Universiteit Utrecht

Data Structures for Point Enclosing Problems on the Circle and the Square

Tomas Ehrencron

Computing Science

Master Thesis

Tomas Ehrencron
ICA-5651808

Computing Science

Supervisors:

Dr. F. Staals
Utrecht University

I.D. van der Hoog MSc
Utrecht University

August 1, 2021

Abstract

A well-studied class of geometrical problems is the class of point enclosing problems. Here we are given a point set P and a convex shape s. Either we have to enclose as many points of P with s or we have to enclose a certain number of points and we minimise the size of s. In both cases the input consists of a set points and value which is either the number of points k that we must enclose or the size r of the shape. We consider specifically the interval for 1-dimensional points and the circle and square for 2-dimensional points. Although these problems have been studied extensively in the last few decades, we take a different approach. Instead of solving one of these problems directly, we construct, for a specific problem, a data structure on the point set such that we can query either a value k or r (depending on the problem) and answer the problem more quickly. Building the data structure might take more time than computing the problem directly, but if we need to solve the same problem for different values of k or r, this approach can be faster than solving the problem from start for each individual input.

Contents

1 Introduction and problem description

Throughout this thesis we define P to be a point set in $\mathbb{R}^m$ with $m \in \{1, 2\}$. Occasionally, we show statements that hold in higher dimensions as well. Then we denote the dimension by d which will always be a constant. We will consider two variants of problems that are closely related. In the first we are given a certain convex shape s and we must position s such that it contains the maximum number of points in P. In the second variant we are given a class of shapes S and an integer k. We then have to find the shape s of S of minimal size that contains at least k points. In both cases points on the boundary of s are contained by s as well. More precisely, we consider the following problems:

- MIN-SEG(k): Given a set P of n points in $\mathbb{R}^1$. Find the smallest interval (or segment) that encloses at least k points of P. We call this interval the *k-smallest interval*.
- MAX-PTS-SEG(r): Given a set P of n points in $\mathbb{R}^1$. Find an interval of size r that encloses the maximum number of points of P.
- MIN-SQUARE(k): Given a set P of n points in $\mathbb{R}^2$. Find the smallest axis-aligned square that encloses at least k points of P. We call this square the *k-smallest square*.
- MAX-PTS-SQUARE(r): Given a set P of n points in $\mathbb{R}^2$. Find an axis-aligned square with sides length r that encloses the maximum number of points of P.
- MIN-CIRCLE(k): Given a set P of n points in $\mathbb{R}^2$. Find the smallest circle that encloses k points of P. We call this circle the *k-smallest circle*.
- MAX-PTS-CIRCLE(r): Given a set P of n points in $\mathbb{R}^2$. Find a circle of diameter r that encloses the maximum number of points of P.
- MIN-AREA(k): Given a set P of n points in $\mathbb{R}^2$. Find the smallest area axis-aligned rectangle that encloses k points of P.
- MAX-PTS-AREA(α): Given a set P of n points in $\mathbb{R}^2$. Find an axis-aligned rectangle with area α that encloses the maximum number of points of P.
- MIN-PERIM(k): Given a set P of n points in $\mathbb{R}^2$. Find the smallest perimeter axis-aligned rectangle that encloses k points of P.
- MAX-PTS-PERIM(α): Given a set P of n points in $\mathbb{R}^2$. Find an axis-aligned rectangle with perimeter α that encloses the maximum number of points of P.
- MIN-DIAG(k): Given a set P of n points in $\mathbb{R}^2$. Find the smallest diagonal axis-aligned rectangle that encloses k points of P.
- MAX-PTS-DIAG(α): Given a set P of n points in $\mathbb{R}^2$. Find an axis-aligned rectangle with diagonal α that encloses the maximum number of points of P.
- MAX-PTS-RECT(l, w): Given a set P of n points in $\mathbb{R}^2$. Find an axis-aligned rectangle with length l and width w that encloses the maximum number of points of P.

Remark 1.0.1. Since our goal was to consider most of these problems, we have found results for these problems in the literature study. For completion we have included all definitions here, but we do not discuss all problems involving the rectangle except for MAX-PTS-RECT(l, w) outside the literature study.

The problems we introduced above are researched extensively as we will see in Section 3. They all fall in the category *function problems* [20] where each valid input has a single output. A different set of problems is the category of *data structure problems*. Here we are first given input which we have to store in a data structure. Then we are given a query on the input that we have to answer using the data structure. The goal is to built the data structure in such a way that we can answer the query efficiently. We have to find a balance between the query time, the preprocessing time to construct the data structure and the space usage

of the data structure. This approach is useful when we have to answer multiple queries on the same input. A data structure can be designed depending on the available space or preprocess time.

In this thesis we aim to find a data structure for the above problems. Specifically, we consider the problems Min-Seg(k), Max-Pts-Seg(r), Min-Square(k), Max-Pts-Square(r), Min-Circle(k), Max-Pts-Circle(r) and Max-Pts-Rect(l, w). In Section 2 we introduce the theory and techniques that we use in this thesis. Section 3 is a literature study where we discuss some of the results that already exist on our set of problems. Then in Section 4 we consider the Min-Seg(k) and the Max-Pts-Seg(r) problems. Although we do not present any major results, we discuss some idea, lemmas and examples. In Section 5 we describe a data structure for both the Max-Pts-Square(r) problem and the Max-Pts-Rect(l, w) problem. Its query time is $O(n \log(n)/\log\log(n))$ which is faster than the current fastest algorithm for these algorithms, but we assume that the maximum number of points covered by the square/rectangle is bounded by a certain function. Then in Section 6 we consider the problems on the circle. For both problems we present a data structure that requires minimal space during construction. Finally, in Section 7 we briefly discuss possible further research directions.

1.1 Relevance

The above problems are interesting in several real-life problems where we have a set of objects that we have to cover or enclose, but our number of resources is only limited. For example, we have to place a cell phone tower in an area with a low density population. If the signal has only a limited range, it might not be strong enough to reach everyone. However, we still want to reach as many people as possible. If the signal distributes equally in all directions, this is simply the Max-Pts-Circle(r) problem.

Now suppose we need a fixed number k of resources that are distributed over an area. Suppose we want to find a place such that the distance to the k closest resources is minimised. This corresponds to the Min-Circle(k). On the other hand, if we need to enclose the resources with a fence and we want to use the minimal number of fences, we have the Min-Perim(k) problem (assuming the fences must be placed axis-aligned).

In other problems we might be interested in different shapes, but it is useful to consider these problems for the most basic shapes. Results we achieve might be applicable for other shapes as well.

2 Preliminaries

In this section we discuss the prerequisite knowledge that is needed to understand this thesis. We mention certain conventions that we use in this thesis. We explain the computational model that we are working with. Since we occasionally use a slightly different model in this thesis, we also mention these small differences. Of course, throughout this thesis, we always state explicitly which model we are working with whenever we deviate from our main model. We also explain some other relevant problems that occur in this thesis. We do not focus on solving these problems, but rather use them to solve the problems that we are interested in. We discuss a construction that we use to solve the Min-Circle(k) problem. Finally, we discuss two important transformations which allow us to reduce a subproblem of the Min-Circle(k) to a related problem which we subsequently solve.

2.1 Basic definitions and conventions

Throughout this thesis we say that a circle C contains a point p if p is an element of the disc enclosed by C. Although we often mean a disc when talking about a circle, we only use the term 'circle', since this is also the term that is used in the literature as we will see in Section 3. When a point lies on the boundary of a disc, we say that it lies on the boundary of the circle, even though this is superfluous.

Since all our problems involving squares or rectangles only consider axis-aligned squares and rectangles, we will not specify that they are axis-aligned each time. Unless specified otherwise, any square and rectangle is axis-aligned.

We will be working with arrangements of (hyper)planes a few times in this thesis, so we briefly explain the terminology that we use. We use the term *hyperplane* in a d-dimensional space to denote a $(d-1)$-dimensional hyperplane unless stated otherwise. To describe the arrangement, we need the following definition.

Definition 2.1.1 ([8])**.** Let P be a finite set of points in $\mathbb{R}^d$. If, for some $d' \leq d$ all points of P do not lie in one $(d'-1)$-dimensional hyperplane, but they do lie in a d'-dimensional hyperplane, then the convex hull of P, $s := \text{conv}\{p \in P\}$, is a d'-dimensional *convex polytope*. In particular, if all points of P do not lie in the same $(d-1)$-hyperplane, then s is a d-dimensional polytope. If $|P| = d' + 1$, then we call s a d'-dimensional *simplex*. The points in the set P are called the *vertices* of s. Note that convex polytopes can also be defined by the intersection of a set halfspaces. In other words a convex polytope is bounded by $(d-1)$-dimensional hyperplanes. This definition, however, is slightly weaker as it allows for unbounded objects. These objects we will call *unbounded convex polytopes* or *unbounded simplices*.

Unless stated otherwise, we will use the term 'convex polytope' to denote a d-dimensional convex polytope and 'simplex' has an analogous convention. In 2-dimensional space a convex polytope is a convex polygon. A simplex becomes then a triangle. A 3-dimensional simplex is a tetrahedron.

Suppose we have a set H of hyperplanes in d dimensions. Then H partitions the total space in convex polytopes of dimensions from 0 to d. We call this collection the *arrangement of* H, denoted by $\mathcal{A}(H)$. A 0-dimensional polytope is the intersection of d hyperplanes and we will call this a vertex of the arrangement. Unless stated otherwise, we use the term 'cell' only for d dimensional polytopes. Specifically, in two dimensions we use the terms vertices, edges and cells for the 0-, 1- and 2-dimensional polytopes respectively. Similarly, in three dimensions we will use the terms vertices, edges, faces and cells, again for the 0-, 1-, 2- and 3-dimensional polytopes respectively.

2.2 Computational model

When considering geometric problems, it is good practice to state explicitly which model of computation we are considering. A computational model tells us how computation as an abstract concept works and thus what steps an algorithm may take to translate the input to the output. Many algorithms make use of the *word RAM* model. In this model each *word* (e.g. number, string etc.) is stored as a finite sequence of bits. The length of a single word (i.e. the number of bits) is at least $\lceil \log n \rceil$ where n is the size of the input. These words are stored in a *RAM* (Random Access Machine) which allows access in constant time. This model is very close to how computers work internally which makes it a useful model to consider.

