$ time and $O(n^2)$ space.*

2.3.3 Two constrained general squares (p7)

Now the squares may rotate independently. We first state a subproblem whose solution is used as a subroutine in the full solution. Then we present an algorithm for solving the decision problem. This algorithm is used to perform a binary search over the sorted set of potential solutions, producing the solution to the optimization problem.

The subproblem: Given a set S of n points in the plane and a point q, find the minimum area square that is centered at q and that covers S. The square may rotate.

The algorithm for solving the subproblem is as follows. Assume q is the origin. Let θ be an angle in $[0, \frac{\pi}{2})$. Consider the projections, $x_i(\theta)$ and $y_i(\theta)$, of a point $p_i \in S$ on the x-axis and y-axis, after rotating the axes by θ. If the distance between p_i and q is d_i, and the angle between the vector p_i and the x-axis at its initial position is θ_i, then we have

$$x_i(\theta) = d_i \cos(\theta_i - \theta) \text{ and } y_i(\theta) = d_i \sin(\theta_i - \theta) \ .$$

A square centered at q rotated by angle θ that has p_i on its boundary is of side length $2 \times \max\{|x_i(\theta)|, |y_i(\theta)|\}$. Note that it is enough to rotate the axes by angle $\theta, 0 \leq \theta < \frac{\pi}{2}$, in order to get all possible sizes of squares centered at q having p_i on their boundary.

Observe the plane (θ, z), on which we graph both $z_{2i-1}(\theta) = 2|x_i(\theta)|$ and $z_{2i}(\theta) = 2|y_i(\theta)|$, $i = 1, \ldots n$. We call the set of these $2n$ functions E_q, and depict them in Figure 2.5. It is easy to see that every pair of functions z_j and z_k intersects at most twice. The upper envelope of the functions in E_q denotes, for each θ, the size $z(\theta)$ of the smallest square (centered at q and rotated by θ) that covers S, and the point (or two points) of S corresponding to the function (or two functions) that attains (attain) the maximum at this θ is the point (are the two points) of S on the boundary of the square. The lowest point on this envelope gives the angle, the size, and the point(s) that determine the minimal square. The upper envelope, and the lowest point on it, can be computed in $O(n \log n)$ time [98], and this is the runtime of the solution of the subproblem above.

For the two squares decision problem we repeat some notations and ideas from the previous section. Let Q_i be a square of the given area $\mathcal{A}$ centered at $p_i \in S$. We define rotation events for Q_i as the angles at which points of S enter or leave Q_i. Denote by U_i the set of points not covered by Q_i at the current rotation angle. Using the subproblem described above, we find the smallest constrained square that covers U_i, by computing n sets E_j, where E_j is the set of $2|U_i|$ functions associated with the center point p_j.

We describe our algorithm for determining whether one of the constrained centers is some fixed point $p_i \in S$. Then we apply this algorithm for each of the points in S. Initially, at $\theta = 0$, we construct all the sets E_j, so that each set contains only the functions that correspond to the points in the initial U_i.

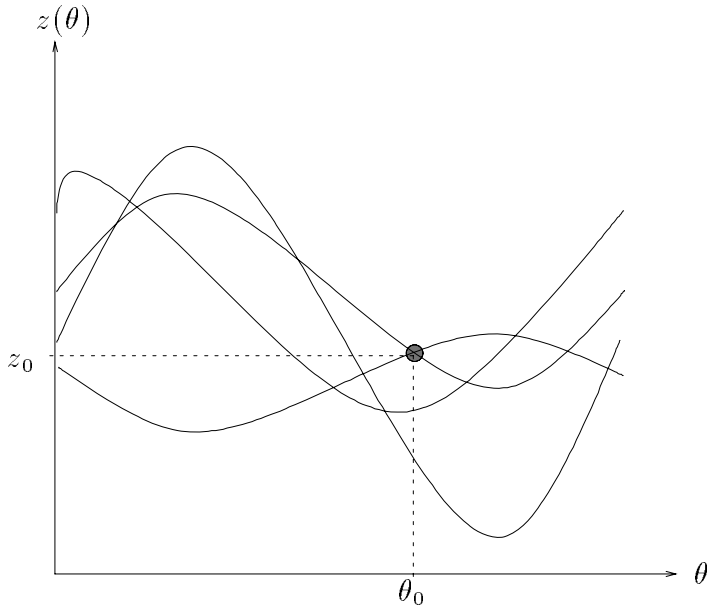

Figure 2.5: The functions z_i and the lowest point (θ_0, z_0) on their upper envelope

The rotation events for this phase are those caused by a point of S entering or leaving Q_i. At each rotation event we update U_i and all the sets E_j. We then check whether there is a point on the upper envelope of one of the E_j's which is below the line $z = \sqrt{\mathcal{A}}$. If there exists a point (θ_0, z_0), $z_0 \leq \sqrt{\mathcal{A}}$ on the upper envelope of some E_j, then the square Q_i at its current position, and the square Q_j at angle θ_0 are the solution to the decision problem.

Updating the upper envelopes corresponding to the sets E_j turns out to be time consuming, therefore we transform the problem of determining whether one of the upper envelopes has a low enough point to a segment stabbing problem as follows. Observe one set E_j. If we draw a horizontal line at $z = \sqrt{\mathcal{A}}$, then each function curve in E_j is cut into at most three continuous subcurves, of which at most two lie below the line. We project all the subcurves of E_j that are below the line on the θ-axis, obtaining a set of segments. Assume the number of points in U_i is k, then if (and only if) there is a point θ_0 on the θ-axis that is *covered* by $2k$ segments then there is a square of the required size, of orientation θ_0, centered at p_j which covers the points of U_i.

We construct a segment tree T_j [83] with $O(n)$ leaves (for the segments obtained from all potential curves in E_j). Each node in the tree contains, besides the standard segment information, the maximum cover of the node (namely, the largest number of segments that can be stabbed in the range of the node, for details see [83]). The root of the tree contains the *maximum cover* of the whole range $0 \leq \theta < \frac{\pi}{2}$. The size of one tree is $O(n)$ and each

update is performed in time $O(\log n)$. Initially, at $\theta = 0$, we insert into T_j the segments corresponding to the curves of the points in U_i, and check whether the maximum cover equals twice the cardinality of U_i. One update to U_i involves at most four segment updates in T_j.

We consider the time and space complexity of the algorithm. For one point p_i as a candidate center, the initial trees T_j are constructed in time $O(n^2 \log n)$, occupying $O(n^2)$ space. There are $O(n)$ rotation events for Q_i, and an update to one T_j is performed in $O(\log n)$ time, totaling $O(n^2 \log n)$ time for all rotation events and all T_j's. The space requirement is $O(n^2)$. Applying the algorithm sequentially for all i in $\{1, \ldots, n\}$ gives $O(n^3 \log n)$ runtime, while the space remains $O(n^2)$.

To find an optimal solution, we perform for each i as above the following. Assume $p_i \in S$ is one of the two centers in the solution. The corresponding square is defined either by another point of S in its corner, or by two points of S on its boundary. So we compute the $O(n^2)$ potential area sizes with p_i as the center. We sort the area sizes and apply binary search to find the smallest area squares that cover S with p_i as one of the centers in the solution. At each of the $O(\log n)$ search steps, we apply the decision algorithm above (just with p_i as one of the centers). We perform this search for all $i \in \{1, \ldots, n\}$. We have shown:

Theorem 2.3.18 *Given a set S of n input points we can find a pair of general constrained squares whose union covers S and such that the area of the larger square is minimized in $O(n^3 \log^2 n)$ time and $O(n^2)$ space.*

Conclusions

We have considered several instances of the center problems, namely, when the centers of the objects are constrained to lie on the input points. Finding nontrivial lower bounds and improving the running time of the above algorithms can be the challenging questions in the near future.

Chapter 3

Facility Location

In this chapter we consider the problems of the following type: "Given a set S of n sites (points) in metric space, position position a point (facility), or a number of facilities, in the plane such that a distance between the facility and given n points is minimized or maximized". In particular, we are interested in the following problems:

p8: Let S be a set of n points in the plane, enclosed in a rectangular region R. Let each point p of S have two positive weights $w_1(p)$ and $w_2(p)$. Find a point $c \in R$ which maximizes

$$\min_{p \in S}\{\max\{w_1(p) \cdot d_x(c,p), w_2(p) \cdot d_y(c,p)\}\},$$

where $d_x(c,p)$ defines the distance between the x coordinates of c and p, and $d_y(c,p)$ defines the distance between the y coordinates of c and p.

p9: Given a set S of n points and a number $1 \leq k \leq n-1$ find a point p such that sum of the $L_1(L_\infty)$ distances from p to all the subsets of S of size k is minimized. For this problem we consider two cases: the *discrete* case – where $p \in S$, and the *continuous* case where p is any point in the plane.

p10: Let S be a set of n points in the plane (called *demand* points), and let $\mathcal{R}$ be a set of m, $m \leq n$, regions in the plane (called *neighborhoods*). Let k be a positive integer (k is the number of facilities, e.g., garbage dumps, to be placed). Find k sites $c_1, \ldots, c_k$ for the k facilities, such that (i) $C = \{c_1, \ldots, c_k\}$ is a *piercing set* for $\mathcal{R}$, that is, each of the neighborhoods in $\mathcal{R}$ is served by at least one facility that is located in the neighborhood. (ii) The minimal distance between a demand point in S and a site in C is maximized.

For these problems we developed a new data structure which allow us to obtain very efficient algorithms.

3.1 Undesirable Facility Location (p8)

In this section we first present a sequential algorithm that answers a decision query of the form: given $d > 0$, determine whether there exists a location $c \in R$ whose x-distance from each point $p_i \in S$ (the distance between the x coordinates of c and p_i) is $\geq d \cdot w_1(p_i)$, and whose y-distance to the points of S is $\geq d \cdot w_2(p_i)$. We will use this sequential algorithm in order to obtain two different algorithms for solving our problem.

The first is based on the parametric search optimization scheme [83] and, thus, we provide a parallel version of the decision algorithm in order to use it.

The second uses another optimization approach, proposed in [82]. The main idea is to represent a set of potential solutions in a compact, efficient way, use a parallel sorting scheme and then look for our solution by some kind of a binary search. The running time of the algorithm is $O(T_s \log n)$, where T_s stands for the running time of the sequential decision algorithm.

3.1.1 The sequential algorithm

The formulation of the decision problem above implies that each point $p_i \in S$ defines a *forbidden* rectangular region

$$R_i = \{r \in R^2 | d_x(r, p_i) < d \cdot w_1(p_i), d_y(r, p_i) < d \cdot w_2(p_i)\}$$

where c cannot reside. Denote by U_R the union of all the R_i. An *admissible location* for c exists if and only if $R \cap U_R \neq \emptyset$. In other words, we are given a set of n rectangles R_i and want to find whether U_R *covers* R. When each point has the same weight in both axes then the combinatorial complexity of the boundary of U_R is linear in the number of points. In our case the boundary of U_R has $\Theta(n^2)$ vertices in the worst case.

The problem of finding whether a set of n rectangles covers a rectangular region R has been solved in $O(n \log n)$ time using the *segment tree* T [83]. We outline this well known sequential algorithm for the sake of clarity of our parallel algorithm.

Denote by $L = \{x_1, \ldots, x_{2n}\}$ the x coordinates of the endpoints of the horizontal sides of the rectangles. We call the elements of L the *instances* of T. Similarly, let $M = \{y_1, \ldots y_{2n}\}$ be the list of y coordinates of the endpoints of the vertical sides of the rectangles. Assume each list is sorted in ascending order. The leaves of the segment tree T contain *elementary segments* $[y_i, y_{i+1})$, $i = 1, \ldots, 2n-1$, in their *range* field. The range at each inner node in T contains the union of the ranges in the nodes of its children.

A vertical line is swept over the plane from left to right stopping at the instances of T. At each instance x, either a rectangle is added to the union or it is deleted from it. The vertical side v of this rectangle is inserted to (or

deleted from) T (v is stored in $O(\log n)$ nodes and is equal to the disjoint union of the ranges of these nodes). The update of T at instance x involves maintaining a *cover number* in the nodes. The cover number at a node counts how many vertical rectangle sides cover the range of this node and do not cover the range of its parent. If at deleting a rectangle the height of R is not wholly covered by all the vertical segments that are currently in T, then the answer to the decision problem is "yes". Namely, we found a point in R which is not in U_R, and we are done. If the answer is "no" then we update T and proceed to the next instance. Thus

Lemma 3.1.1 *Given a fixed $d > 0$ we can check in $O(n \log n)$ time, using $O(n)$ space, whether there exists a point $c \in R$, such that for every point $p_i \in S$ the following holds: $d_x(c, p_i) \cdot w_1(p_i) \geq d$ and $d_y(c, p_i) \cdot w_2(p_i) \geq d$.*

3.1.2 The parallel version and the optimization

In order to produce an efficient parallel algorithm for the decision problem we add some information into the nodes of T. This information encaptures the cover information at each node, as will be seen below.

Let $L = \{x_1, x_2, \ldots, x_{2n}\}$ be the list of instances as above. Let the projection of a rectangle R_j on the x axis be $[x_i, x_k]$. We associate with R_j a *life-span* integer interval $l_j = [i, k]$. Let v_j be the projection of R_j on the y axis. The integer interval l_j defines the instances at which the segment v_j is stored in T during the sequential algorithm. We augment T by storing the life-span of each vertical segment v_j in the $O(\log n)$ nodes of T that v_j updates. We further process each node in T so that it contains a list of *cover two* life-ranges. This is a list of intervals consisting of the pairwise intersections of the life-spans in the node. For example, assume that a node s contains the life-spans $[1, 7], [3, 4]$ and $[5, 6]$. The list of cover two at s is $[3, 4]$ and $[5, 6]$. If a vertical segment is to be deleted from s at instances x_1, x_2 or x_7, then s will be exposed after the deletion. But if the deletion occurs at instance x_3, x_4, x_5 or x_6 then, since the cover of s is 2 at this instance, s will not be exposed by deleting v_j.

Our parallel algorithm has two phases: phase I constructs the augmented tree T and phase II checks whether R gets exposed at any of the deletion instances.

Phase I The segment tree T can be easily built in parallel in time $O(\log n)$ using $O(n \log n)$ processors [15]. Unlike in Lemma 3.1.1 above, where we store in each node just the cover number, here we store for each segment its life-span in $O(\log n)$ nodes. Thus T occupies now $O(n \log n)$ space [83]. Adding the cover two life-span intervals is performed as follows.

We sort the list of life-spans at each node according to the first integer in the interval that describes a life-span. We merge the list of life-spans at each node as follows. If two consecutive life-spans are disjoint we do not do

anything. Assume the two consecutive life-spans $[k_1 k_2]$ and $[g_1, g_2]$ overlap. We produce two new *life-ranges*:

(a) the life-range of cover at least one – $[k_1, \max(k_2, g_2)]$ and

(b) the life-range of cover at least two – $[g_1, \min(k_2, g_2)]$.

We continue to merge the current life-range of cover at least one from item (a) above with the next life-span in the list till the list of life-spans is exhausted. We next merge the cover two life-ranges into a list of disjoint intervals by taking the unions of overlapping intervals. At this stage each node has two lists of life-ranges. But this does not suffice for phase II. For each node we have to accumulate the cover information of its descendants. Starting at the leaves we recursively process the two lists of life-ranges at the nodes of T separately. We describe dealing with the list of cover one. Assume the two children nodes of a node s contain intersecting life-ranges, then this intersection interval is an interval of instances where all the range of s is covered. We copy the intersection interval into the node s. When we are done copying we merge the copied list with the node's life-span list as described above, and then merge the list of cover two with the copied list of cover two, by unioning overlapping intervals.

Phase II. Our goal in this phase is to check in parallel whether, upon a deletion instance, the height of R is still fully covered or a point on it is exposed. We do it as follows. Assume that the vertical segment v is deleted from T at the j^{th} instance. We go down the tree T in the nodes that store v and check whether the life-span lists at **all** these nodes contain the instance j in their list of cover two. If they do then (the height of) R is not exposed by deleting v.

Complexity of the algorithm.

