Geometric Applications of Posets*

Michael Segal Klara Kedem
Department of Mathematics and Computer Science,
Ben-Gurion University of the Negev, Beer-Sheva 84105, Israel

October 28, 1999

Abstract

We show the power of posets in computational geometry by solving several problems posed on a set S of n points in the plane: (1) find the $n-k-1$ rectilinear farthest neighbors (or, equivalently, k nearest neighbors) to every point of S (extendable to higher dimensions), (2) enumerate the k largest (smallest) rectilinear distances in decreasing (increasing) order among the points of S, (3) given a distance $\delta > 0$, report all the pairs of points that belong to S and are of rectilinear distance δ or more (less), covering $k \geq \frac{n}{2}$ points of S by rectilinear (4) and circular (5) concentric rings, and (6) given a number $k \geq \frac{n}{2}$ decide whether a query rectangle contains k points or less.

Keywords: Algorithms, posets, nearest neighbors, optimization, distances.

1 Introduction

1.1 Problems

Given a set S of n points in the plane and an integer k we solve the following problems in this paper:

*Work by Klara Kedem has been supported by a grant from the U.S.-Israeli Binational Science Foundation.

1. 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.

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

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

4. 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.

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

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

1.2 Background

Most of the problems mentioned above have been considered in previous papers [7, 8, 9, 11, 18]. We summarize our and previous results in Table 1. Dickerson et al. [7] present an algorithm for the first problem which runs in time $O(n \log n + nk \log k)$, and works for any convex distance function. Eppstein and Erickson [11] solve the first problem on a random access machine model in time $O(n \log n + kn)$ and $O(n \log n)$ space. In the algebraic

Pbm	*Previous results*	*Our results*
1	$O(n \log n + kn)$ [9]	$O(n \log n + (n-k)n)$
2	$O(n \log n + \frac{k \log k \log n}{\log \log n})$ (expected) [8]	$O(n + k \log n)$
3	$O(n \log n + k)$ [9]	$O(n \log n + k)$
4	open, constrained open, non constrained	$O(n(n-k) \log (n-k)$ $O(n(n-k)^4 \log n)$
5	open	$O(n^2 + n(n-k) \log n)$
6	Preprocess: $O(n \log n)$ Query: $O(\log n)$ [4]	$O(n + (n-k) \log n)$ $O(\log (n-k))$

Table 1: Summary of best previous results and our results

desicion tree model their time bound increases by a factor of $O(\log \log n)$. Flatland and Stewart [12] present an algorithm for the first problem which runs in time $O(n \log n + kn)$ in the algebraic decision tree model. Finally, a recent paper of Dickerson and Eppstein [9] describes an $O(n \log n + kn)$ time and $O(n)$ space algorithm for the first problem, it works for any metric and is extendable to higher dimensions. For our best knowledge only two papers, one by Dickerson and Shugart [8] and one by Katoh and Iwano [14] present an algorithm for the second problem (for the *largest* k distances). The algorithm in [8] 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 [14] presents an algorithm for the second problem for 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. [7] 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. [18], Salowe [19], Dickerson and Eppstein [9] also solved this problem 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 second problem (for the smallest k distances) is also presented in [9], spending $O(n \log n + k \log k)$ time and using $O(n + k)$ space. Dickerson and Eppstein [9] also considered the third problem: find all the 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 greater than δ.

Problem 6 is a variant of the orthogonal range search where we are given a set S of n points and want to find which points are enclosed by the query rectangle. This problem was efficiently solved by Bentley [4] in $O(\log n + m)$ query time, where m is the number of points contained in the given query rectangle, using the range search tree and with preprocessing time and space $O(n \log n)$.

Some variations of problems 4 and 5 have been considered in previous papers. Efrat et al.[10] consider the problem 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 [2, 16, 21]. The best known running time is $O(n^{\frac{3}{2}+\epsilon})$, given in [2], where $\epsilon > 0$ is an arbitrary small constant. The paper of Barequet et al. [3] 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 $\delta > 0$. Segal and Kedem [20] considered the problem of enclosing k ($k \geq \frac{n}{2}$) points in the smallest axis parallel rectangle. Their algorithm runs in time $O(n+k(n-k)^2)$ and uses $O(n)$ space. Their method and algorithm are one of the tools used in this paper, and we review it below. It is based on *posets* (partially ordered sets) [1].

Segal and Kedem [20] describe how to construct a poset such that a subset R of S contains the $n-k$ elements of S with the largest x coordinates. They represent S as a tournament tree. The tournament tree can be implemented as a heap. The operations of creating R and updating the tournament tree run in times $O(n+(n-k)\log n)$ and $O(\log n)$ respectively. Space requirement is $O(n)$. (For more details see the full version of [20].) With the use of posets they devise an algorithm that finds the smallest axis parallel rectangle with $k \geq \frac{n}{2}$ points in $O(n+k(n-k)^2)$ time and $O(n)$ space.