However, in the area of computational geometry this model has some downsides. Because numbers are represented by a finite sequence, we cannot represent all real numbers, since floating point numbers have only finite precision. Even if we only allow finite precision numbers as input, this presents us with problem, since it is difficult to represent, for example, intersections of circles, in finite precision numbers.

An alternative to the word RAM is the *real RAM* model. This is an extension of the word RAM, adding a second kind of RAM, the real RAM. This RAM can store real numbers with infinite precision. An algorithm can access these values and perform operations on them in constant time. Which operations we can do exactly, differs between papers, but generally we allow addition, subtraction, multiplication and division. Taking the square root and some other simple functions are also allowed in some papers [30]. The result is that an algorithm only does 'simple' comparisons, meaning that each comparison involving a parameter x can be written as $p(x) = 0$ or $p(x) > 0$ where p is a low-degree polynomial, i.e. we can compute its roots in constant time.

There are some ways that the two kinds of RAM can interact. The first one is that we can cast values in the word RAM to the real RAM. The integer value can then be used as a real number. While determining which interaction we allow, we consider how powerful the model becomes. For example, if we allow real numbers to be cast back to the word RAM in constant time, then it turns out that in this model P = PSPACE [32]. Although the model is of course not 'wrong', it does allow an unreasonable amount of computational power, since in the word RAM model it is still unknown whether P = PSPACE. Therefore, we do not allow real numbers to be cast back to the word RAM with for example a floor or ceiling function. An important remark is that not all published papers use the exact same model. Some algorithms that are presented, do make use of operators like the floor function. We will explicitly state whenever this happens.

Lastly, we want to consider some slight variations in the meaning of the Big O notation. In this thesis we sometimes express the space or time usage in the number of words and sometimes we consider the number of bits explicitly. For example, if we store n values each between 0 and n, then we often state that this costs $O(n)$ space. This is true if you consider the number of words that are used. However, in some contexts it makes more sense to consider how many bits are used in total. For instance, when performing bit operations. In that case the memory usage becomes $O(n \log n)$ as we need $\log(n)$ bits to represent each number. In this thesis we will state explicitly which variant we use, if there is a possible confusion.

2.3 Related problems

Apart from the main problems we mentioned in the introduction, there are some other problems that we will see in this thesis. We mainly use these problems to solve our main problems. The first one is the Simplex Range Reporting problem. In the Simplex Range Reporting problem we must preprocess a set P of points in $\mathbb{R}^d$. A query then consists of a simplex s. We have to report the points of P that are enclosed by s. Sometimes we are only interested in the number of points in s. We refer to that problem as the Simplex Range Counting problem.

A special case of the above problem is the Halfspace Range Reporting problem [1]. In stead of a simplex, we must report the set of points that lie in a given halfspace. For simplicity, we assume that the hyperplane defining the halfspace is non-vertical. For any point set P there always exists a simplex that reports the same points as the halfspace. Before we show this formally, we need a different lemma.

Lemma 2.3.1. *Suppose we have a set P of n points in $\mathbb{R}^d$. Then there exists a d-dimensional simplex s such that all points of P lie in the interior of s.*

Proof. We use induction on the dimension d. For $d = 1$, this is trivial. We choose $s = [p_{\min} - 1, p_{\max} + 1]$ where $p_{\min} := \min\{p \in P\}$ and $p_{\max} := \max\{p \in P\}$. Now suppose we have shown this lemma for $d = d'$. Now we show that it also holds for $d = d' + 1$. We project P on the plane $x_{d'+1} = c$ where we choose c such that all points of P lie above this plane. So let $P' = \{(p_1, \ldots, p_{d'}, c) \mid (p_1, \ldots, p_{d'}, p_{d'+1})\}$. By assumption, we can find a set S' of $d' + 1$ points in the $x_{d'+1} = c$ plane that form a d'-dimensional simplex and contain all points of P' in its interior. Now for some arbitrary $(p_1, \ldots, p_{d'}, 0) \in S'$ we define $p^m = (p_1, \ldots, p_{d'}, m)$ with $m \in \mathbb{N}$. Then let $s^m := \text{conv}(S' \cup \{p^m\})$ be a $(d' + 1)$-dimensional simplex. Note that $s^{m_1} \subset s^{m_2}$ if $m_1 \leq m_2$. For every $p \in P$ there exists a $M_p \in \mathbb{N}$ such that p lies in the interior of s^{M_p}. Then we can choose $M_{\max} = \max\{M_p \mid p \in P\}$ and $s^{M_{\max}}$ contains all points of P in its interior. □

Lemma 2.3.2. *The* Halfspace Range Reporting *problem is a special case of the* Simplex Range Reporting *problem.*

Proof. Suppose we have a set P of n points in $\mathbb{R}^d$ and a halfspace H. Let P_H be the set of points of P that are contained in H. Without loss of generality we can assume that H is the space above some hyperplane h. Our argument here is very similar to the proof of Lemma 2.3.1. Let P_h be the orthogonal projection of P on h. Now we use Lemma 2.3.1 to find a set S' of d points that form a $(d-1)$-dimensional simplex in the plane h that contains all points of P_h in its interior. Let $\overrightarrow{v}$ be the unit vector perpendicular to h pointing in the positive direction of the dth coordinate. We take some arbitrary p from S'. We define $p^m = p + m\overrightarrow{v}$ and let $s^m := \text{conv}(S' \cup p^m)$. With a similar argument as before there exists a $M_p \in \mathbb{N}$ for every $p \in P_H$. Then, if $M_{\max}$, we have that $s^{M_{\max}}$ encloses P_H. Since $s^{M_{\max}}$ does not lie below h, it contains no points of $P \setminus P_H$. □

Analogously to the Simplex Range Counting problem, we also have the Halfspace Range Counting problem.

Finally, we define the Vertical Ray Stabbing problem. Here we are given a set of n $(d-1)$-dimensional non-vertical hyperplanes in $\mathbb{R}^d$ and a query point q in $\mathbb{R}^d$. We must report the hyperplanes that lie above q. The Vertical Ray Counting problem is then the same problem where we are only interested in the number of hyperplanes.

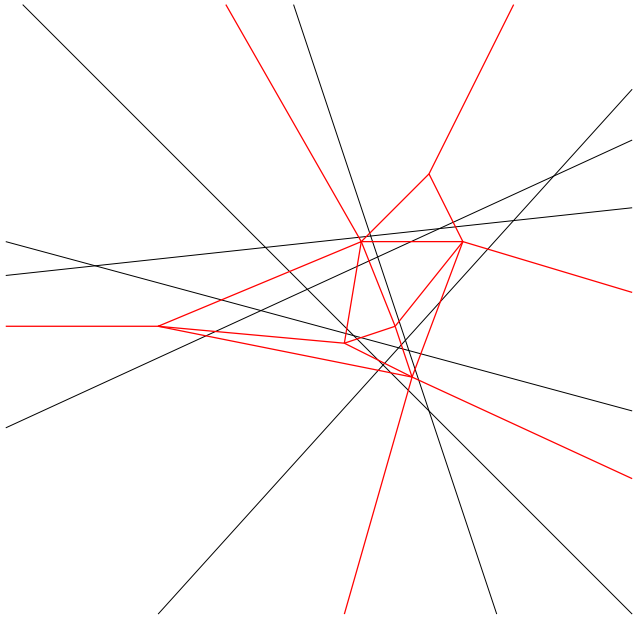


Figure 1: A (1/2)-cutting of a set of lines in $\mathbb{R}^2$. Each of the cells is intersected by at most three lines.

2.4 $(1/r)$-cuttings

An important construction that is often used in divide-and-conquer algorithms, is the $(1/r)$-cutting. Suppose we have a set of $(d-1)$-hyperplanes H in d dimensions. A $(1/r)$-cutting subdivides the space in d-dimensional simplices or *cells* such that each cell is only intersected by a small number of hyperplanes. Formally, for $1 \leq r \leq n$ a $(1/r)$-*cutting* (on H), consists of (possibly unbounded) simplices that partition[1] $\mathbb{R}^d$ such that the interior of each simplex is intersected by at most n/r hyperplanes. In some literature this is referred to as an ε-cutting which has the property that each cell is intersected by at most εn hyperplanes. Furthermore, we define the *size* of a $(1/r)$-cutting as the number of simplices it has. Figure 1 shows an example of a (1/2)-cutting of a set of six lines.

The fact that we only consider the interior of a cell is just a minor detail, but otherwise the following problem could happen. Suppose d is some large constant and we choose $r = n/2$. If a cell also includes its boundary, then it could happen that an $(1/r)$-cutting does not exist. If $n > d$ and all hyperplanes of H are pair-wise non-parallel, then there exists a point p that is intersected by d hyperplanes. But since p lies in some cell, either its interior or on the boundary, this cell is intersected by more than $n/r = 2$ hyperplanes. Since we will never choose $r = O(n)$, n/r will always be sufficiently larger than d, so this fact is not really important.

What is important, is the observation that for any H and $1 \leq r \leq n$ it is possible to create a $(1/r)$-cutting. Indeed, we can always consider the cutting that coincides with the arrangement and that splits cells into simplices. Of course, this creates a cutting that is of the same complexity as the arrangement itself. The goal, thus, is to find an arrangement, whose space usage depends only on r. If r is small, then each cell may intersect many hyperplanes, so the cutting probably does not need to have many cells. Chazelle and Friedman showed that a cutting of size $O(r^d)$ always exists [12] and that it can be computed in polynomial time (if d is constant). Later, Chazelle proved a better result which is the following:

Theorem 2.4.1 (Chazelle, Theorem 3.3 [11])**.** *Given a collection H of n hyperplanes in $\mathbb{R}^d$, for any $r \leq n$ it is possible to compute a $(1/r)$-cutting for H of size $O(r^d)$ in time $O(nr^{d-1})$.*

This theorem presents us with a lot of freedom as we can freely choose r and the space usage and construction time depend on r. Note that size of the cutting is optimal.

[1] Technically, it is not a full partition, since the simplices have overlapping boundaries.