It is easy to show that the life-range lists do not add to the amount of required storage. The number of initial life-span intervals is $O(n \log n)$. The number of initial life-ranges of cover two cannot be greater than that. It has been shown [31] that copying the lists in the nodes in the segment tree to their respective ancestors does not increase the asymptotic space requirement. The augmentation of T is performed in parallel time $O(\log n)$ with $O(n \log n)$ processors as follows. We allocate a total of $O(n \log n)$ processors to merge the life-span ranges in the nodes of T, putting at each node a number of processors which is equal to the number of life-span ranges in the node. Thus the sorting and merging of the life-span ranges is performed in parallel in time $O(\log n)$.

The checking phase is performed in parallel by assigning $O(\log n)$ processors to each deletion instance. For the deletion of a vertical segment v, one processor is assigned to each node that stores v. These processors perform in parallel a binary search on the cover two life-ranges of these nodes. Thus the checking phase is performed in time $O(\log n)$.

Summing up the steps of the parallel algorithm, we get a total of $O(\log n)$ parallel runtime with $O(n \log n)$ processors and $O(n \log n)$ space. Plugging this algorithm to the parametric search paradigm [79] we get

Theorem 3.1.2 *Given a set S of n points in the plane, enclosed in a rectangular region R, and two positive weights $w_1(p)$ and $w_2(p)$ for each point $p \in S$, we can find, in $O(n \log^3 n)$ time, a point $c \in R$ which maximizes*

$$\min_{p \in S}\{\max\{w_1(p) \cdot d_x(c,p), w_2(p) \cdot d_y(c,p)\}\}.$$

3.1.3 Another approach

By carefully looking at the respective Voronoi diagram we have the following crucial observation.

Observation 3.1.3 *Assume that the optimal solution is not attained on the boundary of the rectangle. Then, without loss of generality, there is an optimal point c, and two points p and q such that either*

$$w_1(p)d_x(c,p) = w_1(q)d_x(c,q) = optimal \quad value,$$

or

$$w_2(p)d_y(c,p) = w_2(q)d_y(c,q) = optimal \quad value.$$

The above observation with a given assumption implies that the optimal value is an element in one of the following four sets:
$S_1 = \{(p_x + q_x)/(1/w_1(p) + 1/w_1(q)) : p, q \in S\}$,
$S_2 = \{(p_x - q_x)/(1/w_1(p) - 1/w_1(q)) : p, q \in S\}$,
$S_3 = \{(p_y - q_y)/(1/w_2(p) - 1/w_2(q)) : p, q \in S\}$,
$S_4 = \{(p_y + p_y)/(1/w_2(p) + 1/w_2(q)) : p, q \in S\}$.

Megiddo and Tamir [82] describe how to search for the optimal value r^* within a set of the form: $S' = \{(a_i + b_j)/(c_i + d_j) : 1 \leq i, j \leq n\}$. Thus there will be given $4n$ numbers $a_i, b_j, c_i, d_j (1 \leq i, j \leq n)$, and we will have to find two elements $s, t \in S'$ such that $s < r^* \leq t$ and no element of S' is strictly between s and t.

Set S' consists of the points of intersection of straight lines $y = (c_i x - a_i) + (d_j x - b_j)$ with the x-axis. The search will be conducted in two stages. During the first stage we will identify an interval $[s_1, t_1]$ such that $s_1 < r^* \leq t_1$ and such that the linear order induced on $\{1, \ldots, n\}$ by the numbers $c_i x - a_i$ is independent of x provided $x \in [s_1, t_1]$. The rest of the work is done in Stage 2.

Stage 1. We search for r^* among the points of intersections of lines $y = c_i x - a_i$ with each other. The method is based on parallel sorting scheme. Imagine that we sort the set $\{1, \ldots, n\}$ by the $(c_i x - a_i)$'s, where x is not known yet. Whenever a processor has to compare some $c_i x - a_i$ with $c_j x - a_j$,

we will in our algorithm compute the critical value $x_{ij} = (a_i - a_j)/(c_i - c_j)$. We use Preparata's [68] parallel sorting scheme with $n \log n$ processors and $O(\log n)$ steps. Thus, a single step in Preparata's scheme gives rise to the production of $n \log n$ points of intersection of lines $y = c_i x - a_i$ with each other. Given these $n \log n$ points and an interval $[s_0, t_0]$ which contains r^*, we can in $O(n \log n)$ time narrow down the interval so that it will still contain r^* but no intersection point in its interior. This requires the finding of medians in sets of cardinalities $n \log n$, $\frac{1}{2} n \log n$, $\frac{1}{4} n \log n, \ldots$ plus $O(\log n)$ evaluations of the sequential algorithm for the decision problem. Since the outcomes of the comparisons so far are independent of x in the updated interval, we can proceed with the sorting even though x is not specified. The effort per step is hence $O(n \log n)$ and the entire Stage 1 takes $O(n \log^2 n)$ time.

Stage 2. When the second stage starts we can assume without loss of generality that for $x \in [s_1, t_1]$ $c_x - a_i \leq c_{i+1} - a_{i+1}$, $i = 1, \ldots, n-1$. Let $j (1 \leq j \leq n)$ be fixed and consider the set S_j of n lines $S_j = \{y = c_i x - a_i + d_j x - b_j, i = 1, \ldots, n\}$. Since S_j is "sorted" over $[s_1, t_1]$, we can find in $O(\log n)$ evaluations of the sequential algorithm for the decision problem a subinterval $[s_1^j, t_1^j]$ such that $s_1^j < r^* \leq t_1^j$, and that no member of S_j intersects the x-axis in the interior of this interval. We work on the S_j's in parallel. Specifically, there will be $O(\log n)$ steps. During a typical step, the median of the remainder of every S_j is selected (in $O(1)$ time) and its intersection point with the x-axis is computed. The set of these n points is then searched for r^* and the interval is updated accordingly. This enablesus to discard a half from each S_j. Clearly a single step lasts $O(n \log n)$ time and the entire stage is carried out in $O(n \log^2 n)$ time.

At the end of second stage we have the values $\{s_1^j\}$ and $\{t_1^j\}, j = 1, \ldots, n$. Defining $s = \max_{1 \leq j \leq n} \{s_1^j\}$ and $t = \min_{1 \leq j \leq n} \{t_1^j\}$ we obtain $s < r^* \leq t$, and no element of S' is strictly between s and t.

The case with the optimal solution attained on the boundary of the rectangle can be treated as subcase of a previous case. Thus we conclude by a theorem.

Theorem 3.1.4 *Given a set S of n points in the plane, enclosed in a rectangular region R, and two positive weights $w_1(p)$ and $w_2(p)$ for each point $p \in S$, we can find, in $O(n \log^2 n)$ time, a point $c \in R$ which maximizes*

$$\min_{p \in S} \{\max\{w_1(p) \cdot d_x(c,p), w_2(p) \cdot d_y(c,p)\}\}.$$

3.2 Desirable Facility Location Problem (p9) - discrete case

The discrete min-sum problem is defined as follows. Given a set S of n points in the plane and a number k. Find a point in S such that the sum of

distances from it to its k nearest neighbors in S is minimized. Our algorithms compute, for each point of S, the sum of distances from it to its k nearest neighbors in S, and output a point which minimizes the sum. First we deal with the special case of the discrete min-sum problem when $k = n - 1$.

3.2.1 The discrete min-sum problem for $k = n - 1$

This min-sum problem appears in [17] with an $O(n^2)$ trivial solution. Below we present an algorithm that solves this problem for the L_1 metric in $O(n \log n)$ time.

The L_1 metric is separable, in the sense that the distance between two points is the sum of their x and y-distances. Therefore we can solve the problem for the x and y-coordinates separately. We regard the x coordinates part. We sort the points according to their x-coordinates. Let $\{p_1, \ldots, p_n\}$ be the sorted points. For each $p_i \in S$ we compute the sum σ_i^x of the x-distances from p_i to the rest of the points in S. This is performed efficiently as follows. For the point p_1 we compute σ_1^x by computing and summing up each of the $n - 1$ distances. For $1 < i \leq n$ we define σ_i^x recursively: assume the x-distance between p_{i-1} and p_i is δ, then $\sigma_i^x = \sigma_{i-1}^x + \delta \cdot (i-1) - \delta \cdot (n-i+1)$. Clearly the sums σ_i^x (for $i = 1, \ldots n$) can be computed in linear time when the points are sorted. We compute σ_i^y analogously. Assume the point $p \in S$ is i^{th} in the x order and j^{th} in the y order. The sum of distances from p to the points in S is $\sigma_{ij} = \sigma_i^x + \sigma_j^y$. The point which minimizes this sum is the sought solution.

Theorem 3.2.1 *Given a set S of n points in the plane sorted in x direction and in y directions, we can find in linear time a point $p \in S$ which minimizes the sum of the L_1 distances to the points in S.*

We can extend this theorem to the case where the distance to be minimized is the sum of squared L_2 distances from a point to the rest of the points of S, since the separability property holds for this case as well. Assume we have computed $\{\sigma_1^x, \ldots, \sigma_n^x\}$ above and let $\tau_i^x = \sum_{j=1}^n (x_j - x_i)^2$. The recursion formula for computing all the squared x-distances is easily computed to be

$$\tau_i^x = \tau_{i-1}^x - 2\delta\sigma_{i-1}^x - n\delta^2$$

where the x-distance between p_{i-1} and p_i is δ.

Corollary 3.2.2 *Given a set S of n points in the plane, sorted in x direction and in y direction, we can find in linear time a point p of S which minimizes the sum of squared L_2 distances to the points in S.*

3.2.2 The general case

We turn to the discrete min-sum problem for $1 \leq k \leq n-1$. We describe the algorithm for the L_∞ metric. It has two phases. In the **first phase** we find, for each point $p_i \in S$, the smallest square R_i centered at p_i which contains at least $k+1$ points of S. We also get the *square size* λ_i which is defined as half the side length of R_i. In the **second phase** we compute for each p_i, $i = 1, \ldots, n$, the sum of the distances from it to the points of S in R_i and pick i for which this sum is minimized.

For the **first phase** we apply a simple version of parametric searching. Assume $q = (q_x, q_y) \in S$ is the query point for which we want to find the smallest square R which contains at least $k+1$ points of S. For a parameter λ, denote by $R(\lambda)$ a square of size λ centered at q. We test whether $R(\lambda)$ contains at least $k+1$ points of S by applying Chazelle's [28, 29] orthogonal range counting. Namely, given a set of n points in the plane and an orthogonal range, find the number of points contained in the range. Chazelle proposes a data structure that can be constructed in time $O(n \log n)$ and occupies $O(n)$ space, such that a range-counting answer for a query region can be answered in time $O(\log n)$.

Clearly the minimum value of λ is the distance from the query point to its k^{th} nearest neighbor. Thus candidate values for λ are $|q_x - p_x|$ and $|q_y - p_y|$ for all $p = (p_x, p_y) \in S$. By performing a binary search in the sets $\{p_x \mid p \in S, p_x > q_x\}$, $\{p_x \mid p \in S, p_x < q_x\}$, $\{p_y \mid p \in S, p_y > q_y\}$ and $\{p_y \mid p \in S, p_y < q_y\}$, we find the smallest λ such that $R(\lambda)$ contains at least $k+1$ points of S.

Lemma 3.2.3 *Given a set S of n points and a positive integer $k < n$. We can find for each point $p_i \in S$ the smallest square centered at p_i that contains at least $k+1$ points of S in total time $O(n \log^2 n)$.*

In the **second phase** we compute, for each point $p_i \in S$, the sum of distances from p_i to its k nearest neighbors, namely, the points of S which are contained in R_i. In order to compute efficiently the sums of distances in all the squares R_i, we apply the orthogonal range searching algorithm for weighted points of Willard and Lueker [102] which is defined as follows. Given n weighted points in d-space and a query d-rectangle Q, compute the accumulated weight of the points in Q. The data structure in [102] is of size $O(n \log^{d-1} n)$, it can be constructed in time $O(n \log^{d-1} n)$, and a range query can be answered in time $O(\log^d n)$. We show how to apply their data structure and algorithm to our problem.

Let $q \in S$ be the point for which we want to compute the sum of distances from it to its k nearest neighbors. Let R be the smallest square found for q in the first phase. Clearly R can be decomposed into four triangles by its diagonals such that the L_∞ distance between all points of S within one triangle is, wlog, the sum of x coordinates of the points of S in this triangle

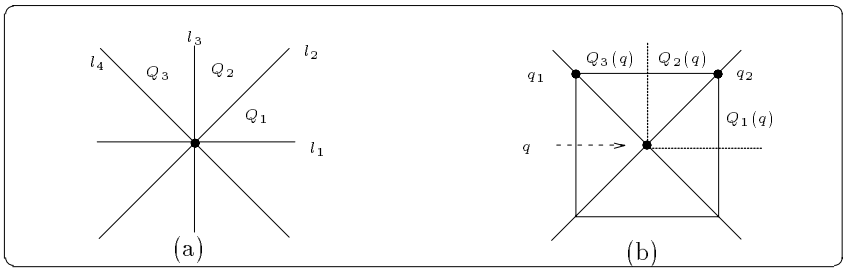

Figure 3.1: (a) The regions Q_i and (b) $Q_i(q)$

minus the x coordinate of q times the number of points of S in this triangle. More precisely, let Δ_u be the closed triangle whose base is the upper side of R and whose apex is q. Denote by σ_u the sum of the L_∞ distances between the points in Δ_u and q, and by N_u the number of points of $S_u = \{S - q\} \cap \Delta_u$, then

$$\sigma_u = \sum_{p_j \in S_u} p_j^y - q_y \cdot N_u.$$

Our goal in what follows is to prepare six data structures for orthogonal range search for weighted points, as in [102], three with the weights being the x coordinates of the points of S and three with the y coordinates as weights, and then to define orthogonal ranges, corresponding to the triangles in R for which the sums of x (y) coordinates will be computed.

We proceed with computing σ_u. Let l_1 be the x axis, l_2 be a line whose slope is 45° passing through the origin, l_3 be the y axis and l_4 a line whose slope is 135° passing through the origin. These lines define wedges (see Figure 3.1 (a)): (1) Q_1–the wedge of points between l_1 and l_2 whose x coordinates are larger than their y coordinates, (2) Q_2–the wedge of points between l_2 and l_3 whose y coordinates are larger than their x coordinates, and (3) Q_3–the wedge of points between l_3 and l_4 whose y coordinates are larger than their x coordinates.

Each of these wedges defines a data structure, as in [102]. Observe, *e.g.*, the wedge Q_1. We transform l_1 and l_2 into corresponding axes of an orthogonal coordinate system, and apply the same transformation on all the points $p_i \in S$. We construct the orthogonal range search data structure for the transformed points with the original y coordinates as weights. (Similarly we construct data structures for the points of S transformed according to Q_2 and Q_3, respectively, for the y sums, and another set of three data structures for the x sums.)

We denote by $Q_i(q)$ the wedge Q_i translated by q. Denote by $Y_i(q)$ the sum of the y coordinates of the points of S in $Q_i(q)$, $i = 1, 2, 3$. Then

$$\sum_{p_j \in S_u} p_j^y = (Y_2(q) + Y_3(q)) - (Y_2(q_1) + Y_3(q_1)) - Y_1(q_1) + Y_1(q_2),$$

where $q_1 = (q_x - \lambda, q_y + \lambda)$ and $q_2 = (q_x + \lambda, q_y + \lambda)$ (see Figure 1(b)). If the segment $[q_1, q_2]$ contains points of S we define q_1 and q_2 as $q_1 = (q_x - \lambda - \epsilon, q_y + \lambda + \epsilon)$ and $q_2 = (q_x + \lambda + \epsilon, q_y + \lambda + \epsilon)$ for some sufficiently small $\epsilon > 0$.

To compute N_u we can use the same wedge range search scheme, but with unit weights on the data points (instead of coordinates). In a similar way we compute the sum σ_d for the lower triangle in R (σ_l and σ_r for the left and right triangles in R respectively) and the corresponding number of points N_d (N_l and N_r).