The runtimes achieved in the previously described papers works for problems 1,4 and 5 are not attractive when the k is close to n. We show that in this case the use of posets can significantly reduce the runtime of the algorithms. Our algorithm for solving the first problem runs in time $O((n-k)n)$ (assuming $k \geq \frac{n}{2}$) and uses linear space. For problem 2 we present two algorithms : for enumerating the largest and smallest distances. The first one runs in time $O(k\log n+n)$, and uses $O(n)$ space. The second algorithm runs in time $O(n\log n + k\log n)$, and uses $O(n)$ space. We solve both cases of problem 3 by a similar technique. For our best knowledge the second case of problem 3 has not been considered before. The runtime and space requirements of both algorithms for Problem 3 are as in [9], namely $O(n\log n + k)$ time and $O(n)$ space. We solve problem 4, rectangular ring containing $k(k \geq \frac{n}{2})$ points for the constrained case in $O(n(n-k)\log{(n-k)})$ time and $O(n)$ space, while for the non constrained case we present an algorithm with runtime $O(n(n-k)^4 \log n)$ and $O(n)$ space. We find a constrained circular ring that covers k ($k \geq \frac{n}{2}$) points (Problem 5) in $O(n^2 + n(n-k)\log n)$ time and $O(n)$ space, and we find a *sector* of a constrained circular ring that covers k points ($k \geq \frac{n}{2}$) in $O(n^2 + nk(n-k)^2)$ time and $O(n^2)$ space. For the sixth problem we obtain an algorithm with $O(n + (n-k)\log n)$ preprocessing time and space and $O(\log(n-k))$ query time. We also show how to extend the algorithms of all the problems to higher dimensional space.

1.3 Motivation

Another algorithm that runs efficiently for large k, $k \geq \frac{n}{2}$ values was presented by Matoušek [17]. It finds the smallest circle enclosing all but a few $(n-k)$ of the given n points in the plane. Given a large integer $\frac{n}{2} \leq k \leq n$ his algorithm runs in time $O(n \log n + (n-k)^3 n^{\varepsilon})$ for some $\varepsilon > 0$.

A possible motivation 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 outlyers in the data. Assuming $n-k$ data points are outlyers, one way to find the k "good" data points is to enclose them in a small given shape (or shapes).

2 Rectilinear nearest neighbors (Problem 1)

Problem: 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 of [20].

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

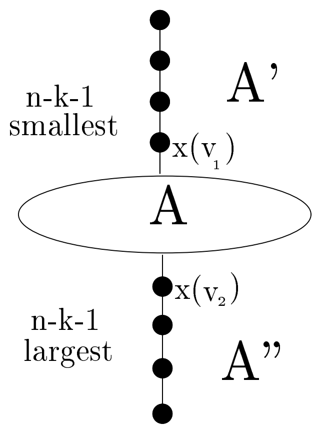

Figure 1: Poset for $n-k-1$ largest and $n-k-1$ smallest values.

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 1).

The number of points in A is $2k+2-n$. Since $\frac{n}{2} \leq k < n$, for a every 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 on 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 [5] 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 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 1 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 2 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.

3 Enumerating rectilinear distances (Problem 2)

Problem 2: 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 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 $1 \le j \le n-2$, then L_i is different from L_{i-1} by deleting $(1,n-j)$ and inserting $(1,n-j-1)$. If for some $j, 1 \le j \le n-2$, 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 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 3: $|L_i| \leq i$, $i = 1, \dots, n-1$, and $|L_i| \leq n-1$, $i = n, \dots, \frac{n(n-1)}{2}$.

Following Corollary 3 we can easily solve Problem 2 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, 20]) 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 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 $\mathrm{O}(\log n)$ 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 3.1 *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 3 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 (2) 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 3.2 *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 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 Reporting δ distances (Problem 3)

In a very recent paper Dickerson and Eppstein [9] considered the following problem:

Problem 3.1: 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 [9] solve Problem 3.1 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:
Problem 3.2: Find all pairs of points in S separated by a L_∞ distance δ or more.

For both Problems 3.1 and 3.2, 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 Problem 3.1, 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 Problem 3.2. The only change is that we compare the output distances with δ. Notice that if we use the algorithm of [9] for sorting the distances then we would end up spending $O(n + k)$ space.

We want to solve first Problem 3.2. The technique is similar to the one we used in solving Problem 1. 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 Problem 3.1 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 3.2) as long as they are less than δ.

Theorem 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".

5 Rectangular rings (problem 4)

In this section we solve problem 4: 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.