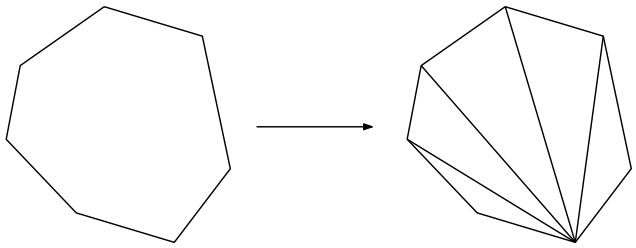


Figure 2: Canonical triangulation of a 2-dimensional polygon.

Lemma 2.4.2. *Suppose we have a set H of $(d-1)$-dimensional hyperplanes in general position. Any $(1/r)$-cutting on H must have size $\Omega(r^d)$.*

Proof. Any d hyperplanes from H will meet in a single intersection point, since we assumed general position. This means that in total there are $\binom{n}{d} = \Theta(n^d)$ points that are intersected by d hyperplanes. As each cell of the cutting is only intersected by n/r hyperplanes, it can contain at most $O((n/r)^d)$ of these points. The result of this observation is that we need $\Omega(r^d)$ cells in the cutting. □

We will now briefly explain how Chazelle's algorithm works. He starts with the 'cutting' that is just the unbounded area $\mathbb{R}^d$. Then each step he refines the cutting by splitting the simplices into smaller ones. Specifically, for some positive constant $r_0 < r$ and for $k = 0, \ldots, \lceil \log_{r_0} r \rceil$ he computes a $(1/r_0^k)$-cutting C_k for H.[2]. Every iteration Chazelle's algorithm iterates over each simplex s of C_{k-1}. Let $H_{|s}$ be the set of hyperplanes of H that intersect s. If $|H_{|s}|$ is at most n/r_0^k, then it is already a valid cell for C_k. Otherwise, Chazelle computes a specific subset R of $H_{|s}$. We will not go into full detail how we choose R exactly, but rather give an intuitive explanation. Note that the arrangement of R bounded by s also splits s in smaller cells. Unfortunately, these cells are not necessarily simplices but convex polytopes. Now R is determined as follows. We start by computing a $(1/r')$-approximation A of $H_{|s}$ for some constant r'. Now, A is a subset of $H_{|s}$ such that for any line segment e we can approximate $H_{|e}$ using $A_{|e}$. Then for this set A we compute the $(1/r')$-net R. This net has the property that for every line segment e for which $A_{|e}$ is sufficiently large, then $R_{|e} > 0$. So instead of approximating the number of hyperplanes that intersect a certain line segment, it can only detect whether this happens a certain number of times. It turns out that for this R we have that each cell in the resulting subdivision is intersected by at most n/r_0^k hyperplanes of H.

Finally, the subdivision that R imposes on s does not necessarily consist of d-dimensional simplices but of convex polytopes. Chazelle ensures that all cells are simplices again by computing what is called a *canonical triangulation* (sometimes called the *bottom vertex triangulation* [3]). This works as follows. Suppose we have some d-dimensional polytope s. If $d = 1$, we are done, since any line segment is already a simplex. Otherwise, we recursively compute the canonical triangulation of its $(d-1)$-dimensional sides. Each side consists then of $(d-1)$-dimensional simplices whose boundaries are $(d-2)$-dimensional simplices. We denote the total set of these boundaries by B. We select the lowest vertex v of s in terms of the dth coordinate. For each boundary $b \in B$ we consider the $(d-1)$-dimensional simplex, defined by the points $b \cup \{v\}$. All these simplices together, subdivide s in d-dimensional simplices. In $\mathbb{R}^2$ this process reduces to connecting each vertex in the polygon to the lowest vertex via an edge. This is shown in Figure 2. Then the polygon is split into triangles. By computing the canonical triangulation for each of the polytopes and joining the subdivisions of all simplices of C_{k-1}, we get C_k.

Since we will alter this algorithm slightly later on, we will briefly discuss part of its time and space analysis. Chazelle shows cutting a simplex s can be done in $O(|H_{|s}|)$ time, where $H_{|s}$ is the set of hyperplanes that intersect s. We will call this the *conflict list of* s. If s is a simplex of the cutting after iteration k, then $|H_{|s}| \leq n/r_0^k$. Moreover, the number of simplices after iteration k, denoted as $|C_k|$ is bounded by $r_1^{(k+1)d}$ for some constant r_1. We ensure that we choose $r_0 \geq r_1$ to get the bound $|C_k| \leq r_0^{(k+1)d}$. If we consider the final

[2]We can assume r_0 exists, because Matoušek already showed the same result when $r \leq n^{1-\delta}$ for an arbitrary small δ [26].

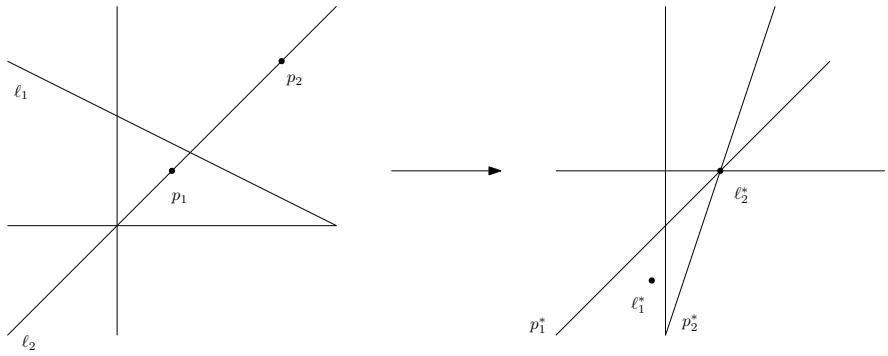


Figure 3: Dual transforms of some points and lines.

iteration where $k = \lceil \log_{r_0} r \rceil$, then we get a cutting of complexity $O(r^d)$. The total runtime now becomes

$$\begin{aligned}
\sum_{0 \le k \le \lceil \log_{r_0} r \rceil} O\left(\frac{n}{r_0^k}\right) |C_k| &\le \sum_{0 \le k \le \lceil \log_{r_0} r \rceil} O\left(n r_0^{k(d-1)+d}\right) \\
&= O\left(n \cdot \sum_{0 \le k \le \lceil \log_{r_0} r \rceil} \left(r_0^{d-1}\right)^k\right) \quad (r_0^d \text{ is constant}) \\
&= O\left(n \left(r_0^{d-1}\right)^{\lceil \log_{r_0} r \rceil}\right) \quad \text{(geometric series)} \\
&= O(n r^{d-1}).
\end{aligned}$$

During the algorithm we assume that for each simplex s we keep track of the list $H_{|s}$. However, then the space usage of the algorithm in the final step becomes $O(r^d \cdot n/r) = O(nr^{d-1})$. Hence, we can compute a $(1/r)$-cutting on a set of n hyperplanes in $O(nr^{d-1})$ time using $O(nr^{d-1})$ space.

2.5 Duality of points and hyperplanes

In this thesis we occasionally use the *duality transforms*. Definitions and properties in this section are taken from [14, Chapter 8]. We start with duality transforms in two dimensions. Then, it is a mapping that projects points on lines and lines on points in the following way.

- A point $p = (p_x, p_y)$ is mapped to the line $p^* : y = p_x x - p_y$.
- A line $\ell : y = ax + b$ is mapped to the point $\ell^* = (a, -b)$.

The images p^* and ℓ^* are called the *dual* of p and ℓ respectively and they lie in the *dual space*. The transformation has some important properties.

- The dual of the dual of an object equals the object: for every point p and line ℓ we have that $(p^*)^* = p$ and $(\ell^*)^*$.
- It is incidence preserving: p lies on ℓ if and only if ℓ^* lies on p^*.
- It is order preserving: p lies above ℓ if and only if ℓ^* lies above p^*.

An example is shown in Figure 3. We see that a line containing two points is mapped to a point that is intersected by two lines.

Remark 2.5.1. It is not possible to transform vertical lines or receive a vertical line as the dual of some point.

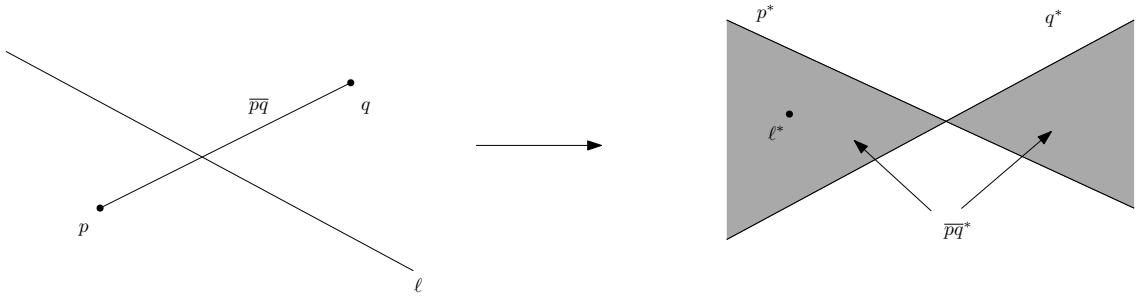


Figure 4: The dual of line segment becomes a double wedge.

If we consider a line segment to be a set of points, then the dual of a line segment becomes what is called a *double wedge* and we can describe it as the area between two lines. If we have a line segment between the points p and q, then the dual $\overline{pq}^*$ is equal to the symmetric difference of the half-planes below p^* and q^*. This is depicted in Figure 4. We also see that the line ℓ that intersects the line segment, is contained in the double wedge in the dual plane. Note that ℓ lies above p and below q, so ℓ^* must lie below p^* and above q^*. Similarly, if a line passes above p and q then its dual will also lie below p^* and q^*, so outside the double wedge.

The duality transform also works in higher dimensions. A point in $\mathbb{R}^d$ is mapped to a $(d-1)$-dimensional hyperplane and vice versa in the following way.

- A point $p = (p_1, p_2, \ldots, p_d)$ is mapped to the hyperplane $p^* : x_d = p_1x_1 + p_2x_2 + \ldots + p_{d-1}x_{d-1} - p_d$.
- A hyperplane $h : x_d = a_1x_1+a_2x_2+\ldots+a_{d-1}x_{d-1}+x_d$ is mapped to the point $h^* = (a_1, a_2, \ldots, a_{d-1}, -a_d)$.