It is possible that R contains more than $k+1$ points – this happens when more than one point of S is on the boundary of R. Our formula for the sum of the L_∞ distances should be

$$D = \sigma_u + \sigma_d + \sigma_l + \sigma_d - \lambda \cdot (N_u + N_d + N_l + N_r - k - 1).$$

Hence, the second phase of the algorithm, requires $O(n \log n)$ preprocessing time and space, and then $O(\log^2 n)$ query time per point $p_i \in S$ to determine the sum of distances to its k nearest points. Thus, for both phases, we conclude

Theorem 3.2.4 *The discrete min-sum problem in the plane for $1 \leq k \leq n-1$ and under L_∞-metric, can be solved in time $O(n \log^2 n)$ occupying $O(n \log n)$ space.*

3.3 Desirable Facility Location Problem (p9) - continuous case

The continuous desirable facility location problem is defined as follows. Given a set S of n points and a parameter $1 \leq k \leq n-1$. Find a point c in the plane such that the sum of distances from c to its k nearest points from S is minimized. We consider the problem where the distances are measured by the L_1 metric.

We create a grid M by drawing a horizontal and a vertical line through each point of S. Assume the points of S are sorted according to their x coordinates and according to their y coordinates. Denote by $M(i,j)$ the grid point that was generated by the i^{th} horizontal line and the j^{th} vertical line in the y and x orders of S respectively. Bajaj [17] observed that the solution to the *continuous* min-sum problem with $k = n-1$ should be a grid point. As a matter of fact it has been shown that for this problem the point $M(\lfloor n/2 \rfloor, \lfloor n/2 \rfloor)$ is the required point. (Where for an even n the solution is not unique and there is a whole grid rectangle whose points can be chosen as the solution.)

For $k < n-1$, we can pick the solution from $O((n-k)^2)$ grid points, since the smallest x-coordinate that c might have is $x_{\lfloor k/2 \rfloor}$, and the largest $x_{n-\lceil k/2 \rceil}$

(similarly for y). This is true since in the extreme case where all the k points are the lowest leftmost points then according to Bajaj the solution to this k points problem is at $M(\lfloor k/2 \rfloor, \lfloor k/2 \rfloor)$. Similarly if the k points are located at any other corner of M. Thus we remain with $(n-k+1)^2$ grid points which are candidates for the solution c. Applying the discrete algorithm of Section 3.2.2, with the query points being the candidate solutions, we obtain the following theorem.

Theorem 3.3.1 *The continuous min-sum problem can be solved in* $(n \log n + (n-k)^2 \log^2 n)$ *time for any positive* $k \leq n-1$.

3.4 Facilities in Regions (p10)

We consider several instances of the following generally-stated problem, which has several applications in, e.g., urban, industrial and military task planning. **Placing Obnoxious Facilities:** Let S be a set of n points in the plane (called *demand* points), and let $\mathcal{R}$ be a set of m, $m \leq n$, regions in the plane (called *neighborhoods*). Let k be a positive integer (k is the number of facilities, e.g., garbage dumps, to be placed). Find k sites $c_1, \ldots, c_k$ for the k facilities, such that (i) $C = \{c_1, \ldots, c_k\}$ is a *piercing set* for $\mathcal{R}$, that is, each of the neighborhoods in $\mathcal{R}$ is served by at least one facility that is located in the neighborhood. (ii) The minimal distance between a demand point in S and a site in C is maximized.

This problem belongs to a class of problems that deal with the location of facilities, both desirable and undesirable, under various conditions. This class of problems occupies researches in operations research, especially in the field of *location science*. Some of the more geometric problems have also been treated in the computational geometry literature. In a typical facility location problem, we need to find a location for some facility, with respect to a given set of demand sites. Both the demand sites and the facility are represented as points in the plane. The chosen location should satisfy a given set of conditions, e.g., minimize the maximal distance to a demand site (known as the 1-center problem). Our problem is somewhat more complex (though definitely realistic) than most of the related problems. There are several facilities, and the desired locations must satisfy both a piercing condition and a distance optimization condition. Notice that if the domain of possible locations for the facility is the entire plane, then the problem becomes impractical and not interesting. Therefore, some constraints on the location of the facility should be specified, e.g, forcing it to lie in some bounded region R.

We assume that the regions in $\mathcal{R}$ are unit axis-parallel squares (actually, translated copies of some axis-parallel rectangle). We consider both the L_∞ case and the L_2 case. In the L_∞ case (resp. L_2 case), we seek the maximal value d^* for which there exist k locations such that (i) none of the locations lies in the interior of a square of edge length $2d^*$ (resp. in a disk of radius d^*) centered at a demand point, and (ii) for each of the squares $R \in \mathcal{R}$, at least one of the k locations lies in R (in other words, the k locations consist of a k-piercing set for the set of squares $\mathcal{R}$). We present efficient solutions for $k = 1, 2$, both under the L_∞ metric and under the Euclidean metric. Our solutions consist of solutions to the corresponding decision problems to which we apply either the sorted matrices technique of Frederickson and Johnson [54] or the parametric searching technique of Megiddo [79] to obtain the maximal value d^*. For $k \geq 3$ we show two examples which in some sense imply that there is not much hope for a subquadratic solution for $k = 3$ or for any other value of k greater than 3. In addition, we also present a solution to

the weighted version of the simplest problem ($k = 1$ under L_∞), and present a lower bound for its corresponding decision problem (and for the decision problem corresponding to the unweighted version).

Consider for example the decision version of the problem in which we need to place two facilities, under the Euclidean metric. In this problem, we need to consider all possible solutions to the 2-piercing problem for $\mathcal{R}$. For each such solution p_1, p_2, we must check whether both p_1 and p_2 are not covered by the n disks of radius d centered at the points of S. By adapting a lemma of Katz et al. [69], and employing, in a sophisticated way, a technique due to Sharir [97] that resembles searching in monotone matrices, we transform the problem into the following *reception problem* (with some additional issues that require attention): Given m transmitters, each of range d (e.g., the transmitters of some communication system), construct a compact data structure that supports coverage queries, i.e., determine whether a query rectangular region is fully covered by the transmitters. In other words, preprocess a set of m congruent disks, so that, given a query rectangle R, one can quickly determine whether R is fully contained in the union of the disks. We present a simple, though non-trivial, solution to this problem, and to the problem where the query regions are constant-size polygons instead of rectangles. We are not aware of any previous solution to these problem. Our solution uses the Voronoi diagram of the centers of the disks and data structures for orthogonal (alternatively, simplex) range searching, and vertical (alternatively, general) ray-shooting among line segments. The construction time is nearly linear for both rectangular queries and polygonal queries, the space required is linear, and the query cost is $O(\log n)$ for rectangular queries and roughly $O(n^{1/2})$ for polygonal queries.

As was pointed in the previous section the problem in which the n demand points lie in a simple polygon R with at most n vertices, and one needs to place a single facility in R, such that the minimal Euclidean distance between a demand point and the facility is maximized, was solved in $O(n \log n)$ time by Bhattacharya and Elgindy [23]. If the polygon R is a rectangle, and each of the demand points p_i is assigned a weight w_i, so that the distance between p_i and a point $q \in R$ is w_i times the L_∞ distance between them, then one can apply the $O(n \log^4 n)$ solution of Follert et al. [52]. The latter bound was improved recently to $O(n \log^2 n)$ runtime algorithm by us [21], where we actually consider a slightly more general problem. In our problem each demand point p_i is assigned two weights w_i^x and w_i^y, and the distance between p_i and q is $w_i^x |p_i^x - q^x| + w_i^y |p_i^y - q^y|$. The algorithm is described in section 3.1. Brimberg and Mehrez [24] solve the following problem: Find k locations in the rectangle R (for k facilities), such that (i) the distance between any two locations is at least some given value d, and (ii) the minimal distance between a demand point and a facility is at least some given value r. Katz et al. [70] present improved solutions to this problem.

3.4.1 The Reception Problem

We consider the following problem: Given n transmitters, each of range r, construct a compact data structure that supports coverage queries, i.e., determine whether a query (rectangular or polygonal) region is fully covered by the transmitters. In other words, preprocess a set of n congruent disks for coverage queries, i.e., determine whether a query region R is fully contained in the union of the disks.

We first present a solution for polygonal region queries, that is, we assume the query regions are simple polygons with at most c vertices, for some constant c. Then we show how the bounds for the preprocessing time and query cost can be improved if the query regions are axis-parallel rectangles.

Polygonal region queries

In this subsection we deal with the following problem: Given a set D of n unit disks in the plane, construct a compact data structure, so that, given a query polygon Q, one can quickly determine whether Q is fully covered by the disks in D. Let S denote the set consisting of the center points of the disks in D. The main components of our data structure are (i) the Voronoi diagram VD of S and a corresponding point location data structure, (ii) a data structure for simplex range searching over a subset of the vertices of VD, and (iii) a data structure for ray shooting over a set of portions of edges of VD.

The preprocessing phase. The preprocessing phase consists of the following three steps:

1. Construct the Voronoi diagram VD of S, and the corresponding point location data structure, both in $O(n \log n)$ time [20].

2. Compute the set V' of all vertices of VD that are not covered by disks in D, by checking, for each vertex v of VD, whether the distance between v and its corresponding Voronoi sites is greater or equal to 1. Construct in $O(n^{1+\varepsilon})$ time a linear-size data structure for simplex range searching queries over V' [77].

3. Compute the set E' of the portions of the edges of VD that are not covered by disks in D. According to the claim below, the size of E' is only $O(n)$, and it can be computed in $O(n)$ time. Construct in $O(n^{1+\varepsilon})$ time a linear-size data structure for ray shooting over E' [4].

Claim 3.4.1 *The number of edge portions in E' is $O(n)$, and E' can be computed from VD in $O(n)$ time.*

Proof. Let e be an edge of VD, and let $p \in S$ be one of the two corresponding Voronoi sites. Notice that if a point on e is covered by one or more of the

disks in D, then this point is necessarily covered by the disk centered at p. Thus, in order to determine the portions of e that are not covered by D, it is enough to consider the disk centered at p, ignoring all other disks. The number of such portions is clearly at most two. ∎

Answering queries. A query with a polygon Q is treated as follows. We first partition Q into a constant number of triangles. (Recall that Q has at most c vertices, for some constant c.) For each of the triangles Δ, we perform a range searching query using the simplex range searching data structure. If the answer obtained to one (or more) of these queries is positive, i.e., one (or more) of these triangles contains points of V', then we conclude that Q is not fully covered by the disks in D, return "NO" and stop. Otherwise, we proceed as follows. For each edge $e = \overline{ab}$ of Q:

- Find the cell C_a (resp. C_b) of VD containing the endpoint a (resp. b) of e, using the point location data structure.
- Calculate the distance d_a (resp. d_b) between a (resp. b) and the point of S defining C_a (resp. C_b). If $d_a \geq 1$ (resp. $d_b \geq 1$), return "NO" and stop.
- If a and b do not lie in the same cell of VD (i.e., if $C_a \neq C_b$), then perform a ray shooting query with the ray emanating from a and containing e, using the ray shooting data structure. If the answer obtained is positive and the hitting point is on e, return "NO" and stop.

If we have reached this point, we may conclude that Q is fully covered by the disks in D and return "YES".

The algorithm for answering a query is quite simple, however, it is not obvious that it is correct. In the following theorem we prove that it is indeed correct, that is, it returns "YES" if the query polygon Q is fully covered by the disks in D, and "NO" otherwise.

Theorem 3.4.2 *The algorithm is correct; it returns "YES" if the query polygon Q is fully covered by the disks in D and "NO" otherwise.*

Proof. Consider a point ψ in Q, that lies in the Voronoi cell of p. ψ is covered by D if and only if it is covered by the disk centered at p. Moreover, the disk centered at p is "responsible" for covering the region $Q \cap C_p$, where C_p is the Voronoi cell of p. It is therefore enough to verify that the point in $Q \cap C_p$ that is the furthest from p, is at distance less than 1 from p. This point, however, is either a vertex of the boundary of C_p, or an intersection point between an edge of the boundary of C_p and the boundary of Q, or a vertex of Q. The range searching queries performed in the first part of the algorithm take care of points of the first kind; by the answers obtained, we

immediately know whether there exist vertices of VD that lie in Q and are not covered. The second part of the algorithm takes care of points of the two other kinds, by checking whether the boundary of Q is fully covered by D. Let e be an edge of the boundary of Q. We first verify that both its endpoints are covered. If both endpoints lie in the same Voronoi cell and they are both covered, then clearly the entire edge is covered. However, if the endpoints lie in different Voronoi cells, it is possible that portions of the interior of e are not covered. However, this can happen if and only if e intersects a segment of E', and this will be detected by the ray shooting query performed for e. ∎

Concerning the complexity of the above algorithm, the preprocessing time is $O(n^{1+\varepsilon})$, which is the time required to construct the range searching and ray shooting data structures [4, 77], the space complexity is $O(n)$, and the query cost, determined by the range searching and ray shooting queries, is $O(n^{\frac{1}{2}+\varepsilon})$. We thus obtain:

Theorem 3.4.3 *Let D be a set of n unit disks in the plane. It is possible to preprocess D in time $O(n^{1+\varepsilon})$, into a linear-size data structure, such that determining whether a constant-size query polygon Q is fully covered by the disks in D can be done in time $O(n^{\frac{1}{2}+\varepsilon})$.*

Remark 3.4.4 *As known, the n^{ε} factors in the theorem above can be replaced by slightly smaller factors. Also the standard storage/query tradeoff can be applied to construct a data structure of size $n \leq m \leq n^2$ with query time $O(n^{1+\varepsilon}/m^{1/2})$. In particular, if the query polygons are of linear size, then we can construct a data structure of size and query cost roughly $O(n^{4/3})$.*

Rectangular region queries

In this subsection we consider the special case where the query regions are axis-parallel rectangles. This is the case to which we refer in Section 3.4.2. For this case, we can obtain better bounds for the preprocessing time and query cost, by replacing the general range searching and ray shooting data structures with standard specialized data structures. More precisely, we use a data structure for orthogonal range searching over the set V' [28], and a data structure for horizontal/vertical ray shooting over the set E' [20]. We thus obtain:

Theorem 3.4.5 *Let D be a set of n unit disks in the plane. It is possible to preprocess D in time $O(n \log n)$, into a linear-size data structure, such that determining whether a query rectangle Q is fully covered by the disks in D can be done in time $O(\log n)$.*

Remark 3.4.6 *The bounds for the somewhat simpler problem, where D is a set of unit axis-parallel squares instead of unit disks remain the same.*

3.4.2 Obnoxious Facilities

In this subsection we solve several instances of the following problem.
Placing Obnoxious Facilities: Let S be a set of n points in the plane (called *demand* points), and let $\mathcal{R}$ be a set of m, $m \leq n$, regions in the plane (called *neighborhoods*). Let k be a positive integer (k is the number of facilities, e.g., garbage dumps, to be placed). Find k locations $C = \{c_1, \ldots, c_k\}$ for the k facilities, such that (i) C is a *piercing set* for $\mathcal{R}$, that is, each of the neighborhoods in $\mathcal{R}$ is served by at least one facility that is located in the neighborhood. (ii) The minimal distance between a demand point in S and a location in C is maximized.

In all the problem instances that we consider, the set of regions $\mathcal{R}$ consists of unit axis-parallel squares. We consider the two problems in which the number of facilities k is one or two, respectively, under the L_∞ metric as well as under the Euclidean metric. For $k \geq 3$ we show two examples which in some sense imply that there is not much hope for a subquadratic solution for $k = 3$ or for any other value of k greater than 3. Obviously, if the set $\mathcal{R}$ is not k-pierceable, then there is no solution. Therefore, we assume that $\mathcal{R}$ is k-pierceable. We can check whether $\mathcal{R}$ is k-pierceable, $1 \leq k \leq 2$, in $O(m)$ time [99].

When solving a problem, we first present a solution to the corresponding decision problem, and then apply the sorted matrices technique of Frederickson and Johnson [54] or the parametric searching technique of Megiddo [79] to obtain a solution to the original problem. That is, we first solve a problem of the form: Determine whether there exist k locations, such that, for each of the m unit squares $r \in \mathcal{R}$, at least one of these locations is in r, and the minimal distance between the n demand points and these locations is at least d, where d is a parameter of the problem. We then apply one of the above techniques to obtain the maximal value d^* for which the decision problem returns a positive answer.