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 from Section 2 to find the $n-k-1$ farthest rectilinear neighbors for each point of S, and the algorithm of [9] 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 $j = 1, \dots, n-k-1$ 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 [9], as well as for our algorithm (Section 2) 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 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 5 This problem can be easily extended to arbitrary, but constant, d-dimension space, $d \geq 3$, the runtime changes by multiplicative d factor.

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 ($p = 1, \dots, n-k$). Each such rectangle defines a center c for which we find the largest rectangle centered at c that contains p points. In [20] 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 [20] 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 $q = k+r-1, \dots, n$. 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

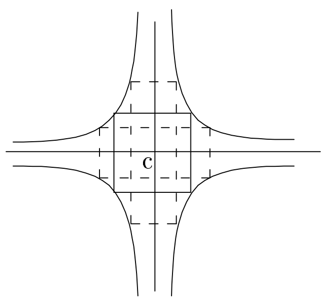

Figure 2: Hyperbolae define the locus of rectangles with given area

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 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 hyperbolae define the locus of all rectangles with a given area $\mathcal{A}$, centered at c (see Figure 2). 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 hyperbolae. 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 hyperbolae until an *event* happens. (the height of a rectangle grows when the width shrinks) An event is 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 hyperbolae. 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 hyperbolae.

In order to solve the optimization problem, we apply the optimization technique of Frederickson and Johnson [13]. 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 [13] 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 [20]: 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 5.2 *Given a set S of n points in the plane, we can find the small-*

est 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 6 This problem can be extended to 3-dimension space. Using the algorithm of [20] and technique of [13] for 3-dimension space we obtain algorithm with runtime $O(n^2(n-k)^6 \log n)$ time.

6 Constrained circular ring (Problem 5)

In this section we solve the following problems: 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 find 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 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 [20] to solve our problem in the following way: apply the algorithm in [20] 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 [20]. 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 [15] and applying this algorithm to each point we get $O(n^2 + nk(n-k)^2)$ time.

Theorem 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.*

7 Query rectangle (Problem 6)

Problem: 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 [4] 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 [6]. 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 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*

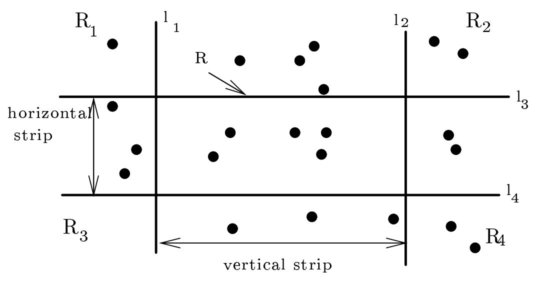

Figure 3: The strips enclose a query rectangle R.

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 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 3 below).

Notice that we count twice the points in the regions $R_i, 1 \leq i \leq 4$ in Figure 3. 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 3. It follows that the query time for such a rectangle is $O(\log(n-k))$. Thus,

Theorem 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.*

References

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

[2] P.K. Agarwal, M. Sharir, "Efficient randomized algorithms for some geometric optimization problems", In *Proc. 11th Annu. ACM Symp. Computational Geometry*, 1995.

[3] G. Barequet, A. Briggs, M. Dickerson, M. Goodrich, "Offset-polygon annulus placement problems", In *Workshop on Algorithms and Data Structures (WADS'97)*, Lecture Notes in Computer Science 1272, Springer-Verlag, 378–391.

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

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

[6] B.M. Chazelle, L.J. Guibas, "Fractional cascading: I. A data structuring technique", Algorithmica, 1, 133–162, 1986.

[7] M.T. Dickerson, R.L.S Drysdale, J-R. Sack, "Simple algorithms for enumerating interpoint distances and finding k nearest neighbors", *Internat. J. Comput. Geom. Appl.*, 2(3):221–239, 1992.

[8] M.T. Dickerson, J. Shugart, "A simple algorithm for enumerating longest distances in the plane", *Inform. Process. Lett.* 45, 269–274, 1993.

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

[10] A. Efrat, M. Sharir, A. Ziv, "Computing the smallest k-enclosing circle and related problems", *Computational Geometry* 4, 119–136, 1994.

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

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

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

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

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

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

[17] J. Matoushek, "On geometric optimization with few violated constraints", *Discrete Comput. Geom.*,14 (1995), 365–384.

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

[19] J. Salowe, "Enumerating interdistances in space", *Internat. J. Comput. Geom. Appl.* 2, 49–59, 1992.

[20] M. Segal, K. Kedem, "Enclosing k points in the smallest axis parallel rectangle", In *Proc. 8th Canad. Conf. Comput. Geom.*, 20–25, 1996.

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