The same properties still hold. Moreover, if we have two points p and q in $\mathbb{R}^d$, then $\overline{pq}^*$ equals the symmetric difference of the halfspaces below p^* and q^*.

We can use this transformation to show the equivalence between the Halfspace Range Reporting problem and the Halfplane Reporting problem. Suppose we have an instance of Halfplane Reporting. Then we can transform the set H of hyperplanes to a set H^* of points in $\mathbb{R}^d$ and the query point q to a hyperplane q^*. Then we must report the points of H^* that lie in the halfspace above q^*. The reverse transformation works similar.

2.6 Transformation of the circle problem

We now present an algebraic transformation that transforms circle containment queries into higher-dimensional halfspace containment queries which we will call the *lifting transformation*. Later, we will use this transformation to solve the Min-Circle(k) problem using the Halfplane Reporting problem. In short, we will map each point in $\mathbb{R}^2$ to a plane in $\mathbb{R}^3$ and a circle to a point in $\mathbb{R}^3$. The transformation has then the property that points lie in the circle if and only if the corresponding planes lie above the point representing the circle. Later, we briefly describe this transformation in higher dimensions, but now we specify it only for $\mathbb{R}^1$ and $\mathbb{R}^2$.

We start with the 1-dimensional case. So we have a set P of n points in $\mathbb{R}$. A point of P is mapped to a line ℓ_p in $\mathbb{R}^2$. Let $f(x) = x^2$ be the unit parabola. A point $p \in P$ is then mapped to the line ℓ_p that is tangent to f in the point $(p, f(p))$. We then get $\ell_p(x) = 2px - p^2$. This situation is shown for two points in Figure 5. For any two points p_1 and p_2 in P we have the following property: if $\ell_{p_1}(q) \geq \ell_{p_2}(q)$ for some $q \in \mathbb{R}$, then $|p_1 - q| \leq |p_2 - q|$. In other words, for any $q \in \mathbb{R}$ we can determine whether q lies closer to p_1 or p_2 by looking which of the corresponding lines takes on a larger value at that point. As a consequence, if $\ell_{p_1}(q) = \ell_{p_2}(q)$, then q is the midpoint of the interval $[p_1, p_2]$. We will prove the above statement for the 2-dimensional case which automatically also proves it for the 1-dimensional case.

Now, we consider the transformation of points in two dimensions. So we have a set P of points in $\mathbb{R}^2$. We will transform a point of P to a plane $h_p : \mathbb{R}^2 \to \mathbb{R}$ in $\mathbb{R}^3$. Let $f(x, y) = x^2+y^2$ be the unit paraboloid. Now h_p is defined such that it is tangent to f at $(p_x, p_y, f(p_x, p_y))$. It follows that $h_p(x, y) = 2p_xx+2p_yy-(p_x^2+p_y^2)$. It follows that $h_p(p_x, p_y) = p_x^2 + p_y^2$, so h_p goes through the desired point. Moreover, we have that $\frac{\partial f}{\partial x}(p_x, p_y) =$

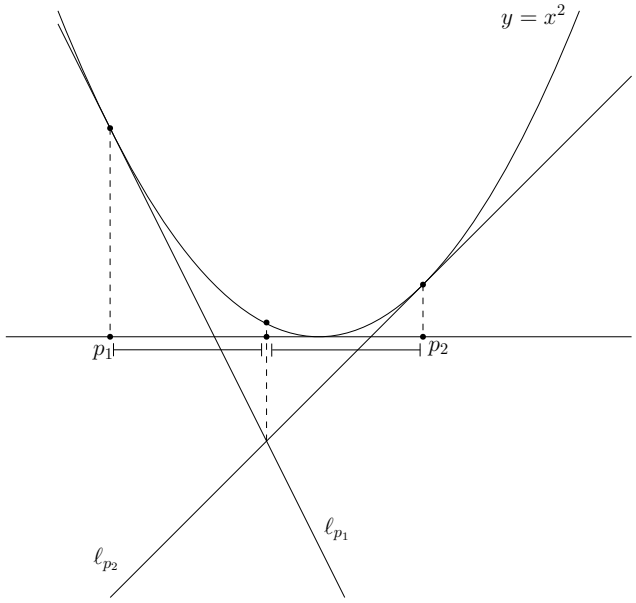


Figure 5: Transformation of one-dimensional points to lines.

$2p_x = \frac{\partial h_p}{\partial x}(p_x, p_y)$ and the same holds for the partial derivative to y. This shows that h_p is tangent to f in $(p_x, p_y, f(p_x, p_y))$. We define $H(P) := \{h_p \mid p \in P\}$ as the set of transformed points. Now we show a useful property of this transformation. Intuitively, we show the following. Suppose we consider a point $p \in P$ and $q \in \mathbb{R}^2$ and let $q' = (q_x, q_y, 0)$. If we draw a vertical line (in the z-direction) through q', then it intersects both f and h_p. The distance between these intersection points is then the distance between p and q squared. Formally, the result is as follows.

Lemma 2.6.1. *For any $p \in P$ and $q = (q_x, q_y) \in \mathbb{R}^2$, we have that $|f(q) - h_p(q)| = ||q - p||^2$. Here $||\cdot||$ is the standard Euclidean norm.*

Proof. This just follows from the definition:

$$\begin{aligned} |f(q) - h_p(q)| &= q_x^2 + q_y^2 - 2p_xq_x - 2p_yq_y + p_x^2 + p_y^2 \\ &= (q_x - p_x)^2 + (q_y - p_y)^2 \\ &= ||q - p||^2 \end{aligned}$$

□

Now if we want to find the point in P that lies closest to q, we have to find which of the planes has the highest intersection point with the vertical line through q'. Furthermore, if we sort the planes top to bottom by intersection point with this line, we get the points of P sorted closest to farthest from q. A corollary of the above observation is that if we consider a point q such that $h_p(q) = h_{p'}(q)$ for two distinct points $p, p' \in P$, then q is equidistant to p and p'. This is analogous to the claim we made about the 1-dimensional points. Note that the lemma above also holds for points in one dimension, since we can place the 1-dimensional points on the x-axis in two dimensions. Then in three dimensions we can consider the cross section of $y = 0$ to get the transformation of 1-dimensional points.

Remark 2.6.2. To avoid confusion with duality transform in Section 2.5 we will use the term *projected space* to refer to the space that contains the transformed objects.

Now we briefly consider the d-dimensional variant. We then consider a set P of points in $\mathbb{R}^d$. The transformation of a point $p = (p_1, \ldots, p_d)$ of P, h_p, then becomes a d-dimensional hyperplane in $\mathbb{R}^{d+1}$ that is tangent to the $(d+1)$-dimensional unit paraboloid $f(x_1, \ldots, x_d) = x_1^2 + \ldots + x_d^2$ in the point $(p_1, \ldots, p_d, f(p))$. A result similar to Lemma 2.6.1 holds then, but since we do not need a more general result than this lemma,

it is outside the scope of this thesis.

The above procedure transforms d-dimensional points to d-dimensional hyperplanes in $\mathbb{R}^{d+1}$. Next we show how we transform a d-dimensional sphere to a $(d+1)$-dimensional point. Again, we only discuss the case where $d = 1, 2$ and we briefly mention the general transformation. We start in one dimension. Let P be a set of points in $\mathbb{R}^1$ and let C be a 1-dimensional circle with centre c and radius r. Then C is just the interval $[c-r, c+r]$. Let p_C denote the transformed circle. We define $p_C = (c, c^2 - r^2)$. Then it holds that any $p \in P$ lies on C if and only if p_C lies on h_p. Furthermore, if p lies in the interior of C, then p_C lies below h_p. We will show these claims for the 2-dimensional case which, similar to before, automatically proves the above statements. We have drawn an example in Figure 6. In the figure we see that the transformation of C_1, p_{C_1}, lies on ℓ_{p_1}, since p_1 lies on the border of C_1. On the other hand p_{C_2} lies above ℓ_{p_2} (and above ℓ_{p_1} for that matter), so C_2 does not contain p_2.

Now let us consider the 2-dimensional case. Suppose we have a circle C with centre $c = (c_x, c_y)$ and radius r. We define $p_C = (c_x, c_y, c_x^2 + c_y^2 - r^2)$. Just as in the 1-dimensional case we want that the distance between p_C and $f(c)$ equals r^2. In that case it follows from Lemma 2.6.1 that a point p lies on C if and only if p_C lies on h_p. We can formalise this as follows.

Lemma 2.6.3. *For any $p \in P$ and circle C with centre $c = (c_x, c_y)$ and radius r, it holds that p lies on C if and only if p_C lies on h_p. Furthermore, p lies in the interior of the disc enclosed by C if and only if p_C lies below h_p.*

Proof. If p lies on C, then by Lemma 2.6.1 we have that $|f(c) - h_p(c)| = ||c-p||^2 = r^2$. Since $f(c) = c_x^2 + c_y^2$, it follows that $h_p(c) = c_x^2 + c_y^2 - r^2$ and we conclude that p_C lies on h_p. Note that it is not possible that $h_p(c) = c_x^2 + c_y^2 + r^2$, because $h_p(x) \leq f(x)$ for all $x \in \mathbb{R}$. So in particular, we have that $h_p(c) \leq f(c) = c_x^2 + c_y^2$. Now let p and C be such that p_C lies on h_p. So $h_p(c) = c_x^2 + c_y^2 - r^2$. Then $|f(c) - h_p(c)| = r^2$ and by Lemma 2.6.1 we have that p lies on C.

Now suppose we have a point $p \in P$ that lies in the interior of C. By Lemma 2.6.1 it follows that $|f(c) - h_p(c)| = ||c-p||^2 < r^2$. Analogous to the situation above, we see that $h_p(c) > c_x^2 + c_y^2 - r^2$ and thus p_C lies below h_p. The reverse implication works with a similar argument. □

Note that a circle is always transformed to a point below the paraboloid, since $c_x^2 + c_y^2 - r^2 < f(c_x, c_y)$ if $r > 0$. Moreover, for any point $p = (p_x, p_y, p_z)$ in the projected space below the paraboloid, there is a circle C such that $p_C = p$. The centre of C is (p_x, p_y) and its radius is equal to $\sqrt{p_x^2 + p_y^2 - p_x}$.