The L_∞ case

$k = 1$

In this problem we wish to place only one facility. This problem is relatively easy, and we present a solution to the weighted version of the problem as well. In the weighted version of the problem, each point $p_i \in S$ has two weights associated with it: $w_1(p_i)$ and $w_2(p_i)$. Solving the decision problem for d, each point $p_i \in S$ defines a *forbidden* region F_i, where the facility may not reside. In the unweighted version F_i is a square of edge length $2d$ centered at p_i, and in the weighted version F_i is the rectangle

$$\{q \in \mathbb{R}^2 \mid d_x(q, p_i) < d \cdot w_1(p_i) \ , \ d_y(q, p_i) < d \cdot w_2(p_i)\} \ .$$

Let U denote the union of the forbidden regions $F_1, \ldots, F_n$. Let $R = \cap\mathcal{R}$. R is a non-empty rectangle (since, by assumption, $\mathcal{R}$ is 1-pierceable). An allowed location for the facility exists if and only if U does not completely cover R. Since it is possible to determine whether a set of n rectangles covers another rectangle R, using a segment tree, in $O(n \log n)$ time, we obtain an $O(n \log n)$ solution to the decision problem in both the unweighted and weighted versions.

In the unweighted version, we can apply the sorted matrices technique of Frederickson and Johnson [54] to obtain in $O(n \log^2 n)$ time the maximal value d^* for which the corresponding decision problem returns "YES". This is based on the observation that d^* is either the x-distance (y-distance) between two points in S, or the x-distance (resp. y-distance) between a point in S and a vertical (resp. horizontal) edge of R. Notice that the number of distances of the latter kind is only $O(n)$, so we can compute them explicitly, sort them, and perform a binary search on the sorted list.

In the weighted version, we apply the parametric search paradigm of Megiddo [79] to obtain in $O(n \log^3 n)$ time the maximal value d^* for which the corresponding decision problem returns "YES". (A parallel algorithm for the decision problem is presented in [21]; it employs $O(n \log n)$ processors and computes the answer in $O(\log n)$ time.) We thus obtain:

Theorem 3.4.7 *Under the L_∞ metric and for $k = 1$, the problem can be solved in $O(n \log^2 n)$ time, and the weighted version of the problem can solved in $O(n \log^3 n)$ time. The corresponding decision problems can be solved in $O(n \log n)$ time.*

A lower bound. We obtain a lower bound of $\Omega(n \log n)$ for the decision problem (in both versions), by showing that even the one-dimensional version of the problem "determine whether a set of n unit squares covers another square" has a lower bound of $\Omega(n \log n)$. Consider the GAP-EXISTENCE problem: Given a set A of n real numbers $A = \{a_1, \ldots, a_n\}$, determine whether there exist two consecutive numbers in the sorted sequence obtained from A, such that the difference between them is greater than 1. Sharir and Welzl [99] observed that this problem has a lower bound of $\Omega(n \log n)$. We transform a_i, $i = 1, \ldots, n$, to the one-dimensional rectangle $[a_i, a_i + 1]$, thus obtaining a set $\mathcal{R}$ of n rectangles. We define $R = [\min_{a_i \in A} a_i \ , \ \max_{a_i \in A} a_i]$. It is clear that R is not covered by the rectangles in $\mathcal{R}$ if and only if there exist two consecutive numbers as above.

$k = 2$

Assuming $\mathcal{R}$ is 2-pierceable (but not 1-pierceable), we need to determine whether there exist two points p_1, p_2, such that $\{p_1, p_2\}$ is a piercing pair for $\mathcal{R}$, and neither p_1 nor p_2 lie in the interior of U, where U is the union of the

squares of edge length $2d$ centered at the points of S. Assume $\{p_1, p_2\}$ is a piercing pair for $\mathcal{R}$, then we can divide the squares in $\mathcal{R}$ into two disjoint subsets $\mathcal{R}_{p_1}$ and $\mathcal{R}_{p_2}$, such that $p_1 \in R_1 = \cap\mathcal{R}_{p_1}$ and $p_2 \in R_2 = \mathcal{R}_{p_2}$. (If some of the squares in $\mathcal{R}$ are pierced by both p_1 and p_2, then there are many ways to do this.) Therefore, we could search for a "good" piercing pair $\{p_1, p_2\}$ (i.e., a piercing pair such that both points are not in the interior of U) by considering all possible partitions of $\mathcal{R}$ into two subsets $\mathcal{R}_1, \mathcal{R}_2$ such that the rectangles $R_1 = \cap\mathcal{R}_1$ and $R_2 = \cap\mathcal{R}_2$ are non-empty. For each such partition, we would have to check for each of the corresponding two rectangles whether it contains a point that is not covered by U. However, this method is very inefficient. Fortunately, we have a claim 2.3.1 that allows us to restrict our search to a quadratic number of partitions. Denote by $X_\mathcal{R}$ the centers of the squares in $\mathcal{R}$, sorted by their x-coordinate (left to right), and by $Y_\mathcal{R}$ the centers of the squares in $\mathcal{R}$, sorted by their y-coordinate (low to high). The claim 2.3.1 gives to us the efficient way to find the piercing pair. More precisely, if p_1 and p_2 are a piercing pair for $\mathcal{R}$, then $\mathcal{R}$ can be divided into two subsets $\mathcal{R}_1$ and $\mathcal{R}_2$, $p_1 \in \cap\mathcal{R}_1$, $p_2 \in \cap\mathcal{R}_2$, such that $\mathcal{R}_1$ can be represented as the union of two subsets $\mathcal{R}_1^x$ and $\mathcal{R}_1^y$ (not necessarily disjoint, and one of them might be empty), where the centers of squares of $\mathcal{R}_1^x$ form a consecutive subsequence of the list $X_\mathcal{R}$, starting from its beginning, and the centers of squares of $\mathcal{R}_1^y$ form a consecutive subsequence of $Y_\mathcal{R}$, starting from the list's beginning.

According to this it is enough to consider partitions in which one of the subsets is obtained by taking the i_x leftmost squares in $\mathcal{R}$ together with the i_y bottommost squares in $\mathcal{R}$, $0 \leq i_x, i_y \leq n$. (The claim 2.3.1 is proven under the assumption that the piercing points p_1 and p_2 are centers of squares in $\mathcal{R}$. However, it is easy to see that the claim is also true without this assumption.)

We further restrict our search by employing a technique, due to Sharir [97], that resembles searching in monotone matrices; for a recent refinement of this technique and applications see [38, 69] and also section 2.3.1. Let M be an $(m+1) \times (m+1)$ matrix, whose rows (skipping row 0) correspond to $X_\mathcal{R}$ and whose columns (skipping column 0) correspond to $Y_\mathcal{R}$. An entry M_{xy} in the matrix is defined as follows. Let D_x be the set of squares in $\mathcal{R}$ such that the x-coordinate of their centers is smaller or equal to x, and let D_y be the set of squares in $\mathcal{R}$ such that the y-coordinate of their centers is smaller or equal to y. Let $D_{xy}^l = D_x \cup D_y$ and $D_{xy}^r = (\mathcal{R} - D_{xy}^l)$, and let $R_{xy}^l = \cap D_{xy}^l$ and $R_{xy}^r = \cap D_{xy}^r$.

$$M_{xy} = \begin{cases} \text{`}YY\text{'} & \text{if } R_{xy}^r \not\subseteq U \text{ and } R_{xy}^l \not\subseteq U \\ \text{`}YN\text{'} & \text{if } R_{xy}^r \not\subseteq U \text{ but } R_{xy}^l \subseteq U \\ \text{`}NY\text{'} & \text{if } R_{xy}^r \subseteq U \text{ but } R_{xy}^l \not\subseteq U \\ \text{`}NN\text{'} & \text{if } R_{xy}^r \subseteq U \text{ and } R_{xy}^l \subseteq U \end{cases}$$

where we assume of course that the empty set is contained in U. It follows that the answer to our decision problem is "YES" if and only if M contains

a 'YY' entry.

In order to apply Sharir's technique the lines and columns of M^1 must be non-decreasing (assuming 'Y' > 'N'), and the lines and columns of M^2 must be non-increasing, where M^i is the matrix obtained from M by picking from each entry only the i'th letter, $i = 1, 2$. In our case this property clearly holds, since, for example, if for some x_0 and y_0, $M^1_{x_0,y_0}$ ='Y', then for any $x' \geq x_0$ and $y' \geq y_0$, $M^1_{x',y'}$='Y'. Thus we can determine whether the *implicit* matrix M contains an entry 'YY' by inspecting only $O(m)$ entries in M, advancing along a *connected* path within M [38]. For each entry along this path, we need to determine whether R^z_{xy} is fully covered by U, $z \in \{l, r\}$. This can be done in $O(\log n)$ time by dynamically maintaining the intersection $\cap D^z_{xy}$, and by utilizing the data structure of Section 3.4.1 (see Remark 3.4.6 there). Thus in $O(n \log n)$ time we can determine whether M contains a 'YY' entry.

Optimization. We show how to find the smallest value d^* for which the matrix M above contains a 'YY' entry. It is easy to verify that d^* is either (i) half the difference between the x-coordinates (alternatively, y-coordinates) of a pair of points in S, or (ii) the horizontal (respectively, vertical) distance between a vertical (respectively, horizontal) edge of a square in $\mathcal{R}$ and a point in S. All these potential values can be represented by four (implicit) sorted matrices; two matrices for each axis.

We define the two sorted matrices corresponding to the x-axis. Let L_x be the sorted list consisting of the x-coordinates of the points in S and the x-coordinates of the vertical edges of the squares in $\mathcal{R}$. Entry (i, j) in matrix M_1 stores the value $(x_j - x_i)/2$, where x_i, x_j are the i-th and j-th elements in L_x, and entry (i, j) in the matrix M_2 stores the value $(x_j - x_i)$. Clearly, these matrices contain several values that do not belong to the set of potential solutions, but this does not affect the running time. We define the two sorted matrices M_3 and M_4 corresponding to the y-axis analogously.

We now apply the Frederickson and Johnson technique [54] to each of the four matrices in order to find the smallest value in these matrices for which the decision algorithm returns "Yes". We thus obtain:

Theorem 3.4.8 *Under the L_∞ metric and for $k = 2$, the problem can be solved in $O(n \log^2 n)$ time. The corresponding decision problem can be solved in $O(n \log n)$ time.*

$k \geq 3$

The largest integer l for which there exists a linear-time algorithm that determines whether $\mathcal{R}$ is l-pierceable (and, if yes, computes an l-piercing set) is 3 [86, 94, 99]. This fact caused us to believe that claim 2.3.1 is also true for a piercing triplet. That is, if p_1, p_2, p_3 is a piercing triplet for $\mathcal{R}$, then $\mathcal{R}$ can be divided into three subsets, such that one of them can be represented as the union of two subsets as in claim 2.3.1. Unfortunately, we came up

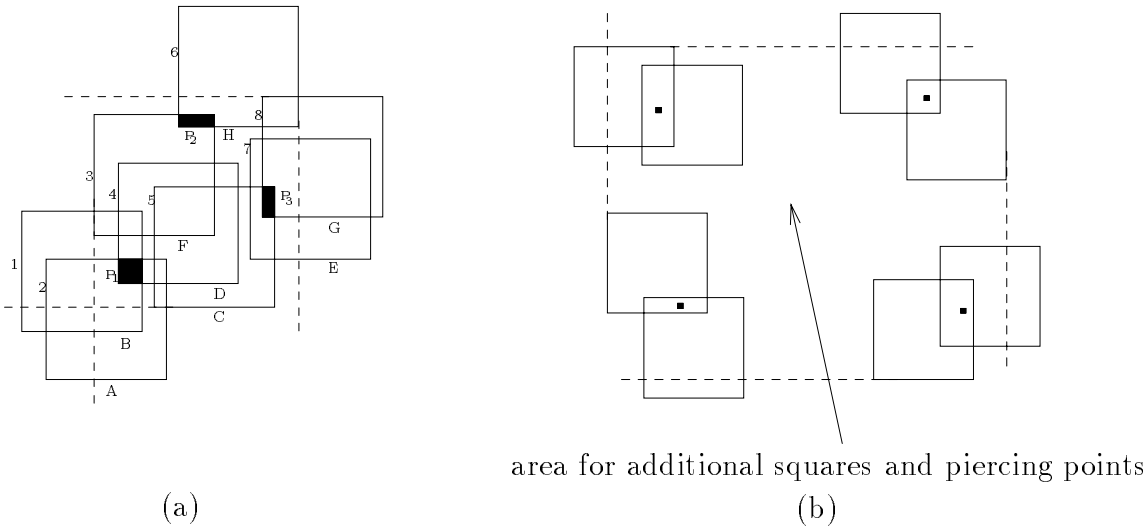

Figure 3.2: Claim 2.3.1 is false for $k \geq 3$

with a counterexample that is depicted in Figure 3.2(a). All piercing triplets for the 8 squares of Figure 3.2(a) must consist of a point from each of the three black rectangles, and it is easy to verify that we cannot divide the set of squares as required.

Figure 3.2(a) is not, however, a counterexample for $k \geq 4$. Since by adding squares and increasing the number of piercing points, the desired property might reappear. For completeness, we also provide a counterexample for $k \geq 4$ depicted in Figure 3.2(b). Assume that each of the four pairs in the figure lies near the corresponding corner of some huge square region s. Then we can add any number of squares around the middle of s and increase the number of piercing points accordingly, without ruining the counterexample.

The above counterexamples provide, in some sense, evidence that it is apparently impossible to obtain subquadratic solutions for $k \geq 3$.

The L_2 case

k=1

The corresponding decision problem is: Determine whether $R = \cap\mathcal{R}$ is completely covered by the n disks of radius d centered at the points of S. (Recall that, by assumption, $\mathcal{R}$ is 1-pierceable, and therefore $R \neq \emptyset$.) We can do this, using the result of Section 3.4.1, in $O(n \log n)$ time.

We apply the parametric searching technique to obtain in $O(n \operatorname{polylog} n)$ time the maximal value d^* for which the decision problem returns "YES"; we

omit the details from this version. (The problematic set of potential values is the one consisting of the radii of all circles that pass through three points in S. This set is also one of the sets of potential values in the planar 2-center problem, solved by Sharir [97] with parametric searching in $O(n \operatorname{polylog} n)$ time. Sharir's solution has been improved by Eppstein to randomized expected time $O(n \log^2 n)$ [48].) We thus obtain:

Theorem 3.4.9 *Under the L_2 metric and for $k = 1$, the problem can be solved in $O(n \operatorname{polylog} n)$ time. The corresponding decision problems can be solved in $O(n \log n)$ time.*

k=2

The decision algorithm is identical to the decision algorithm in the L_∞ case (Section 3.4.2), except for the component that deals with queries of the form: determine whether a query rectangle is fully contained in U. Now U is the union of n disks, each of radius d, so we use our solution to the Reception Problem with axis-parallel rectangular queries (Section 3.4.1). We obtain an $O(n \log n)$-time decision algorithm.

We apply the parametric searching technique in the same manner as above, to obtain in $O(n \operatorname{polylog} n)$ time the maximal value d^* for which the decision problem returns "YES".

Theorem 3.4.10 *Under the L_2 metric and for $k = 2$, the problem can be solved in $O(n \operatorname{polylog} n)$ time. The corresponding decision problems can be solved in $O(n \log n)$ time.*

Conclusion

The dual problem, where the facilities are "friendly" or desirable, is also interesting. For $k = 2$ and under the L_∞ (resp. L_2) metric, this problem (actually, its corresponding decision problem) becomes: Find a pair of points which serves as a piercing pair for both the set $\mathcal{R}$ of unit squares and the set of squares (resp. disks) of radius d centered at the demand points. In the L_∞ case, this can be done by simply finding in $O(n)$ time a piercing pair for the union of the two sets of squares. In the Euclidean case, we would like to employ claim 2.3.1 in a sophisticated way, as we did in Section 3.4.2. Here, we need to determine, for each pair of rectangles that is generated, whether the set of disks can be pierced by choosing a point in each of the two rectangles. This can be done apparently by adopting Sharir's solution to the (decision problem of the) planar 2-center problem [97] (see also [48]).