We do want to mention that the above transformation is closely related to a different transformation of the circle [14, Chapter 8.5]. Here a circle C is transformed into a plane h_C. This plane intersects the unit paraboloid such that if we map this intersection down on $\mathbb{R}^2$, then we get the original circle back. A point in $\mathbb{R}^2$ is then mapped to the unit paraboloid. This point lies inside C, if and only if its transformation lies below h_C. This transformation is more common and works almost identical to the one we used. Although we do not use this one, we want to show the relation between the two.

3 Related work

In this section we want to give an overview of the current results so far, concerning the problems we have described before. We will discuss not just the fastest algorithms for each subproblem, since the techniques used in older algorithms can still be useful. Moreover, the 'fastest' algorithm may depend on the input. Many papers we will discuss present algorithms where the running time is expressed in both n and k, meaning that some algorithms may be more efficient for smaller values of k than others, but not for larger values. Other papers present algorithms with expected running times or arbitrary approximations with a running time expressed in some error ε. We will not discuss all techniques and algorithms that are used in the algorithms in great detail, but we give a broad overview and consider some of them in depth. Table 1 shows the results that we discuss in this section.

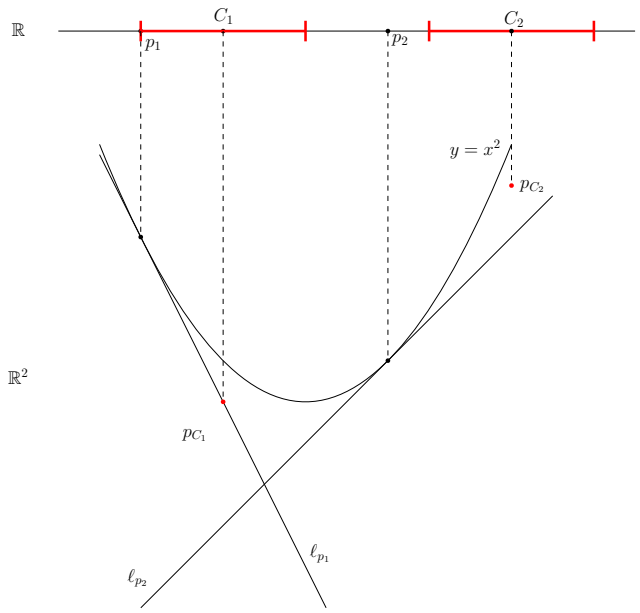


Figure 6: Transformation of one-dimensional circles to points.

3.1 Problems on the square

We start with the problem Max-Pts-Square(r). There is a well-known algorithm that solves this problem in $O(n \log n)$ time. We give a brief explanation here. For a more detailed explanation, see Section 5.1. We solve it by transforming it into a related problem. Suppose we place a square of size r centred at each vertex of P. We call the number of squares that enclose a point $q \in \mathbb{R}^2$ the *depth of* q. The *depth* of the arrangement of squares is the maximum depth over all points in $\mathbb{R}^2$. Now if we consider a point q with depth k, then we enclose k points of P if we place a square of size r centred around q. We see that Max-Pts-Square(r) is equivalent to finding the depth of the arrangement of squares of size r centred around the points of P. We accomplish this by use of a plane sweep algorithm with a segment tree as status structure [14, Chapter 10]. Normally, the goal of a segment tree is to report the intervals that contain a certain point. For this reason we store a segment in $O(\log n)$ nodes of the segment tree, resulting in $O(n \log n)$ space in total. However, if we are not interested which points we enclose, but only how many, we can just store the depth in the arrangement in which case we only need $O(n)$ space. An equivalent result was discovered by Eppstein and Erickson [17] that also generalises to higher dimensions. They adapt an algorithm by Overmars and Yap [29] to find the depth of an arrangement of axis-aligned boxes. More precisely, if the boxes are d-dimensional, the algorithm runs in $O(n^{d/2} \log n)$ time using $O(n^{d/2})$ space.

The Max-Pts-Square(r) problem is often used as a subroutine in algorithms that solve Min-Square(k). Observe that a solution square for Min-Square(k) either has two points of P on the top and bottom boundary or on the left and right boundary. This means that we can reduce the set of possible square sizes to the set of horizontal and vertical differences between two points. Note that this set has size $O(n^2)$. The most trivial approach we could take is performing a binary search on this set to find the smallest square that can enclose k points. Normally this algorithm would take $O(n^2 \log n)$ time, since each iteration of the binary search we need to find the median value of $O(n^2)$ numbers. Luckily, we can come up with a better strategy to find a median (or at least a value that splits the set in constant fractions). Using a search technique proposed by Frederickson and Johnson [19] that performs this search in $O(n \log n)$ time, Eppstein and Erickson achieved an $O(n \log^2 n)$ algorithm for this problem using $O(n)$ space [17, Lemma 5.7].

An even better result, albeit randomised, was achieved by Chan [9] using a general technique that can be applied to several other geometric minimisation problems. We will discuss the method using our problem in greater detail than other results, since it is generally applicable. An important property that Chan's idea exploits is that the decision variant of a problem can be computed more efficiently than the original problem.

Problem	Exact	Approximation
MIN-SQUARE(k)	$O(n \log^2 n)$ in $O(n)$ space [17] Expected $O(n \log n)$ in $O(n)$ space [9] $O(n + (n-k)\log^2(n-k))$ in $O(n)$ space [24]	
MAX-PTS-SQUARE(r)	$O(n \log n)$ in $O(n)$ space [17] and via sweep line with Segment Tree $O((n-k)\log(n-k))$ in $O(n)$ space [24]	
MIN-CIRCLE(k)	Expected $O(k(n-k)\log n + n\log^3 n)$ $O(nk\log^2 n)$ in $O(nk)$ space [16] $O(nk\log^2 n \log(n/k))$ in $O(n\log n)$ space [16] Expected $O(n\log n + kn)$ in $O(nk)$ space [27] Expected $O(n\log n + nk\log k)$ in $O(n)$ space [27] Expected $O(nk)$ in $O(n+k^2)$ space [21]	2-approximation in $O(n)$ time [21] $(1+\varepsilon)$-approximation in expected $O(n + n \cdot \min(\frac{1}{k\varepsilon^3}\log^2\frac{1}{\varepsilon}, k))$ time [21]
MAX-PTS-CIRCLE(r)		
MIN-AREA(k)	$O(nk^2 \log n + n\log^2 n)$ [15] $O(n^2\log n)/O(nk\log(n/k)\log k)$ [10]	$(1+\varepsilon)$-approximation in expected $O((1/\varepsilon^3)\log(1/\varepsilon)\cdot n\log n)$ [10]
MAX-PTS-AREA(α)	$O(n^{5/2}\log n)$ [22]	Whp $(1-\varepsilon)$-approximation in $O((n/\varepsilon^4)\log^3 n\log(1/\varepsilon))$ time [15]
MIN-PERIM(k)	$O(n\log n + nk\log k)$ [10]	
MAX-PTS-PERIM(α)	$O(n^{5/2}\log n)/O(nk^{3/2}\log k)$ [22]	$(1-\varepsilon)$-approximation in $O\left(n + \frac{n}{k\varepsilon^5}\log^{5/2}\left(\frac{n}{k}\right)\log\left(\frac{1}{\varepsilon}\log\left(\frac{n}{k}\right)\right)\right)$ time [22]
MIN-DIAG(k)		
MAX-PTS-DIAG(α)	$O(n^{5/2}\log n)$ [22]	

Table 1: An overview of the results that we will discuss

In our case, the decision variant is the problem that, given a value k and real r, asks whether we can enclose k points with a square of size r. We can solve this problem using the algorithm for MAX-PTS-SQUARE(r). Now suppose we call the specific problem instance we are interested in $\mathcal{P}$ and the general problem Π with $w(\mathcal{P})$ its solution. The main idea is to divide the problem in several subproblems $\mathcal{P}_1, \ldots, \mathcal{P}_r$ such that $w(\mathcal{P}) = \min\{w(\mathcal{P}_1), \ldots, w(\mathcal{P}_r)\}$. A requirement is that the size of each of the subproblems is at most a constant fraction $\alpha < 1$ compared to the original problem. The algorithm starts by recursively calculating the value $w(\mathcal{P}_1)$. The next step is then not to also calculate $w(\mathcal{P}_2)$, but to check first if $w(\mathcal{P}_2)$ is smaller than $w(\mathcal{P}_1)$ using the decision variant. If this is not the case, we do not have to compute $w(\mathcal{P}_2)$ explicitly, saving us some work. Otherwise, we calculate it recursively. We continue with the next subproblem until we are done. In the worst case we have that $w(\mathcal{P}_1) \geq w(\mathcal{P}_2) \geq \ldots \geq w(\mathcal{P}_r)$. Then the decision variant will answer 'yes' each time and we have to calculate the next value explicitly. However, if we choose the permutation

of the subproblems randomly, the probability of this happening is quite small. On average the number of subproblems we have to calculate explicitly is $1+1/2+1/3+\ldots+1/r = O(\log r)$. With some analysis, Chan showed the following lemma.

Lemma 3.1.1 ([9]). *Let $\alpha < 1$, $\varepsilon > 0$ and r be constants, and let $D(\cdot)$ be a function such that $D(n)/n^{\varepsilon}$ is monotonically increasing in n. Given any problem $\mathcal{P} \in \Pi$, suppose that within $D(|\mathcal{P}|)$ time,*

(i) we can decide whether $w(\mathcal{P}) < t$ for any given $t \in \mathbb{R}$, and

(ii) we can construct r subproblems $\mathcal{P}_1, \ldots, \mathcal{P}_r \in \Pi$, each of size at most $\lceil \alpha|\mathcal{P}| \rceil$, so that

$$w(P) = \min\{w(\mathcal{P}_1), \ldots, w(\mathcal{P}_r)\}.$$

Then we can compute $w(\mathcal{P})$ in $O(D(|\mathcal{P}|))$ expected time.

Note that there are some requirements in order to use this lemma. For example, the algorithm for the decision variant cannot be 'too fast'. If we have an algorithm that runs in $O(\log n)$ time, this technique cannot be applied. However, in our case it does work, since we have an $O(n \log n)$ algorithm for the decision variant. We can create the subproblems as follows. We draw a vertical line such that $\lceil n/5 \rceil$ points lie on the left of this line. The area to the right we call W_1. We also draw a vertical line such that $\lceil n/5 \rceil$ lie on the right and we call the area to the left W_2. Similarly, we draw two horizontal lines with corresponding half-planes W_3 and W_4. The intersection of these half planes is denoted by R_0. This construction is drawn in Figure 7. Now we distinguish between two situations.

1. The optimal square contains R_0.
2. The optimal square is contained in one of the four half planes.

Suppose the first situation holds. The $O(n \log n)$ algorithm for Max-Pts-Square(r) can be extended with the additional requirement that the square must contain a certain rectangle. In this case specifically we require that R_0 must lie inside the square. Then we remove all points in R_0 from the set P as they will automatically lie inside a square that contains R_0. These are roughly $n/5$ points, so we get a problem of size approximately $4n/5$. We also decrease k with the number of points that lie in R_0. Now suppose the optimal square lies in W_i for $i \in \{1, 2, 3, 4\}$. We can remove the $n/5$ points that lie outside this half plane. Again we get a subproblem with size approximately $4n/5$. In the end we have five subproblems for which at least one has a solution that is equal to that of the original problem. Now we can apply Chan's Lemma and we have an expected $O(n \log n)$ algorithm for solving Min-Square(k).

Mahapatra et al. [24] aimed to solve Min-Square(k) for large values of k, in particular if $k > n/2$. They exploit the fact that if k is close to n, then certain points in the 'centre' of P have to lie in the optimal square. They divide P in (not necessarily disjoint) sets P_b, P_t, P_l, P_r and P_f. Then they define P_b as the $(n-k)$ bottom most points and similarly P_t, P_l and P_r for top, left and right respectively. Finally, they define $P_f = P \setminus (P_b \cup P_t \cup P_l \cup P_r)$. In a small proof they show for $k > n/2$ that all points of P_f lie inside the optimal square. This of course still holds if P_f is empty. Note that the number of points outside of P_f is $O(n-k)$. Using the same technique for Max-Pts-Square(r) as we presented at the start, they solve this problem in $O((n-k)\log(n-k))$ time after we have removed the points in P_f. Now for Min-Square(k) they note that the optimal square has either two points on the top and bottom, or on the left and right side of the square. Without loss of generality we assume that they lie on the top and bottom. Since there are only $O(n-k)$ relevant points left, there are $O((n-k)^2)$ possible squares that we have to consider. Mahapatra et al. use a prune-and-search technique to reduce the number of possible squares by $1/4$ each iteration without computing all these distances explicitly. In each iteration they use the above algorithm for max-Pts-Square(r) as a subroutine. In the end $O(\log(n-k))$ iterations are necessary, resulting in an $O(n + (n-k)\log^2(n-k))$ time algorithm.

3.2 Problems on the circle

When we change our shape from a square to a circle, it changes the problem quite significantly. If we approach this variant in the same way, we achieve algorithms that are less efficient. Suppose we take our

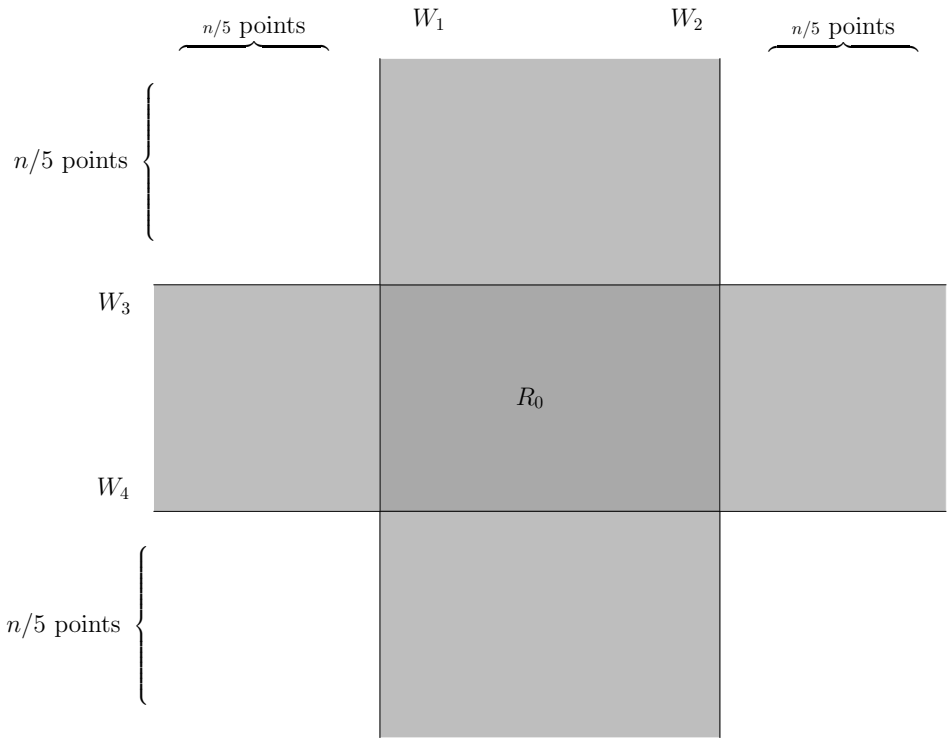


Figure 7: Construction of subproblems.

$O(n \log n)$ algorithm for Max-Pts-Square(r) and try to apply it to Max-Pts-Circle(r). We can still try to find the largest depth in the arrangement that now consists of circles. However, when we considered the arrangement of squares, there were $O(n)$ events that all took $O(\log n)$ time. However, n circles can have $\Omega(n^2)$ intersection points. Note that n squares can also have $O(n^2)$ intersection points, but they only have $O(n)$ different x-coordinates.

Fortunately, there are other approaches we can take, but before we consider those, we want to consider this problem for specific values of k. If $n = k$, this corresponds to the smallest enclosing circle. This problem can be solved in expected $O(n)$ time as follows [14, Chapter 4]. The algorithm takes a random permutation of the points and starts with a circle that encloses the first two points. Then we iteratively add the other points while maintaining the smallest enclosing circle. Suppose we have a circle C at a certain step and we consider a new point p. Then there are two situations possible. In the first, p is already contained in C and we do not have to update it. In the second situation, p is not contained in C. We can show that the new smallest circle has p on its boundary. We determine the new circle by solving the subproblem where we need to find the smallest enclosing circle for a set of points that has p on its boundary. We solve this in a similar fashion as the original problem. We choose a random permutation, start with two points and iteratively add the others. The difference is that our intermediate circles all have p on their boundary. Again in each step there are two possibilities. However, if the new point q lies outside the current circle, we solve the subproblem where we need to find the smallest enclosing circle that has both p and q on its boundary. Again, we take the same approach, until we get the same subproblem, but with three points. This is an easy problem, since there is exactly one such circle. At first glance, this seems like a highly inefficient algorithm, since we iterate through the point set very often. However, similarly to Chan's algorithm we use the fact that it is unlikely that we need to recalculate the circle each iteration. With some analysis we see that each subproblem and the original problem can be solved in expected $O(n)$ time.

Another special case is when $k = 2$ which is equal to the closest pair problem where we just need to find the pair of points that lie closest to each other. It is known that we can find such a pair in $O(n \log n)$ time [34] and that this is also a lower bound by reduction from the element uniqueness problem [7], just as we saw with Min-Seg(k). However, if we assume a slightly stronger model, we can achieve better results. Rabin presented an expected $O(n)$ algorithm assuming a FindBucket function which takes $O(1)$ time [31].

This function can, given a set of predetermined intervals in $\mathbb{R}^1$, find for any real number r to which interval r belongs. Although Rabin focused on the power of randomisation, Fortune and Hopcroft showed that Rabin's result was partly possible because of his stronger model. They showed that, under the same assumptions, we can solve the CLOSEST PAIR PROBLEM in deterministic $O(n \log \log n)$ time [18].

The first technique for the general MIN-CIRCLE(k) we want to discuss is the Voronoi diagram. A Voronoi diagram is a planar subdivision built on a point set P. To avoid confusion we will call the points of P *sites*. Each cell of the diagram corresponds to a site p and contains exactly those points that lie closest to p. Figure 8 shows a Voronoi diagram for a set of sites. We can extend this idea such that a cell corresponds to a set of multiple sites. Suppose we have $p, q \in P$, then we can try to find the set of points for which the two closest sites are p and q. We can subdivide the plane into cells that each correspond to two sites. This is called the second order Voronoi diagram. Generally, in the kth order Voronoi diagram, each cell corresponds to a subset of k sites. It turns out that the solution to MIN-CIRCLE(k) encloses exactly one of these subsets. We can determine this diagram in expected $O\left(k(n-k)\log n + n\log^3 n\right)$ and the complexity of the diagram is $O(k(n-k))$ [3]. Of course, we still need to figure out which cell corresponds to the smallest circle. Before we do that we discuss some properties of the kth order Voronoi diagram that are shown in [23]. First, we note that two adjacent cells differ only in one site. More precisely, if we denote the two point sets that correspond to these cells by A and B, then we have that $A = P' \cup \{a\}$ and $B = P' \cup \{b\}$ for some $P' \subset P$ and $a, b \in P$. The line segment that separates the two cells is part of the bisector of a and b. If we consider a vertex v (not a site!) of the kth order Voronoi diagram, then three cells meet here. There are two possibilities such that each pair neighbouring cells differ in one site. We denote the sets of sites that correspond to these three cells by A, B and C. In the first case we have that $A = P' \cup \{a\}$, $B = P' \cup \{b\}$ and $C = P' \cup \{c\}$ with $P' \subset P$ such that $|P'| = k-1$ and $a, b, c \in P$. The points a, b and c are all equidistant to v. It turns that a circle C centred around v, going through a, b and c encloses all sites of P'. Thus C encloses $k+2$ sites in total. In the second case we have that $A = P' \cup \{a, b\}$, $B = P' \cup \{b, c\}$ and $C = P' \cup \{a, c\}$ where $P' \subset P$ with $|P'| = k-2$. Again, the circle C with centre v going through a, b and c encloses all points of P'. However, C now encloses a set of $k+1$ points.