Chapter 4

k-point Problems

The problems considered in this chapter can be defined as follows. "Given a set S of n points in metric space and some positive integer k (which is usually between 1 and n) find some property of the set S that depends on k". We present a list of problems that we will deal with them and then describe separately the corresponding algorithms. All the solution based on the new framework based on posets [1]. The problems we consider in this chapter are: Given a set S of n points in the plane, and given an integer k,

p11: Find the smallest axis parallel rectangle (smallest perimeter or smallest area) that encloses exactly k points of S.

p15: Find the the $n-k-1$ farthest rectilinear neighbors (under L_∞ metric) to all points of S, where $\frac{n}{2} \leq k \leq n-1$. Thus we implicitly find (but do not report) the k nearest rectilinear neighbors to all points of S.

p16: Enumerate the k largest (smallest) rectilinear distances in decreasing (increasing) order.

p17: Given a distance $\delta > 0$, report all the pairs of points of S which are of rectilinear distance δ or less (more).

p19: Find the smallest "rectangular" axis-aligned (constrained or not constrained) ring that contains k ($k \geq \frac{n}{2}$) points of S. A *rectangular ring* is two concentric rectangles, the inner rectangle fully contained in the external one. As a measure we take the maximum width or area of the ring. By *constrained* we mean that the center of the ring is one of the points of S.

p19 and **p20:** Find the smallest constrained circular ring (or a sector of a constrained ring) that contains k ($k \geq \frac{n}{2}$) points of S.

p12: Given a number $k \geq \frac{n}{2}$, decide whether a query rectangle contains k points or less.

4.1 Rectangle with k points inside (p11)

Given a set S of n points in the plane and an integer k, we want to find the smallest axis-parallel rectangle (smallest in term of perimeter or area) enclosing exactly k points of S. This problem has been investigated by many researchers, some of whose results we cite below. They considered the problem for any $k \leq n$. Aggarwal et al. [3] present an algorithm which runs in time $O(k^2 n \log n)$ and uses $O(kn)$ space. Eppstein et al. [49] and Datta et al. [36] show that this problem can be solved in $O(n \log n + k^2 n)$ time; the algorithm in [49] uses $O(kn)$ space, while the algorithm in [36] uses $O(n)$ space. These algorithms are efficient for small k values, but become inefficient for large k's. Notice that for $k = n$ the smallest enclosing rectangle is trivially found in $O(n)$ time.

The algorithm we present below is more efficient than those cited above for k values in the range $\frac{n}{2} < k \leq n$. It is based on *posets* (partially ordered sets) [1] and runs in time $O(n + k(n-k)^2)$ and $O(n)$ space. When $k = n$ our algorithm runs in $O(n)$ time. We also extend our algorithm to higher dimensions and find the smallest axis-parallel box that contains k out of n given points in d-space, $d \geq 3$. This algorithm runs in time $O(n + k(n-k)^2 + d(n-k)^{2(d-1)})$ and occupies $O(dn)$ space. We assume that all the points of S are in general position, i.e., that no two points have the same coordinate in any axis. Finally, we shortly discuss slight improvements of other algorithms, when more efficiency is obtained by taking into account the size of k relative to n.

Remark 4.1.1 *Another algorithm that runs efficiently for large k values was presented by Matoušek [78]. It finds the smallest circle enclosing all but few of the given n points in the plane. Given a large integer $k \leq n$, his algorithm runs in time $O(n \log n + (n-k)^3 n^{\varepsilon})$ for some $\varepsilon > 0$.*

4.1.1 The Algorithm

First we describe an algorithm which finds the smallest enclosing rectangle that contains k x-consecutive points of S. The techniques used in this algorithm will be applied in our general algorithm, which is described afterwards.

Enclosing k x-consecutive points

Given S as above, we restrict the problem to finding the smallest rectangle that covers k points of S whose x coordinates are consecutive. The x coordinate of an uncovered point of S is either among the $n-k$ smallest x coordinates or the $n-k$ largest ones. We cannot afford to spend $O(n \log n)$ time on sorting the points of S according to their x coordinates, and therefore apply a partial order selection method (see Aigner [1]). A *poset* is a

partially ordered set of elements. Figure 4.1 below illustrates a poset R, where the largest $n-k+1$ points of S are sorted according to the order and the bottom $k-1$ points are known to be smaller but are not sorted. The construction of a poset $R \subset S$ containing the $n-k$ elements of S with the largest x coordinates is easy. One way of doing this is to put n items into a binary heap and perform $n-k$ remove-max operations. In this way we collect the $n-k$ largest elements in S into an oredered set R in total time of $O(n + (n-k)\log n)$. We use this binary heap to find the point $v \in S$ with the $(n-k+1)^{st}$ largest x value in S.

Let $L = S - R$; clearly v is the point with the largest x coordinate in L (denoted by $\max_x^L$). Denote by $x(v)$ $(y(v))$ the x- (y-) coordinate of v. We construct three binary heaps for L. They have k nodes each. We put the points of L into the heaps. The heap K_1 will be used to dynamically find the point with the smallest y coordinate in L (denoted by $\min_y^L$). The heap K_2 will be used to dynamically find the point with the largest y coordinate in L ($\max_y^L$), and D will help find the point with the smallest x coordinate in L ($\min_x^L$). Finding the initial values above involves $3 * (k-1)$ comparisons in the corresponding binary heaps.

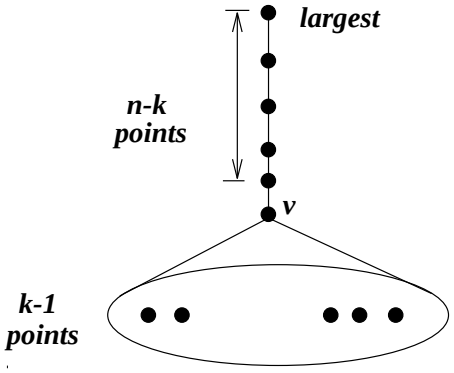

Figure 4.1: A poset

Finding the rectangle

We slide a sweepline from left to right, starting at the leftmost point r of S. At this point we compute the perimeter (area) of the rectangle defined by $\min_x^L$, $\max_x^L$, $\min_y^L$ and $\max_y^L$. The next event is to slide the sweepline to the next leftmost point of S: r is deleted from L, and v_1, the smallest point of R, is inserted into L, so that L always contains k points. The new $\max_x^L$ is $x(v_1)$. The next leftmost point in S is found using the binary heap D. This is the new $\min_x^L$. We update the binary heaps K_1 and K_2, deleting r and inserting v_1. Thus we get the updated, possibly unchanged, $\min_y^L$ and $\max_y^L$. Notice that we do not need to update D at all.

It is easily seen that each update takes $O(\log k)$ time, and the procedure is repeated $n-k$ times. Hence the total time involved in updates is $O((n-$

$k)\log k)$. The initial construction of K_1, K_2 and D, is performed in total time of $O(n + (n-k)\log n)$.

Summing up the runtimes of constructing the heap and all the updates, we get

Theorem 4.1.2 *The smallest rectangle that contains a given number k, $\frac{n}{2} < k \leq n$, of x-consecutive points in a set of n points in the plane, can be found in time $O(n + (n-k)\log n)$.*

The smallest rectangle containing k arbitrary points

To avoid tedious notations we assume that the names of the points correspond to their x-ordering, though this does not mean that the points are sorted. In general the outline of our algorithm is as follows: initially we fix the leftmost point of the rectangle to be the leftmost point of S. At the next *stage* the leftmost point of the rectangle is fixed to be the second left point of S, etc. Within one *stage*, of a fixed leftmost rectangle point, r, we pick the rightmost point of the rectangle to be the q'th x-consecutive point of S, for $q = k + r - 1, \ldots, n$. For fixed r and q the x boundaries of the rectangle are fixed to be the x-coordinates of r and q respectively, and we go over a small number of possibilities to choose the upper and lower boundaries of the rectangle so that it will enclose k points.

In more detail, we initially produce the posets R, D, K_1 and K_2 as in the former algorithm. We use them as before but with a slight modification to the maintenance of K_1 and K_2 as we describe below. We also use two auxiliary *sorted* lists A_1 and A_2 that are initially set to be empty. They will collect the information found throughout the algorithm, of the lowest points ($\min_y^L$) and highest points ($\max_y^L$), respectively. The maximum size of A_1 and A_2 is $n-k$ each. Since the lists A_1 and A_2 are short we can afford $O(n-k)$ time update operation on them (search, insert, delete). As before, D and R are not updated throughout the algorithm.

For the initial rectangle (say $r = 1$ and $q = k$) we compute the perimeter (area) of the rectangle by the initial $\min_x^L$, $\max_x^L$, $\min_y^L$ and $\max_y^L$. The point that attains $\min_y^L$ ($\max_y^L$) is stored as the first element in A_1 (A_2).

For the next step, r remains fixed and $q = k+1$, the vertical slab between r and q contains the first $k+1$ x-consecutive points. Trivially there are two rectangles R_1 and R_2 containing k of these points within this slab that are defined by the x boundaries at r and at q. The y boundaries of R_1 are the second smallest y in K_1 and the first largest in K_2, and of R_2 the first smallest y in K_1 and the second largest in K_2. The second values found in K_1 and K_2 are stored in A_1 and A_2 respectively. We compute the area (perimeter) of these two rectangles and check for minimum.

Letting q vary from $k+1$ to n, for each q we first update the data structures (see below) and then find the next smallest (largest) element in K_1 (K_2) and add it to the corresponding list A_1 (A_2). If $q = k + p$ then A_1 (A_2) has p entries, and we simply need to compute the areas of the rectangles bounded by r as $\min_x^L$, q as $\max_x^L$, and p $\min_y^L$ values from A_1 with their corresponding p $\max_y^L$ values from A_2.

Updating K_1 and A_2 upon varying q

1. If $y(q)$ is greater than the maximum y value in A_2, then no update of K_1 is required. This is because $y(q)$ will never get to act as $\min_y^L$ in the slab defined by r and q. We find the point with the maximum y value in A_2 by going over all its ($< n - k$) entries. We add q to A_2.

2. If $y(q) < \max(y)$ for the entries in A_2, then we can delete the point p which attains $\max(y)$ from K_1, and insert the point q into K_1. As in the former case p will not participate as a lower y boundary of a rectangle in this slab. We remove the point p from A_2.

We update K_2 and A_1 symmetrically. Each heap update takes $O(\log k)$ time, and a list update takes $O(n - k)$ time. The heaps K_1 and K_2 remain of size k.

For each new *stage* ($r' = r + 1$) we find the next smallest point (r') in S by removing the next minimal x point from the heap D. If r was in A_1 (A_2) we delete it. The heaps K_1 and K_2 undergo too many changes in stage r to be of any use at this stage. So we keep copies of the initial K_1 and K_2 from the previous stage r, and we only update them by deleting r and inserting $q = r + k - 1$ instead of r in the heaps. (These will serve as initial K_1 and K_2 at the next stage.) We continue as in stage $r = 1$, by incrementing q up to n and checking all the rectangles that contain k points between r and q. We finish when $r = n - k + 1$ and $q = n$.

It is easy to see that we check all the rectangles that contain k points. Not all the rectangle possibilities in the above algorithm yield feasible rectangles. See, e.g., in Figure 4.2, where the rectangle whose x boundaries are determined by r and q, and the y boundaries are defined by the corresponding p^{th} points in A_1 and A_2. Checking whether a rectangle is feasible or not is immediate and does not change the complexity of the algorithm.

We sum up the runtimes of all the components of the algorithm:

- Computing R and initially constructing the heaps: $O(n + (n-k)\log n)$
- Copying the heaps K_1 and K_2 and initially updating them per each stage is: $O(k)$. For all stages $O(k(n - k))$.
- Total time for updating K_1, K_2, A_1 and A_2, for all the steps in one stage: $O((n-k)((n-k)+\log k))$. Summing up to $O((n-k)^2 \log k + (n-k)^3)$ for all the $n - k$ stages.

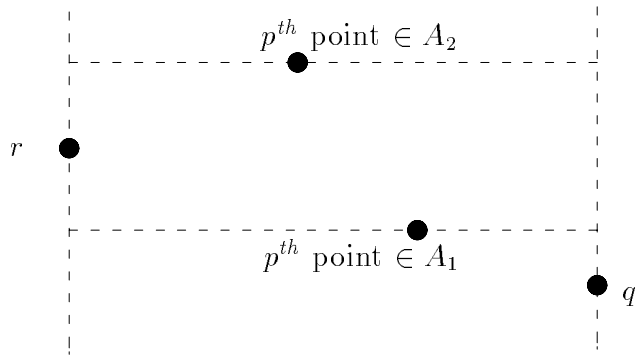

Figure 4.2: An infeasible rectangle

- The number of possible rectangles at each stage is bounded by the number of rectangles in the first stage: $\Sigma_{j=1}^{n-k} j = O((n-k)^2)$. Knowing A_1 and A_2 we invest $O(1)$ time in computing the area (perimeter) of each rectangle. The number of possible rectangles at all stages: $O((n-k)^3)$.

Since $k > n/2$ some of the above summands can be neglected and we yield

Theorem 4.1.3 *The smallest rectangle that contains a given number k, $\frac{n}{2} < k \leq n$, of points from a set of n points in the plane can be found in time $O(n + k(n-k)^2)$ and $O(n)$ space.*

The d-dimensional algorithm

We extend the planar algorithm to the smallest box containing k points in the d-dimensional space. Assuming we have an algorithm A_{d-1} for solving the $(d-1)$-dimensional problem in time T_{d-1}. Then the d-dimensional algorithm is as follows. We project S on all the $(d-1)$-dimensional hyperplanes, call these sets $S_1, \ldots, S_d$. We describe an algorithm for only one set, say S_i. (The whole process will be later repeated similarly for all $S_j, 1 \leq j \leq d$.)

1. We use the algorithm A_{d-1} to find all the $(d-1)$-dimensional boxes that contain k to n points of S_i.

2. For each box found in the former step we use the i^{th} axis to bound exactly k points of S in the d-dimensional box which is the cross of the $(d-1)$-dimensional box and a segment in the i axis (like we treated the y axis in the 2-dimensional problem).

It can be easily verified that the runtime of this algorithm is $(n-k)^2 T_{d-1}$. Thus we conclude by theorem,

Theorem 4.1.4 *The smallest box that contains a given number k, $\frac{n}{2} < k \leq n$, of points from a set of n points in d-space ($d \geq 3$) can be found in time $O(n + dk(n-k)^{2(d-1)})$ and $O(dn)$ space.*

4.1.2 Slight improvements of other algorithms

We achieve improvements on runtimes of other problems that deal with some k-set problems under the L_∞ metric. For example, an algorithm for finding the minimum L_∞ diameter of a $k-$point subset of a set of n points in the plane is described in [49]. It runs in time $O(n \log^2 n)$. This algorithm can be improved to run in $O(n \log n \log (n-k))$ time for $k > \frac{n}{2}$. Eppstein and Erickson [49] use an $O(n \log n)$ time algorithm for placing a fixed-size axis-aligned square and then apply the technique of sorted matrices for the optimization step [54]. Applying our techniques we can solve the problem by dealing only with $(n-k)^2$ distances along each coordinate axis, instead of $O(n^2)$ distances as [49] do. Searching over this matrix adds a factor of $O(\log (n-k))$ instead of $O(\log n)$.

Recently, Glozman et al. [57] gave a simple algorithm for a problem posed (and solved) by Salowe [91]: Given a set S of n points in the plane, they [57, 91] determine, in time $O(n \log^2 n)$, which pair of points of S defines the k^{th} distance (smallest or largest) under the L_∞ metric. Both papers have the same decision algorithm, but for the optimization step [91] apply parametric search, while [57] apply sorted matrices. For $k \leq \frac{n}{2}$ it is enough to keep in the optimization matrix only $O(k^2)$ distances on each coordinate axis instead of all the $O(n^2)$. Thus the optimization will add only a factor of $O(\log k)$ instead of $O(\log n)$ as in [57].

4.2 Rectilinear nearest neighbors (p15)

The problem is: Find the the $n-k-1$ farthest rectilinear neighbors to all points of S, where $\frac{n}{2} \leq k \leq n-1$. Thus we implicitly find (but do not report) the k nearest rectilinear neighbors to all points of S. We will use the technique prom previous section and also described in [95].