Now we show how we can find the smallest circle that encloses k points. As we will show later in Lemma 6.0.1 there are two possible cases for this circle. Its boundary contains either three points of P or it contains two points that form a diagonal of the circle. In the first case, it turns out that the centre occurs as vertex in either the $(k-1)$th or the $(k-2)$th order Voronoi diagram. In diagrams we only consider the vertices for which the adjacent cells together correspond to a set of k sites. In the second case we only consider the $(k-1)$th order Voronoi diagram. Then each pair of adjacent cells corresponds together to a set of k sites. Let a and b be the two sites that form the line segment separating the two cells. Then a circle at the midpoint between a and b that passes through a and b encloses k points and has a and b on a diagonal.

To find the solution of the MIN-CIRCLE(k) we only have to determine the $(k-1)$th and $(k-2)$th order Voronoi diagram. The number of vertices and edges in both these diagrams is $O(k(n-k))$. We can solve the problem in $O(k(n-k)\log n + n\log^3(n))$ time using $O(k(n-k))$ space.

A completely different approach was taken by Efrat et al. in [16]. They use the parametric search techinque that was introduced by Megiddo [28]. Since this is a more general technique, we will first explain the method in more depth.

The technique requires a minimisation (or maximisation) problem. More precisely, the problem consists of a predicate $Q : \mathbb{R} \to \{\top, \bot\}$ such that for all $x, y \in \mathbb{R}$ where $x \leq y$ we have that $Q(x)$ implies $Q(y)$. In our case Q reports whether we can enclose k points with a circle of radius x. We are interested in the smallest value x^* for which $Q(x^*)$ holds. Formally, we search for $x^* = \inf\{x \in \mathbb{R} : Q(x)\}$. Suppose we have an algorithm A that takes a parameter x and checks whether $x \leq x^*, x \geq x^*$ or $x = x^*$ in $O(T)$ time. Remember that in the real RAM model, we assume that A only does simple comparisons. Now we determine x^* by pretending to run it as an argument of A. We maintain an interval that will always contain x^*. Initially, this is $\mathbb{R}$. Every time A does a comparison on x^* using a polynomial p, we calculate its roots $x_1, \ldots, x_m$. Then we run A on these values to get a smaller interval for x^*. For example, p gives roots x_1 and x_2 with $x_1 \leq x_2$. Running A on x_1 gives $x^* > x_1$ and running it on x_2 gives $x^* < x_2$. In that case we know that $x^* \in (x_1, x_2)$. This should be enough information to continue the algorithm in the branch of x^*. If we continue, we eventually find the correct value of x^*, since A should be able to find out that x^* is the value

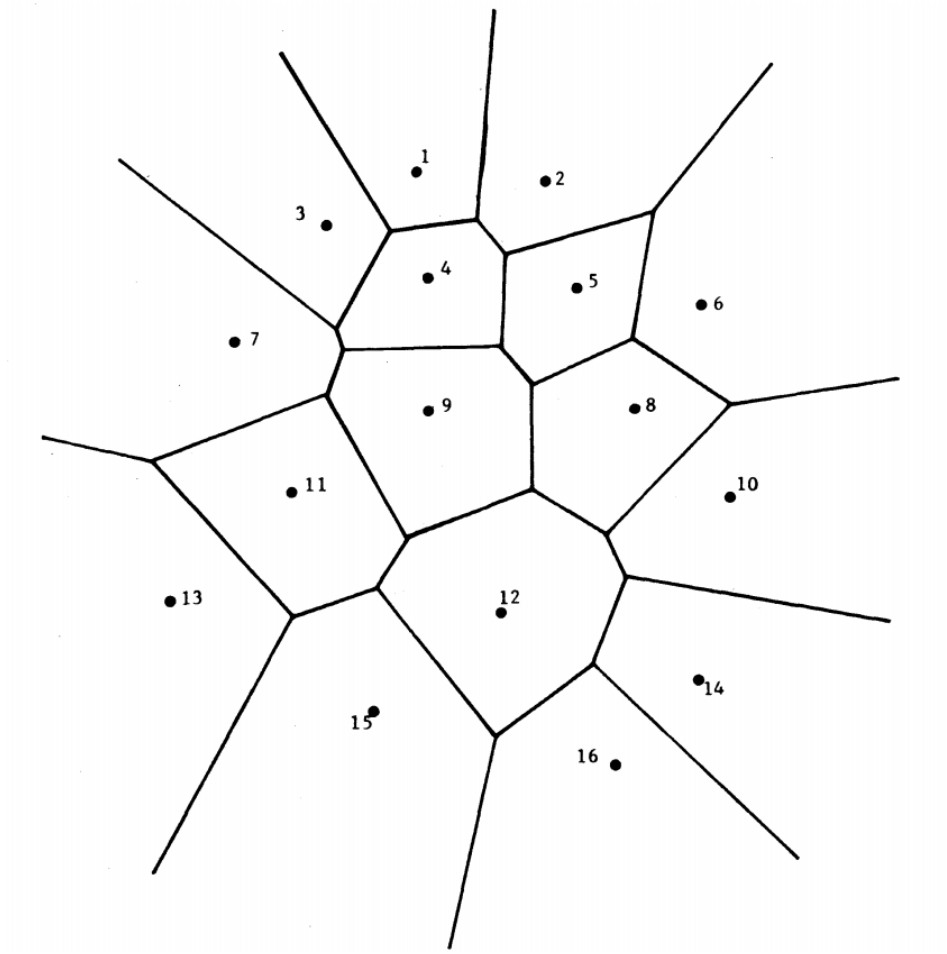


Figure 8: Voronoi diagram for a set of points (taken from [23]).

we are looking for. The total runtime then becomes $O(T^2)$.

We can even improve upon this runtime. If we receive roots $x_1, \ldots, x_m$, then instead of running the algorithm on all roots, we can use a median selection algorithm combined with a binary search to get a runtime of $O(m + T \log m)$ as opposed to the $O(mT)$ runtime for the naive approach. Before we assumed that the number of roots at a certain step is always constant. However, a larger number of roots can help us increase the runtime of the algorithm. One way to achieve this is by considering a parallel algorithm $A_{\mathcal{P}}$. Suppose that $A_{\mathcal{P}}$ runs in $O(T_P)$ when using P processors. If each processor gives us a constant number of roots in each step, the total number of roots per step becomes $O(P)$. With the above technique we can determine which step we should go to next in $O(P + T \log P)$ time. This results in a total runtime of $O(T_P P + T_P T \log P)$ time.

Now Efrat et al. considered the arrangement of circles that we described above, but instead of performing a plane sweep algorithm they observe that if the arrangement of circles has depth at least k, then there must be a point on the boundary of one of the circles that also has depth at least k. They determine for each circle the minimum radius to get a boundary point of depth k using Megiddo's technique. They achieve this in $O(n \log^2 n)$ time, resulting in an overall time complexity of $O(n^2 \log^2 n)$ when computing the radii for all circles. This is their first naive approach. They improve this result by using a Lemma from Pach [35] which states that if there are at least $9nk$ intersecting pairs of circles, then the depth of the arrangement is at least k. The first step then is to find a radius r_{init} such that there are at least $9nk$ intersecting pairs of circles. Then we know that the optimal radius is at most r_{init}. Moreover, if we only consider radii that are at most r_{init}, we know that only the $9nk$ intersecting pairs will be relevant, meaning that we can precalculate the 'neighbours' of each circle, so we do not have to check all possible intersections during the algorithm. This

leads to an algorithm that runs in $O(nk \log^2 n)$ time using $O(nk)$ storage. Although determining r_{init} in the required time is not trivial, we leave these details out. Efrat et al. also present a slightly modified version that runs in $O(nk \log^2 n \log(n/k))$ time using $O(n \log n)$ space.

The final two results we want to discuss, do not consider the exact same general problem as we described it above. They both construct regular grids to partition the point set in smaller sets which is not generally possible. Instead, we either need to make assumptions about the input or we need to add operations to our model, making it stronger, but the results more restrictive. The first paper is by Matoušek [27] and he makes the implicit assumption of bounded spread. He assumes that each pairwise distance in P lies in the range $[\delta/2, 2n\delta]$ where δ is some horizontal or vertical distance between two points. Of course, this does not hold generally, since we can always place vertices further apart. Matoušek's algorithm resembles that of Efrat et al. However, he starts by finding an r_0 such that $k \leq \text{depth}(r_0) = O(k)$ where depth(r) is the maximum depth of the arrangement of circles with radius r (as described earlier). He determines r_0 in $O(n \log n)$ time. As a consequence, a circle intersects at most $O(k)$ other circles. Intuitively, we can see this as follows. Suppose we have a circle C. We can only place a constant number of circles that intersect C such that the depth of that arrangement is equal to 2. So, if we allow the depth to be one higher, we can only add a constant number of circles. If the number of circles only increases by a bounded constant each time, that means that if the depth is $O(k)$, then C intersects with $O(k)$ circles. He goes on to determine r_p^* which is the smallest r such that a circle of radius r around point p contains a point x of depth k in the arrangement of circles of radius r. Because there are only $O(k)$ neighbours, he can determine r_p^* in $O(k \log^2 k)$ expected time. Calculating this value for each $p \in P$ results in an algorithm of $O(n \log n + nk \log^2 k)$ expected time. He improves upon this result by starting with r_0 and filtering out the points q for which $r_q^* \geq r_0$, which he can do efficiently. From the remaining points he picks p at random and calculate r_p^*. Again he filters out some of the points. After doing this $O(\log n)$ times, he ends up with the optimal r^*. Note that this approach greatly resembles the idea of Chan. He ends up with two variants: an $O(n \log n + nk \log k)$ expected time algorithm that use $O(n)$ space and an algorithm that takes $O(n \log n + nk)$ expected time using $O(nk)$ space.