We define the nearest x-neighbor of a point $p_i \in S$ as point $q \in S$, such that $|x(p_i) - x(q)| = \min\{|x(p_i) - x(p)|, p \in S, p \neq p_i\}$, where $x(p)$ is the x-coordinate of p. First we find the k nearest x-neighbors for each point of S. To solve this subproblem we find the points with the $n-k-1$ smallest and the $n-k-1$ largest x-coordinates by posets [1]. Let A' (respectively A'') be the set of the $n-k-1$ points of S with the smallest (largest) x-coordinates. Note that from the technique in [1] it follows that A' and A'' are sorted. Let A be the set of points of S with x-coordinates between those of the points of A' and A'' ($A = S - A' - A''$) (see Figure 4.3).

The number of points in A is $2k + 2 - n$. Since $\frac{n}{2} \leq k < n$, for every

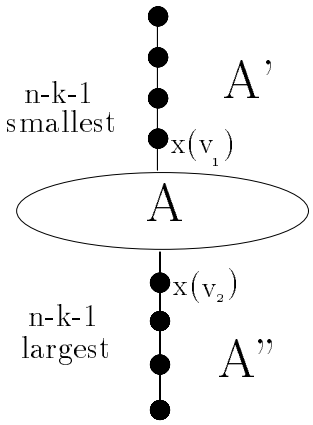

Figure 4.3: Poset for $n-k-1$ largest and $n-k-1$ smallest values.

point $p_i \in S$ all the points of A are among the k nearest x-neighbors of p_i, and the $n-k-1$ farthest x-neighbors of p_i can be only in $A' \cup A''$. For the same reason, for a point $p_i \in A'$ we will look for the farthest x-neighbors in A'' and among all the points in A' whose x-coordinate is smaller than $x(p_i)$. Symmetrically, if $p_i \in A''$ we will look for the farthest x-neighbors in A' and among all the points in A'' whose x-coordinate is greater than $x(p_i)$. Assume $p_i \in A$. Then by a simple merge of A' and A'' we can find the $n-k-1$ points farthest from p_i. If $p_i \in A'(A'')$ then we perform a similar merge on $A''(A')$ and the set containing all the points in $A'(A'')$ whose x-coordinate is smaller (greater) than $x(p_i)$.

Returning to the two-dimensional problem, we store all the points of S in an array T. We create separate posets for the x and y axes. We call them the x-poset and the y-poset. Entry i for point p_i in T will contain 2 pointers: one to the leaf in the x-poset containing p_i, and one to the leaf in the y-poset. Our goal is to find for every point $p_i \in S$ all the $n-k-1$ farthest rectilinear neighbors.

We create a set L of candidate neighbors with their L_∞ distances. For each point $p_i \in S$ it is enough to store the entry i_1 (i_2) in A' (A'') where the search for the $n-k$ farthest x-neighbors halted. Symmetrically for the y-neighbors. There is a possibility that the same point appears in both the set of farthest x-neighbors and the farthest y-neighbors of p_i. We go over all the $n-k-1$ farthest y-neighbors of p_i and check if their corresponding x-coordinate is in the range $[1, i_1]$ and $[i_2, n]$ in the x-poset. If the answer is "YES" then the same point, say p_j, appears as the farthest neighbor of p_i in both axes, we choose the maximum distance of the two distances. Assume, that the maximum distance was obtained on the x-axis. Then we put into the set L the point p_j with a flag noting it x and skip in the x-poset and y-poset to the next farthest points. At the end of the process L has l points,

where $(n-k-1) \leq l \leq 2(n-k-1)$. We find the $(n-k-1)$th point in L using the linear time selection algorithm of [25] and thus solve the problem.

Considering the time complexity. Creating the posets takes $O(n+\Sigma_{i=k}^{n} \log i)$ $= O(n + (n-k)\log n)$ time. The merge step over A', A'' and the selection take $O(n-k)$ time per point of S. The required storage, $O(n)$, is used for storing the posets, the auxiliary array T, L, and the indices. We conclude by the following theorem:

Theorem 4.2.1 *Given a set S of n points in the plane, we can find the the $n-k-1$ rectilinear farthest neighbors of all the points in S (or, equivalently, k nearest rectilinear neighbors) in $O(n+(n-k)\log n+n(n-k)) = O((n-k)n)$ time, using linear space.*

Remark 4.2.2 *This problem can be easily extended to d-dimensional space, $d \geq 3$. Perform, for each axis $i, 3 \leq i \leq d$ the same algorithm as for the y axis in the previous algorithm. The set L has $(n-k-1) \leq l \leq d(n-k-1)$ points, and the $(n-k-1)$th point in L is determined by the selection algorithm. So the total runtime and space remain unchanged for a constant dimension d.*

Remark 4.2.3 *The algorithm described above still works when $k < \frac{n}{2}$. First we sort all the points according to their x and y-coordinates. Then for each point we find the $n-k-1$ farthest neighbors in both axes by the same algorithm as before, create L and use the selection algorithm. In this case we add factor of $O(n \log n)$ to the runtime of the algorithm.*

4.3 Enumerating rectilinear distances (p16)

The problem is: Given a set S of n distinct points in the plane, let $D = \{d_1, d_2, \ldots, d_N\}$, where $N = \frac{n(n-1)}{2}$ and $d_1 \geq d_2 \geq d_3 \geq \ldots \geq d_N$ denote the rectilinear distances determined by all the pairs of points in S. For a given positive integer $k \leq N$, we want to enumerate all the k pairs of points which realize the k *largest* distances in D. For some values of k we do not need to know the total order of the points (in x or y axis). For example, if $k = 1$ then the maximum and minimum values of the x and y coordinates suffice.

As in the previous section we first show an algorithm that enumerates all the k pairs of points which realize the k largest distances on the x axis.

Assume that the points of S are sorted by their x-coordinate in increasing order and name them by this order, namely points $1, 2, \ldots, n$. For d_1 we know that the points 1 and n (according to the sorting) realize this distance. We denote this pair by $(1, n)$. One can also think about the interval $[1, n]$ containing the n x-consecutive points. We will use the notation (i, j) to denote both the pair of points i and j and the interval $[i, j]$. The next distance, d_2, can be realized by one of the *candidate pairs* $(1, n-1)$ or $(2, n)$.

Depending on the pair that realized d_2, the distance d_3 has also two candidate pairs. It is possible that the number of candidate pairs in step i will grow, if, for example, the pair $(1, n-1)$ realized d_2 and the pair $(2, n)$ realized d_3, then the candidates for realizing d_4 are the pairs $(1, n-2), (2, n-1), (3, n)$. We denote the set of candidate pairs for distance i by L_i. This is the set of pairs of points that can potentially realize d_i, after the pair that realized d_{i-1} is known. An interval (ζ, ψ) is *nested* in (ξ, η) if $(\zeta, \psi) \subseteq (\xi, \eta)$. Throughout the algorithm we will make sure that L_i does not contain nested intervals.

We say that the candidate pair (i, j), where $i < j+1$ *blocks* $(i+1, j)$ and $(i, j-1)$ because the x-distance defined by points i and j is greater than the distances defined by the pairs $(i+1, j)$ and $(i, j-1)$.

Claim 4.3.1 *L_i differs from L_{i-1}, $i \geq 2$ by at most three candidate pairs : one that is deleted from L_{i-1} and at most two new pairs that are inserted into L_i.*

Proof. For L_1 we have only candidate pair $(1, n)$. L_2 consists of the pairs $(2, n)$ and $(1, n-1)$. If, wlog, the pair $(1, n-1)$ in L_2 realizes d_2, then L_3 will consist of $(2, n)$ and $(1, n-2)$. This is because $(2, n)$ blocks $(3, n)$ and $(2, n-1)$. If the distance defined by the pair $(2, n)$ is always smaller than the distances defined by the pairs $(1, n-j)$ for $1 \leq j \leq n-2$, then L_i is different from L_{i-1} by deleting $(1, n-j)$ and inserting $(1, n-j-1)$. If for some $j, 1 \leq j \leq n-2$, the distance realized by the pair $(2, n)$ is greater than the distance realized by the pair $(1, n-j)$, then the candidate pairs for the next stage are changed by inserting two candidate pairs $(3, n)$, $(2, n-1)$ and deleting $(2, n)$ and $(1, n-j)$ remains as a candidate as well. Thus, we conclude that if at some stage i there is only one pair (ξ, η) in L_i, then at the next stage this pair is deleted, and two new pairs $(\xi+1, \eta)$ and $(\xi, \eta-1)$ (if they exist) are inserted into L_{i+1} as candidate pairs. If, at some stage i there are several candidate pairs and one of them, e.g. (ξ, η) realizes d_i, then for the next stage this pair is deleted and $(\xi+1, \eta)$ and $(\xi, \eta-1)$ (if exist) are inserted into L_{i+1} unless there is exists candidate pair in L_i (except for (ξ, η)) that blocks them. Thus, we delete one candidate pair and insert at most two candidate pairs. ∎

We define *left* and *right neighbors* of a pair (ξ, η) as follows: a left neighbor of (ξ, η) is every pair $(\mu, \eta-1), \mu < \xi$. A right neighbor of (ξ, η) is every pair $(\xi+1, \mu), \mu > \eta$.

Throughout the updates of L_i we do not re-insert a pair that had been used before to realize a distance $d_j, j < i$. Moreover, we avoid storing nested intervals in L_i. As we reach stage $i-1$ we find which pair of L_{i-1} realizes d_{i-1}. Assume (ξ, η) realizes d_{i-1}. We update L_{i-1} to get L_i. We delete (ξ, η) from L_{i-1}. If L_{i-1} contained a left (right) neighbor of (ξ, η) then we do not add the pair $(\xi, \eta-1)$ $((\xi+1, \eta))$ to L_i. Otherwise we add these pairs to L_i. This ensures that L_i does not contain nested intervals.

Claim 4.3.2 *If a pair (ξ, η) realizes d_i, then it will not be added as a candidate pair in L_j, for $j > i$.*

Proof. We prove by induction. L_1 consists of only one interval $(1, n)$. L_2 contains two candidate pairs $(1, n-1)$ and $(2, n)$ that define intervals that overlap but are not nested. The pair $(1, n)$ will not be inserted to $L_j, j > 1$, since we always decrease the interval. Assume we are at stage i. By the induction hypothesis L_i does not contain nested intervals. Assume that $(\xi, \eta) \in L_i$ realizes d_i. (ξ, η) can donate two new overlapping intervals to L_{i+1}: namely, $(\xi + 1, \eta)$ and $(\xi, \eta - 1)$. We look at the neighbors of (ξ, η) in L_i. If there exists a left neighbor of (ξ, η), then we do not add $(\xi, \eta - 1)$ to L_{i+1} in our algorithm (same for the right neighbor). Clearly, (ξ, η) will not re-appear in the next stages because we only decrease the range of intervals and since there is no nesting there is no interval that contains (ξ, η). ∎

Corollary 4.3.3 $|L_i| \leq i$, $i = 1, \ldots, n-1$, *and* $|L_i| \leq n-1$, $i = n, \ldots, \frac{n(n-1)}{2}$.

Following corollary 4.3.3 we can easily solve problem **p16** for one axis. Since the number of candidates for each stage does not exceed $n - 1$, it suffices to find the updates to the candidate list L_i at each stage i, and then find which pair realizes d_i. Naively we can carry out one stage in $O(n)$ time, therefore the k largest distances are found in $O(kn)$ time and linear space. This runtime can be improved by using tournament trees ([1, 95]) with $n-1$ leaves, each storing a candidate pair. Initially we store only one candidate pair, namely $(1, n)$, and the other leaves are empty. As we proceed to L_i we make at most three updates to the tree. The pair that realizes d_i is the winner of the tournament. The update of the tournament tree for L_{i+1} proceeds as follows: If we do not need to add anything we just empty the leaf occupied by the winner for d_i and continue to find the second best (the pair for d_{i+1}) in the tournament tree. If we add one pair, we replace the contents of the leaf that contained the winner with the new pair and update the path to the root to find the pair realizing the next distance. If we add two pairs, than we put one pair instead of the winner's leaf, another pair into the current available leaf (we always have one due to corollary 4.3.3) and update two paths to the root to find the next winner. We take care of not inserting a nested interval by maintaining an array U whose i'th entry is either empty or contains a pointer to the leaf containing the pair (i, j) in the tournament tree for some j. (Notice that there can be only one leaf containing i as the first point, since there is no nesting). The leaves of the tournament tree point to their corresponding entries in U, and each non empty entry in U points also to the closest non empty pairs in U, backwards and forward respectively.

An update of the tree takes $O(\log n)$ time, so the runtime of this algorithm is improved from $O(kn)$ to $O(n + k \log n)$.

Returning to the L_∞ metric. We perform the algorithm for the x axis simultaneously with the algorithm for the y axis. We first compute the winner

in both trees and compare the two distances: the largest current x-distance and the largest current y-distance. We choose the largest between them. We check whether these two distances are defined by the same pair of points. If they are, then we choose the largest distance, report the pair and proceed with both the algorithms to the next step (namely, updating the tournament trees, and finding the next winners). If they are not, then we check whether the larger of the distances has been reported before (in $O(1)$ time we compute the distance in the other axis and compare it to the distance we have in that axis at this stage of the algorithm). If it has been reported, we move to the next step in this axis, and if not we report this pair of points and proceed to the next stage.

Theorem 4.3.4 *Given a set S of n points in the plane and a number k we can enumerate the k largest rectilinear distances in nonincreasing order in $O(n + k \log n)$ time, using only $O(n)$ space.*

Remark 4.3.5 *If U is implemented as a linked list, and the tournament tree is implemented as a heap then the space is $O(\min(k, n))$.*

The second case of problem **p16** is: enumerate the k smallest rectilinear distances in increasing order. The idea is similar to the algorithm above. We first show an algorithm that enumerates all the k pairs of points which realize the k smallest distances on the x axis. We assume that the points of S are sorted by their x-coordinate, in increasing order. A candidate pair for realizing d_1 is either one of the neighboring pairs $(\xi, \xi + 1)$, for $\xi = 1, \ldots, n-1$, We choose the pair that realizes the smallest distance by creating a tournament tree of pairs. At the following step we perform similar updates to the tournament tree, namely, delete the pair that realized d_1 and insert at most two new candidate pairs, avoiding nested pairs. The algorithm that we apply here is almost identical to the previous one, except that here the distances increase, and we have to initially sort the coordinates of the points.

Theorem 4.3.6 *Given a set S of n points in the plane and a number k we can enumerate the k smallest rectilinear distances in nondecreasing order in $O(n \log n + k \log k)$ time, using only $O(n)$ space.*

Remark 4.3.7 *These enumerating problems can be extended to arbitrary, but constant, d-dimensional space, $d \geq 3$. Runtime and space are changed by a multiplicative d factor .*

4.4 Reporting δ distances (p17)

In a recent paper Dickerson and Eppstein [41] considered the following problem:

p17: Given a set S of n distinct points in d-dimensional space, $d \geq 2$, and a distance δ. For each point p in S report all pairs of points (p, q) with q in S such that the distance from p to q is less than or equal to δ.

This problem and the problem of enumerating the k smallest distances in nondecreasing order are closely related. If δ of this problem is the unique k^{th} largest distance of the enumerating problem, then the two solutions are identical. The paper [41] solve Problem **p17** in $O(n \log n + k)$ time and $O(n)$ space algorithm, where k is the number of distances not greater than δ, and the distances are not ordered. Our algorithm reports these distances *sorted* in the same time and space complexity for L_∞.

Another variant of this problem, that has not been considered before, is: Find all pairs of points in S separated by a L_∞ distance δ or more.

For both variants of the above problem, if we want the distances sorted, we can use our algorithms from the previous section to get $O(n + k \log n)$ algorithm with linear space for the first vesrion, where k is the number of distances not greater than δ, and $O(n \log n + k \log k)$ time algorithm with linear space for the second version. The only change is that we compare the output distances with δ. Notice that if we use the algorithm of [41] for sorting the distances then we would end up spending $O(n + k)$ space.

We want to solve first the second version of the problem. The technique is similar to the one we used in solving Problem **p15**. We first describe an algorithm for the x axis.

Throughout the algorithm we will maintain a poset (which is initially empty) that will contain the largest and the smallest x values of the points that have been encountered in the algorithm (as will be seen below). Pick an arbitrary point $p_1 \in S$. The farthest x-neighbor of p_1 can be the point with the smallest (or largest) x coordinate. The smallest point is added to the set s_x and the largest to the set g_x. After we find which point is the farthest x-neighbor of p_1 (say it is p_i and assume wlog $p_i \in s_x$), we check whether $|x(p_1) - x(p_i)| \geq \delta$. If $|x(p_1) - x(p_i)| < \delta$, then we know that there is no point $q \in S$, such that $|x(p_1) - x(q)| \geq \delta$. If $|x(p_1) - x(p_i)| \geq \delta$ we continue to find the next farthest x-neighbor of p_1 and update s_x and g_x accordingly. It can either be a point with x-coordinate adjacent to $x(p_i)$ in s_x or the next farthest point in g_x. The algorithm for p_1 ends when on both ends of s_x and g_x the distance is smaller than δ. We end up with a poset P_x, where s_x and g_x are sorted in x order and the rest of the points in $S - s_x - g_x$ are not sorted. Similarly, we work on the y distance for p_1, and create P_y, s_y and g_y.

In order to find the δ L_∞ distances for p_1 we go over P_x and P_y. If the same point, p_j, appears either in x or in y sets, then we can output the pair (p_1, p_j) and proceed to the next points till we got all the points whose distance from p_1 is not smaller than δ. We repeat the process with $p_2 \in S$. As for p_1 the x-farthest point is the point with the largest or smallest x-coordinate, but this point is already in g_x or s_x. So we go over P_x as was created for

p_1. We might add points to g_x, s_x, if all the distances $|p_2, q| > \delta, q \in s_x$ or $q \in g_x$ or not. Now we use the sets s_x, g_x, s_y, g_y computed before and report the appropriate pairs that have the required distance (not smaller than δ). There are two possibilities, (1) no points are added to s_x (or s_y, g_x, g_y), or (2) some are added. The number of elements in $s_x(g_x, s_y, g_y)$ does not decrease.

Considering the time complexity. The worst case is when we have to know the total x-order and y-order of all the points in S. The worst case runtime is $O(n \log n + k)$ and the space is $O(n)$.

The algorithm for the first version of the problem is very similar to the above algorithm. The main difference is that instead of starting at the farthest neighbors and constructing $P_x(P_y)$ incrementally, we now sort the $x(y)$ coordinates of the points of S (so we do not need the posets). For each point p_i we go over its x (and y) nearest neighbors in left (up) and right (down) directions and report the distances (similar to algorithm for the second version) as long as they are less than δ.

Theorem 4.4.1 *Given a set S of n points in the plane and a distance $\delta > 0$ we can report all the pairs of points of S which are of rectilinear distance δ or more (less) in $O(n \log n + k)$ time, using only $O(n)$ space.*

Note that in the theorem above k is the number of L_∞ distances for the case of "more than δ", and k is the number of distances measured along x and y axes for the case of "less than δ".

4.5 Rectangular rings (p19)

The problem is: Given a set S of n points in the plane, find the smallest rectangular axis-aligned ring (constrained or non-constrained) that contains $k, k \geq \frac{n}{2}$ points of S. As a measure we take the width (for constrained ring) or area (for non-constrained ring) of the ring.

4.5.1 Constrained rectangular ring

This problem can be translated to the following one:
For every point $p_i \in S$ find the $n-k$ nearest and $n-k$ farthest rectilinear (under L_∞ metric) neighbors. We can use our algorithm for problem **p15** from Section 4.2 to find the $n-k-1$ farthest rectilinear neighbors for each point of S, and the algorithm of [41] to find the $n-k-1$ nearest neighbors. Given the set of the $n-k-1$ nearest neighbors N_i of $p_i \in S$ and the set of the $n-k-1$ farthest neighbors F_i, we sort N_i and F_i according to their L_∞ distance from p_i. There are exactly $n-k-1$ rings centered in p_i and containing k points. The rings $j = 1, \ldots, n-k-1$ are determined by the j'th points in the sorted N_i and F_i respectively, where the j'th point from

F_i determines the outer rectangle and the j'th point from N_i determines the inner rectangle.

The runtime of the algorithm in [41], as well as for our algorithm for Problem **p15** is $O(n-k)$ for one point $p_i \in S$ (after $O(n \log n)$ time for preprocessing). We spend $O((n-k)\log(n-k))$ time for sorting N_i and F_i for each point $p_i \in S$, and then go over the corresponding rectangles. Therefore,

Theorem 4.5.1 *Given a set S of n points in the plane, we can find the smallest rectangular axis-aligned constrained ring that contains $k, k \geq \frac{n}{2}$ points of S in $O(n \log n + n(n-k)\log(n-k))$ time, using $O(n)$ space.*

Remark 4.5.2 *This problem can be easily extended to arbitrary, but constant, d-dimension space, $d \geq 3$, the runtime changes by multiplicative d factor.*

4.5.2 Non-constrained rectangular ring

We find the smallest rectangular ring that contains $k, k \geq \frac{n}{2}$ of given n points by first computing all the rectangles which contain $k+p$ points ($p = 1, \ldots, n-k$). Each such rectangle defines a center c for which we find the largest rectangle centered at c that contains p points. In [95] an algorithm for finding the smallest axis-aligned rectangle that contains $k, k \geq \frac{n}{2}$ points is presented. The outline of algorithm from [95] is as follows: initially fix the leftmost point of the rectangle to be the leftmost point of S. At the next stage the leftmost point of the rectangle is fixed to be the second left point of S, etc. Within one stage, of a fixed leftmost rectangle point, r, we pick the rightmost point of the rectangle to be the q'th x-consecutive point of S, for $q = k + r - 1, \ldots, n$. For fixed r and q the x boundaries of the rectangle are fixed, and we go over a small number of possibilities to choose the upper and lower boundaries of the rectangle so that it will enclose k points. This algorithm runs in time $O(n + (n-k)^3)$. We use it for computing all the rectangles which contain $k+p$ points ($p = 1, \ldots, n-k$). We denote the external rectangle by $\mathcal{R}$.

We modify the problem of finding the smallest rectangle with a given center, that contains p points, to find the largest rectangle with a given center, that contains p points. Notice that the external rectangle $\mathcal{R}$ defines the range of boundaries for the internal rectangle. Our algorithm goes over all the possible rectangles with the given center that contain p points and chooses the largest among them as follows. Let Q be an inner rectangle that contains p points. We extend its boundaries until it almost meets, but does not contain another point of S, within the boundaries of $\mathcal{R}$.

The naive approach for finding the largest rectangle with a given center that contains p points is to go over all pairs of points that together with the

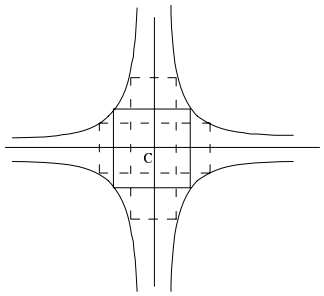

Figure 4.4: Hyperbolas define the locus of rectangles with given area

center c define a rectangle, check whether this rectangle contains p points and find the largest rectangle among those that do. The total running time is $O(n^3)$.

Another approach to this problem is to define the following decision problem: For a given area $\mathcal{A}$ does there exist a rectangle centered at c that covers exactly p points and whose area is $\mathcal{A}$. For the decision algorithm we sort the points of S according to their x and y coordinates respectively. Four hyperbolas define the locus of all rectangles with a given area $\mathcal{A}$, centered at c (see Figure 4.4). Observe the halfspace defined by the hyperbola H that contains the origin. We consider all the points of S which are inside the intersection of the four halfspaces that correspond to the four hyperbolas. Denote this set by $S' \subseteq S$. Each point $s \in S'$ defines two rectangles with center c and the given area: where s either determines the *width* of the rectangle, or its height. For the time being we look at the rectangle whose width is determined by s. Let s be the point that determines the widest rectangle Q and assume that s is to the left of c.

We shrink the width of the rectangle, keeping its corners in the corresponding hyperbolas until an *event* happens. (The height of a rectangle grows when the width shrinks) An event occurs when a point is added or deleted from the rectangle during the width shrinking. We check if the newly obtained rectangle contains p points. If the obtained rectangle does contain p points, we are done; otherwise we continue to shrink the rectangle until the next event. We perform the same actions for the height as well.

For speeding up the running time of this algorithm we define four subsets U, D, R, L of S' corresponding to the halfplanes that bound Q. R is the set of all the points of S' contained in the halfplane to the right of the left side of Q and are within the interior of the hyperbolas. L (U, D) is the set of points to the left (up, down) of the right (upper, lower) side of the rectangle Q. We define $p_r(p_l)$ to be the point x-closest to Q in $R(L)$ and $p_u(p_d)$ to be the point y-closest to Q in $U(D)$. Assume that the number of points contained in Q is r and we are shrinking Q in x direction until the next event. Assume that the

x-closest neighbor of $p_r(p_l)$ in $R(L)$ is $p_r^h(p_l^h)$ and the y-closest neighbor of $p_u(p_d)$ in $U(D)$ is $p_u^v(p_d^v)$. Thus, our event is when one of p_r^h, p_l^h or p_u^v, p_d^v enters or exists the rectangle Q. If Q contained r points and the next event is a point from R or L, then the new rectangle will contain $r-1$ points, otherwise $r+1$. We update p_r, p_l, p_u, p_d (and also the subsets U, D, R, L). When we reach a rectangle with p points we first extend its boundaries with $\mathcal{R}$ until it almost touches the $p+1$'th point and then we move to the next step (with the same center). During the process for this center we keep the largest area inner rectangle encountered so far. The algorithm for solving the decision problem works in time $O(n)$ after preprocessing of $O(n \log n)$, because we can carry each step in constant time, except for the first step where we have to compute the points that lie in the interior of the hyperbolas.

In order to solve the optimization problem, we apply the optimization technique of Frederickson and Johnson [54]. We define the matrix of distances as follows: one dimension of the matrix contains the sorted x-distances from the center (multiplied by 2), and the other dimension contains the sorted y-distances from the center (multiplied by 2). The matrix values are potential area values of the rectangle. We perform a binary search on the matrix to find the optimal area. Since the rows and columns of the matrix are sorted, we can use the linear time selection algorithm of [54] to find the largest axis-parallel rectangle centered at c and containing p points in $O(n \log n)$ time.

The analysis follows this of [95]: There are $O((n-k)^4)$ external rectangles, and for each of them we apply an $O(n \log n)$ algorithm for finding the largest internal rectangle. So, the total runtime is $O(n(n-k)^4 \log n)$ with linear space. We conclude by the following theorem:

Theorem 4.5.3 *Given a set S of n points in the plane, we can find the smallest area rectangular axis-aligned ring that contains $k, k \geq \frac{n}{2}$ points of S in $O(n(n-k)^4 \log n)$ time, using $O(n)$ space.*

Remark 4.5.4 *This problem can be extended to 3-dimension space. Using the algorithm of [95] and technique of [54] for 3-dimension space we obtain algorithm with runtime $O(n^2(n-k)^6 \log n)$ time.*

4.6 Constrained circular ring (p19 and p20)

The problem is: Given a set S of n points, find the smallest *constrained* circular ring (or a sector of a constrained circular ring) that contains k points ($k \geq \frac{n}{2}$) of S. We first describe an algorithm that finds the smallest width circular ring containing k points ($k \geq \frac{n}{2}$), and centered at some point $p_i \in S$. We need to know the sorted order of the $n-k$ closest points to p_i and $n-k$ farthest points from p_i and then proceed as in the algorithm for finding a constrained rectangular ring. The time for computing the $n-k$ closest and $n-k$ farthest points for p_i is $O(n + (n-k)\log n)$. Thus we can conclude by

Theorem 4.6.1 *Given a set S of n points in the plane, we can find the smallest width constrained ring that contains $k, k \geq \frac{n}{2}$ points of S in $O(n^2 + n(n-k)\log n))$ time, using $O(n)$ space.*

Now we describe how to find minimal **area** sector of a constrained ring that contains $k, k \geq \frac{n}{2}$, points. We first describe an algorithm that finds the smallest area sector of a ring containing k points ($k \geq \frac{n}{2}$) centered at point $O(0,0)$. We start with finding for $O(0,0)$ the ordering of S points with respect to the polar angle around the origin. We use the algorithm in [95] (and also section above) to solve our problem in the following way: apply the algorithm in [95] for a smallest axis-aligned rectangle with k points using a polar coordinate system (ρ, θ). This yields the smallest area sector of a ring centered at the origin and containing k points of S. We proceed as in the algorithm of [95]. The running time of this algorithm is $O(n + k(n-k)^2)$. We can use this ring-algorithm as a subroutine to solve the following problem: Find the smallest area sector of a constrained ring (centered on an input point) containing k points. We can perform an angular sort of all the points in $O(n^2)$ time and space [74] and applying this algorithm to each point we get $O(n^2 + nk(n-k)^2)$ time.

Theorem 4.6.2 *Given a set S of n points in the plane, we can find the smallest area sector of a constrained ring that contains k points ($k \geq \frac{n}{2}$) points of S in $O(n^2 + nk(n-k)^2)$ time using $O(n^2)$ space.*

4.7 Query rectangle (p12)

The problem is: Given a set S of n points in the plane and a number k ($\frac{n}{2} \leq k \leq n$) we want to preprocess the points in order to answer efficiently whether k or more points are enclosed by a query rectangle. The naive approach to this problem is to build a range tree [19] on the set S. When a query rectangle R is given, we can answer how many points are inside of R in $O(\log n)$ time using the fractional cascading technique of [32]. The preprocessing time and space is $O(n \log n)$. Notice that we did not use the parameter k at all. In order to improve the preprocessing time and space and also the query time we use the following observation.

Observation 4.7.1 *In order for the query rectangle to contain at least k points, the vertical strip defined by the vertical sides l_1, l_2 of the query rectangle R must be located between the $n-k$ smallest and $n-k$ largest x values of the points of S and the horizontal strip defined by the horizontal sides l_3, l_4 of the query rectangle R must be located between the $n-k$ smallest and $n-k$ largest y values of the points of S.*

Using this observation we proceed as follows. First we evaluate the smallest and the largest $n-k$ x values of the points of S (denote by S_x) and the

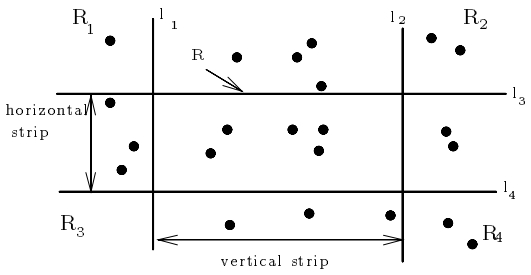

Figure 4.5: The strips enclose a query rectangle R.

smallest and the largest $n-k$ y values of the points of S (denote by S_y). Next, by a binary search, we find how many points are in the left halfplane of l_1, in the right halfplane of l_2, in the upper halpfplane of l_3 and in the lower halpflane of l_4 (See Figure 4.5).

Notice that we count twice the points in the regions $R_i, 1 \leq i \leq 4$ in Figure 4.5. We can compute how many points are in these regions by building, at the beginning of the algorithm, a range search tree but only for the points with either x-coordinate in S_x or y-coordinate in S_y. We have $O(n-k)$ such points. Thus the construction of the tree takes $O((n-k)\log(n-k))$ time with $O((n-k)\log(n-k))$ space. Now we can compute how many points are in the four query rectangles that correspond to the regions $R_i, 1 \leq i \leq 4$ in the Figure 4.5. It follows that the query time for such a rectangle is $O(\log(n-k))$. Thus,

Theorem 4.7.2 *Given a set S of n points in the plane and a number k ($\frac{n}{2} \leq k \leq n$), we can preprocess the points of S in $O((n-k)\log(n-k))$ time with $O((n-k)\log(n-k))$ space to answer in $O(\log(n-k))$ time whether k or more points are enclosed by a query rectangle.*

Bibliography

[1] M. Aigner, *Combinatorial search*, Wiley-Teubner Series in CS, John Wiley and Sons, 1988.

[2] P. Agarwal and J. Erickson "Geometric range searching and its relatives", TR CS-1997-11, Duke University, 1997

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

[4] P. K. Agarwal and J. Matoušek "Ray shooting and parametric search" *SIAM J. Computing*, 22, pp. 794–806, 1993.

[5] P. K. Agarwal and M. Sharir "Efficient randomized algorithms for some geometric optimization problems", *Discrete Comput. Geom.*, 16, pp. 317–337, 1996.

[6] P. Agarwal, B. Aronov, M. Sharir, S. Suri, "Selecting distances in the plane", *Algorithmica*, 9, pp. 495–514, 1993.

[7] P. Agarwal and M. Sharir. "Planar geometric location problems", *Algorithmica*, 11, pp. 185–195, 1994.

[8] P. K. Agarwal and M. Sharir, "Planar geometric location problem and maintaining the width of a planar set", *Proc. 2nd ACM-SIAM Symp. on Discrete Algorithms*, pp. 449–458, 1991.

[9] P. K. Agarwal, B. Aronov, M. Sharir, Subhash Suri, "Selecting distances in the plane", *Algorithmica*, 9, pp. 495–514, 1993.

[10] P. Agarwal, M. Sharir, S. Toledo, "Applications of parametric searching in geometric optimization", *J. Algorithms*, 17, pp. 292–318, 1994.

[11] P. Agarwal, M. Sharir, S. Toledo "An efficient multi-dimensional searching technique and its applications", Tech. Report CS-1993-20, Dept. Comp. Sci., Duke University, 1993.

[12] P. Agarwal, M. Sharir, E. Welzl "The discrete 2-center problem", *Proc. 13th ACM Symp. on Computational Geometry*, pp. 147–155, 1997.

[13] T. Asano, B. Bhattacharya, J. M. Keil, F. Yao "Clustering algorithms based on minimum and maximum spanning trees", *Proc. 4th ACM Symp. on Computational Geometry*, pp. 252–257, 1988.

[14] E. Assa and M. Katz, "3-piercing of d-dimensional boxes and homothetic triangles", *Int. J. Comp. Geom. and Appls*, to appear.

[15] M. Attalah, R. Cole, M. Goodrich, "Cascading divide and conquer: a technique for designing parallel algorithms", *SIAM Journal on Computing*, 18(3), pp. 499–532, 1989.

[16] F. Aurenhammer and H. Edelsbrunner "An optimal algorithm for for constructing the weighted Voronoi diagram in the plane", *Pattern Recognition*, 17(2), pp. 251–257, 1984.

[17] C. Bajaj, "Geometric optimization and computational complexity", Ph.D. thesis, Tech. Report TR-84-629, Cornell University, 1984.

[18] G. Barequet, A. Briggs, M. Dickerson, M. Goodrich "Offset-polygon annulus placement problems", *Lecture Notes in Computer Science*, 1272, pp. 378–391, 1997.

[19] J. L. Bentley "Decomposable searching problems", *Info. Proc. Lett.* 8, pp. 244–251, 1979.

[20] M. de Berg, M. van Kreveld, M. Overmars, O. Schwartzkopf *Computational Geometry, Algorithms and Applications*, Springer-Verlag, 1997.

[21] S. Bespamyatnikh, K. Kedem, M. Segal "Optimal facility location under various distance functions" Tech. Report 98-07, Dept. of Math and Comp. Science, Ben-Gurion University.

[22] S. Bespamyatnikh and M. Segal "Covering the set of points by boxes", *Proc. 9th Canadian Conference on Computational Geometry*, pp. 33–38, 1997.

[23] B. Bhattacharya and H. Elgindy "An efficient algorithm for an intersection problem and an application", Tech. Report 86-25, Dept. of Comp. and Inform. Sci., University of Pennsylvania, 1986.

[24] J. Brimberg and A. Mehrez "Multi-facility location using a maximin criterion and rectangular distances", *Location Science* 2, pp. 11–19, 1994.

[25] M. Blum, R. Floyd, V. Pratt, R. Rivest, R. Tarjan "Time bounds for selection", *Journal of Computer and System Sciences*, 7(4), pp. 448–461, 1973.

[26] T. Chan "Geometric Applications of a Randomized Optimization Technique", In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pp. 269–278, 1998.

[27] T. Chan "On enumerating and selecting distances", In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pp. 279–286, 1998.

[28] B. Chazelle, "Filtering search: A new approach to query-answering", *SIAM J. Comput.*, 15, pp. 703–724, 1986.

[29] B. Chazelle, "A functional approach to data structures and its use in multidimensional searching", *SIAM J. Comput.*, 17, pp. 427–462, 1988.

[30] B. Chazelle, H. Edelsbrunner, L. Guibas, M. Sharir "Diameter, width, closest line pair, and parametric searching", *Discrete Comput. Geom.*, 10, pp. 183–196, 1993.

[31] B. Chazelle and H. Edelsbrunner and L. Guibas and M. Sharir "Algorithms for bichromatic line segment problems and polyhedral terrains", *Algorithmica*, 11, pp. 116–132, 1994.

[32] B. Chazelle and L. Guibas "Fractional cascading: I. A data structuring technique", Algorithmica, 1, pp. 133–162, 1986.

[33] R. Cole, "Parallel merge sort", *SIAM J. Computing*, 17(4), pp. 770–785, 1988.

[34] R. Cole, J. Salowe, W. Steiger, E. Szemerédi. "An optimal-time algorithm for slope selection", *SIAM J. Comput.*, 18, pp. 792–810, 1989.

[35] T. Cormen, C. Leiserson and R. Rivest *Introduction to algorithms*, The MIT Press, 1990.

[36] A. Datta, H.-P. Lenhof, C. Schwarz, M. Smid, "Static and dynamic algorithms for k-point clustering problems", *J. Algorithms*, 19, pp. 474–503, 1995.

[37] L. Danzer and B. Grünbaum, "Intersection properties of boxes in R^d", *Combinatorica* 2(3), pp. 237–246, 1982.

[38] O. Devillers and M. Katz, "Optimal line bipartitions of point sets", *Int. J. Comput. Geom. and Appls*, to appear.

[39] M. Dickerson, R. L. Scot Drysdale, J-R. Sack "Simple algorithms for enumerating interpoint distances and finding k nearest neighbors", *Int. J. Comput. Geom. and Appls.*, 2(3), pp. 221–239, 1992.

[40] M. Dickerson and J. Shugart "A simple algorithm for enumerating longest distances in the plane", *Inf. Process. Lett.* 45, pp. 269–274, 1993.

[41] M. Dickerson and D. Eppstein "Algorithms for proximity problems in higher dimensions", *Computational Geometry: Theory and Applications* 5, pp. 277–291, 1996.

[42] Z. Drezner "The p-center problem: heuristic and optimal algorithms", *Journal of Operational Research Society*, 35, pp. 741–748, 1984.

[43] Z. Drezner "On the rectangular p-center problem", *Naval Res. Logist. Q.*, 34, pp. 229–234, 1987.

[44] H. Ebara, N. Fukuyama, H. Nakano, Y. Nakanishi "Roundnes algorithms using the Voronoi diagrams", *Abstracts 1st Canad. Conf. Comput. Geom.*, pp. 41, 1989.

[45] A. Efrat and M. Sharir "A near-linear algorithm for the planar segment center problem", *Discrete Comput. Geom.*, 16, pp. 239–257, 1996.

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

[47] H. Elgindy and M. Keil "Efficient algorithms for the capacitated 1-median problem", *ORSA J. Comput*, 4, pp. 418–424, 1982

[48] D. Eppstein "Faster construction of planar two-centers", *Proc. 8th ACM-SIAM Symp. on Discrete Algorithms*, pp. 131–138, 1997.

[49] J. Erickson and D. Eppstein "Iterated nearest neighbors and finding minimal polytopes", *Discrete Comput. Geom*, 11, pp. 321–350, 1994.

[50] R. Y. Flatland, C. H. Stewart "Extending range queries and nearest neighbors", *Proc. 7th Canad. Conf. Comput. Geom.*, pp. 267–272, 1995.

[51] F. Follert "Lageoptimierung nach dem Maximin-Kriterium", Diploma Thesis, Univ. d. Saarlandes, Saarbrucken, 1984.

[52] F. Follert, E. Schömer, J. Sellen, "Subquadratic algorithms for the weighted maximin facility location problem", *Proc. 7th Canad. Conf. Comput. Geom.*, pp. 1–6, 1995.

[53] G. Frederickson and D. Johnson "The complexity of selection and ranking in $X+Y$ and matrices with sorted columns", *J. Comput. Syst. Sci.*, 24, pp. 197–208, 1982.

[54] G. Frederickson and D. Johnson, "Generalized selection and ranking: sorted matrices", *SIAM J. Comput.* 13, pp. 14–30, 1984.

[55] G. Frederickson. "Optimal algorithms for tree partitioning", *Proc. 2nd ACM-SIAM Symp. on Discrete Algorithms*, pp. 168–177, 1991.

[56] G. Frederickson and D. Johnson. "Finding k^{th} paths and p-center by generating and searching good data structures", *J. Algorithms*, 4, pp. 61–80, 1983.

[57] A. Glozman, K. Kedem, G. Shpitalnik, "On some geometric selection and optimization problems via sorted matrices", *Computational Geometry : Theory and Applications*, 11, pp. 17–28, 1998.

[58] M. Goodrich, "Geometric partitioning made easier, even in parallel", *Proc. 9th Annu. CM Sympos. Comput. Geom.*, pp. 73-82, 1993.

[59] J. Hershberger, "A faster algorithm for the two-center decision problem", *Inf. Process. Lett.*, 47, pp. 23–29, 1993.

[60] J. Hershberger and S. Suri "Finding tailored partitions", *J. Algorithms*, 12, pp. 431–463, 1991.

[61] M. Houle and G. Toussaint, "Computing the width of a set", *IEEE Trans. Pattern Anal. Mach. Intell.*, PAMI-10, pp. 761–765, 1988.

[62] R. Z. Hwang, R. C. T. Lee, R. C. Chang "The slab dividing approach to solve the Euclidian p-center problem", *Algorithmica*, 9, pp. 1–22, 1993.

[63] R. Z. Hwang, R. C. Chang, R. C. T. Lee "The generalized searching over separators strategy to solve some NP-hard problems in subexponential time", *Algorithmica*, 9, pp. 398–423, 1993.

[64] J. Jaromczyk and M. Kowaluk, "An efficient algorithm for the Euclidian two-center problem", *Proc. 10th ACM Sympos. Comput. Geom.*, pp. 303-311, 1994.

[65] J. Jaromczyk and M. Kowaluk "Orientation independent covering of point sets in R^2 with pairs of rectangles or optimal squares", *European Workshop on Comp. Geometry*, University of Muenster, pp. 54–61, 1996.

[66] J. Jaromczyk and M. Kowaluk, "The two-line center problem fom a polar view: A new algorithm and data structure", *Lecture Notes in Computer Science*, 955, pp. 13–25, 1995.

[67] N. Katoh, K. Iwano, "Finding k farthest pairs and k closest/farthest bichromatic pairs for points in the plane", *Int. J. Comput. Geom. Appl.* 5, pp. 37–52, 1995.

[68] F. Preparata "New parallel-sorting schemes", *IEEE Trans. Comput.*, C-27, pp. 669–673, 1978.

[69] M. Katz, K. Kedem, M. Segal "Constrained Square-Center Problems", *Computational Geometry: Theory and Apllications*, to appear.

[70] M. Katz, K. Kedem, M. Segal "Improved algorithms for placing undesirable facilities" *European Workshop on Comp. Geometry*, Antibes, 1999.

[71] M. Katz and F. Nielsen, "On piercing sets of objects", In *Proc. 12th ACM Symp. on Computational Geometry*, pp. 113–121, 1996.

[72] M. Katz and M. Sharir "An expander-based approach to geometric optimization", *SIAM J. Comput.*, 26(5), pp. 1384–1408, 1997.

[73] V. B. Le and D. T. Lee "Out-of-roundness problem revisited", *IEEE trans. Pattern Anal. Mach. Intell* PAMI-13, pp. 217-223, 1991.

[74] D. T. Lee and Y. T.Ching "The power of geometric duality revised", *Inf. Process. Lett.* 21, pp. 117–122, 1985.

[75] H-P. Lenhof and M.Smid "Sequential and parallel algorithms for the k closest pairs problem", *Internat. J. Comput. Geom. Appls.* 5, pp. 273–288, 1995.

[76] C. Makris and A. Tsakalidis "Fast Piercing of Iso–Oriented Rectangles", *Proc. 9th Canad. Conf. Comput. Geom.*, pp. 217–222, 1997.

[77] J. Matoušek "Efficient partition trees" *Discrete Comput. Geom.* 8, pp. 315–334, 1992.

[78] J. Matoušek "On geometric optimization with few violated constraints", *Discrete Comput. Geom.*, 14, pp. 365–384, 1995.

[79] N. Megiddo "Applying parallel computation algorithm in the design of serial algorithms", *Journal of ACM*, 30, pp. 852–865, 1983.

[80] N. Megiddo "Linear time algorithms for linear programming in R^3 and related problems", *SIAM J. Comput.*, 12, pp. 759–776, 1983.

[81] N. Megiddo, "On the complexity of some geometric problems in unbounded dimension", *J. Symbolic Comput.*, 13, pp. 182–196, 1984.

[82] N. Megiddo and A. Tamir "New results on the complexity of p-center problems", *SIAM J. Comput.*, 12(4), pp. 751–758, 1983.

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

[84] J. Mitchell and E. Wynters "Finding Optimal Bipartitions of Points and Polygons", *Lecture Notes in Computer Science*, 519, pp. 202–213, 1991.

[85] C. Monma and S. Suri "Partitioning points and graphs to minimize the maximum or the sum of diameters", *Graph Theory, Combinatorics and Applications*, 2, pp. 899–912, 1991.

[86] D. Nussbaum "Rectilinear p-Piercing Problems", *Proc. Int. Symp. on Symb. and Alg. Comput.*, pp. 316–323, 1997.

[87] M. Overmars and C. Yap, "New upper bounds in Klee's measure problem", *SIAM J. Comput*, 20, pp. 1034–1045, 1991.

[88] M. Overmars and J. van Leeuwen, "Maintenance of configurations in the plane", *J. Comput. Syst. Sci.*, 23, pp. 166–204, 1981.

[89] F. Preparata and M. Shamos "Computational Geometry: An Introduction", Springer-Verlag, New York, NY, 1985.

[90] J. Salowe "Enumerating interdistances in space", *Int. J. Comput. Geom. Appls.*, 2, pp. 49–59, 1992.

[91] J. Salowe "L-infinity interdistance selection by parametric search", *Inf. Process. Lett.*, 30, pp. 9–14, 1989.

[92] H. Samet, "The design and analysis of spatial data structures", Addison-Wesley, 1990

[93] M. Segal and K. Kedem "Geometric applications of posets", *Computational Geometry : Theory and Applications*, 11, pp. 143–156, 1998.

[94] M. Segal "On the piercing of axis-parallel rectangles and rings", *Inter. J. Comp. Geom. Appls*, to appear.

[95] M. Segal, K. Kedem "Enclosing k points in the smallest axis parallel rectangle", *Inf. Process. Lett.*, 65, pp. 95–99, 1998.

[96] M. Segal and S. Bespamyatnikh "Rectilinear static and dynamic center problems", *manuscript.*

[97] M. Sharir, "A near-linear algorithm for the planar 2-center problem", *Discrete Comput. Geom.*, 18, pp. 125–134, 1997.

[98] M. Sharir and P. Agarwal, *Davenport-Shintzel sequences and their applications*, Cambridge University Press, New-York, 1995.

[99] M. Sharir and E. Welzl, "Rectilinear and polygonal p-piercing and p-center problems", *Proc. 12th ACM Symp. on Computational Geometry*, pp. 122–132, 1996.

[100] M. Smid and R. Janardan "On the width and roundness of a set of points in the plane", *Proc. 7th Canad. Conf. Comput. Geom.*, pp. 193–198, 1995.

[101] L. Valiant, "Parallelism in comparison problems", *SIAM J. Computing*, 4, pp. 348–355, 1975.

[102] D. Willard and G. Lueker, "Adding range restriction capability to dynamic data structures", *Journal of ACM*, 32, pp. 597–617, 1985.