The second paper we want to discuss is by Har-Peled and Mazumdar [21]. They do allow the use of the floor function. Their algorithm consists of many steps. We will highlight some of these steps, but not go into full detail. They start with a simple 2-approximation where they partition the point set in $m = O(n/k)$ horizontal and vertical slabs such that each slab contains at most $k/4$ points. The resulting non-uniform grid has the nice property that the optimal circle must contain one of the grid points. Otherwise, the circle intersects with only one vertical and one horizontal slab resulting in at most $k/2$ points. Then they determine for each of the $O((n/k)^2)$ grid points p the smallest circle that contains k points and has p as its centre. The smallest of these circles is then a 2-approximation of the optimal circle and can be determined in $O(n(n/k)^2)$ time. They improve this running time to expected $O(n(n/k))$ by taking a random sample where they choose each vertex with probability $1/k$. Then for each point p of the random sample they determine the smallest circle with p as its centre that contains k points. If the random sample contains a point of the optimal circle, then we have again a 2-approximation, using a similar reasoning as before. They check whether this is the case using a regular grid such that each grid cell contains a small number of points *if* the sample contains a point of the optimal circle. This succeeds with a constant probability, so after some iterations, they find a suitable sample. They continue with a linear time approximation. Instead of directly calculating the size of the grid, they improve it overal several rounds by starting with a small subset of the points and adding more points iteratively while improving the grid size. After some analysis, this results in an $O(n)$ expected time 2-approximation. They apply then Matoušek's algorithm [27] on certain subsets constructed with the resulting grid to get an expected runtime $O(nk)$ using $O(n + k^2)$ space. They conclude their paper with a $(1+\varepsilon)$-approximation using $O\left(n + n \cdot \min\left(\frac{1}{k\varepsilon^3} \log^2 \frac{1}{\varepsilon}, k\right)\right)$ expected time.

3.3 Problems on the rectangle

In the case of the rectangle, we immediately have multiple variants. Although they seem to differ quite a lot, we will see that some approaches work for more than one variant.

First, we start with the 'simple' problem Max-Pts-Rect(ℓ, w). We see that the same algorithm that we used for Max-Pts-Square(r), can be used here as well. The only thing that changes is the moments at which our events take place. Thus, we can solve this problem in $O(n \log n)$ time using $O(n)$ space.

Compared to the results of the square and the circle, many results of the rectangle variants are more recent. We start with a result by Aggarwal et al. in [4]. They first present a simple $O(n^3)$ time algorithm to compute the MIN-PERIM(k) problem. They use that the four boundaries of the optimal rectangle each contain a point of P. They fix two points of P on the left and lower sides and determine for this specific case the optimal rectangle. This is possible in linear time after presorting the points horizontally and vertically. They use this procedure on all $O(n^2)$ possible lower-left pairs to get the optimal rectangle. Then they also present a different algorithm that is more efficient for small values of k. By computing the $(6k-6)$th order Voronoi diagram this leads to an $O(k^2 n \log n)$ time algorithm using $O(nk)$ space.

Next, we consider a result by Segal and Kedem [33]. Their algorithm is focused on values of k that are at least $n/2$ and is especially efficient if k is close to n. They present an algorithm that solves both MIN-AREA(k) and MIN-PERIM(k) in $O(n + k(n-k)^2)$ time using $O(n)$ space. They start with a simpler problem where we want to find the smallest rectangle that contains k x-consecutive points. Note that this can easily be computed by sorting the points by x-coordinate in $O(n \log n)$ time. Segal and Kedem, however, only sort the x-largest (so rightmost) $n-k$ points. Then they perform a sweep line on the leftmost points, while maintaining the top, bottom, left and right side of the rectangle using tournament trees. This results in a $O(k + (n-k)\log(kn))$ time algorithm which is indeed faster than the naive approach for large values of k. For the actual problem they perform a sweep line from the left as well. The main difference is that they do a second sweep line per event starting at k points to the right. So, if the first sweep line is at the ith point starting from the left, they start the second sweep line at the $(i+k-1)$th point. These two points will be the left and the right side of the rectangle. With similar tournament trees and some additional structures, they can determine the smallest rectangle for each pair points from the previous event quite efficiently. This leads to a $O(n + k(n-k)^2)$ time algorithm that uses $O(n)$ space.

The previous results were quite old. Now we want to consider some more recent papers. De Berg et al. [15] studied the case where k is small, but specifically for MIN-AREA(k). They use a divide-and-conquer approach. They draw a horizontal line ℓ, splitting P in two equally sized sets, and consider the cases where the optimal rectangle either intersects ℓ or it lies completely above or below ℓ for which they go in recursion. Now let $\Phi(Q, q, k)$ be the area of the minimum area rectangle R such that R contains k points of Q and q lies either on the top or the bottom side of R. De Berg et al. show that they can compute this value in $O(|Q|^2)$ time. Then, in the case that the optimal rectangle intersects ℓ, they determine for each $p \in P$ a set Q_p where $|Q_p| = O(k)$ such that $\min\{\Phi(Q_p, p, k) \mid p \in P\}$ equals the area of the optimal rectangle. With a series of lemmas they show that the defined Q_p has indeed these properties and that they can compute the minimal area rectangle in $O(nk^2 \log n + n \log^2 n)$ time. This is nearly linear for small values of k. They end their paper with a randomised approximation algorithm for MAX-PTS-AREA(α). It returns with high probability in $O((n/\varepsilon^4)\log^3 n \log(1/\varepsilon))$ time a rectangle of at most area α that covers $(1-\varepsilon)\kappa^*$ points where κ^* is the maximum possible number of points that can be covered by a rectangle of area at most α.

Kaplan et al. [22] were the first to break the cubic barrier. An important comment is that they use a different model as they use floor functions and hash tables in their algorithms. They find an $O(n^{5/2} \log n)$ algorithm that works for MAX-PTS-AREA(α), MAX-PTS-PERIM(α) and MAX-PTS-DIAG(α). For the latter two problems they improve their algorithm to get a k-sensitive runtime of $O(nk^{3/2} \log k)$. They continue with a $(1-\varepsilon)$-approximation for MAX-PTS-AREA(α) that runs in $O\left(n + \frac{n}{k\varepsilon^5} \log^{5/2}\left(\frac{n}{k}\right) \log\left(\frac{1}{\varepsilon} \log\left(\frac{n}{k}\right)\right)\right)$. They end with an $O(nk^{3/2} \log k \log n)$ time algorithm for MIN-PERIM(k) by using their algorithm for MAX-PTS-PERIM(α) a logarithmic number of times.

Another recent result is by Chan and Har-Peled [10] where they studied both the MIN-AREA(k) and MIN-PERIM(k) problem. Their result is especially significant as it is the first near quadratic time algorithm for these problems. Previous algorithms had not yet accomplished this for certain values of k. They start with an $O(n^2 \log n)$ time algorithm using a divide-and-conquer approach. During their algorithm they keep two horizontal slabs σ and τ each containing q points where σ lies above τ or $\sigma = \tau$. The subproblem they consider each time is the minimum rectangle (by area or perimeter) such that the top side lies in σ and the bottom side in τ. They solve this problem by splitting both σ and τ in two equally sized horizontal slabs respectively σ_1 and σ_2 and τ_1 and τ_2. Then they consider the four possible pairs between the smaller slabs. They manage an $O(q^2)$ space datastructure. This data structure is built on a set of n 1D points of whom q are marked. They can delete or unmark marked points and report the shortest interval containing k points

all in $O(q)$ time. They use this data structure for the slabs to quickly remove points that lie above σ or below τ. The marked points are those that lie inside the slab and a point is unmarked when it moves to the space between the two slabs. Since there are q points in the slabs, updating the data structure when splitting the slabs can be done in $O(q^2)$ time. This results in a total runtime of $O(q^2 \log q)$ for the subproblem. In the original problem they start with $\sigma = \tau$ being the complete set of points. Then $q = n$, so their algorithm runs in $O(n^2 \log n)$ time. A known reduction of the minimum perimeter problem where the problem is reduced to $O(n/k)$ instances of size $O(k)$ results in an $O(n \log n + nk \log k)$ time algorithm. They also present a reduction themselves for the area problem which results in $O((n/k) \log(n/k))$ instances of size $O(k)$. This leads to an $O(nk \log(n/k) \log k)$ time algorithm. Chan and Har-Peled continue with many other results that follow from their algorithm. The most important one, relating to this thesis, is the $(1+\varepsilon)$-approximation for Min-Area(k) that runs in $O((1/\varepsilon^3) \log(1/\varepsilon) \cdot n \log n)$ expected time.

4 1-dimensional variant

In this section we consider the 1-dimensional problems. For some point set P in $\mathbb{R}^1$, we want to solve the Min-Seg(k) and the Max-Pts-Seg(r) problems. Since the points of P have an ordering, statements like 'p is larger than q' are well-defined. We first start with a direct algorithm for both of them.

Lemma 4.0.1. *Let P be a set of n points in $\mathbb{R}^1$ and let k be an integer such that $1 \leq k \leq n$. We can solve the* Min-Seg(k) *problem, i.e. finding the smallest interval containing k points, in $O(n + (n-k)\log(n-k))$ time using $O(n)$ space.*

Proof. We start with an $O(n \log n)$ time algorithm. We sort the points in ascending order. We denote the points of P by $p_1, p_2, \ldots, p_n$ where p_i is the ith point in the sorted set. It is evident that the smallest interval containing k points has a point on both its left and right boundary. Otherwise, we can shrink the interval, while not reducing the number of points it contains. Now we consider the intervals $[p_1, p_k], [p_2, p_{k+1}], \ldots, [p_{n-k+1}, p_n]$ and we determine which one is the smallest. We have $O(n)$ such intervals and determining the length of each interval can be done in $O(1)$. The sorting step is the most expensive and we achieve an $O(n \log n)$ time algorithm that uses $O(n)$ space.

To get to a k-sensitive running time, we use the following fact. If $k > n/2$, then the points $p_{n-k+1}, \ldots p_k$ are contained in any interval that contains k points. It is thus not necessary to sort these points. We only need to sort the first $n-k$ and the last $n-k$ points. We can find these sets in $O(n)$ time using quickselect with median of medians to select a pivot. Then we only need to sort two sets of size $n-k$, resulting in a running time of $O(n + (n-k)\log(n-k))$. □

For the Max-Pts-Seg(r) we can achieve very similar results.

Lemma 4.0.2. *Let P be a set of n points in $\mathbb{R}^1$ and let r be a positive real number. We can solve the* Max-Pts-Seg(r) *problem, i.e. finding an interval of length r that contains the maximum number of points of P, in $O(n \log n)$ time using $O(n)$ space.*

Proof. The approach is almost identical to that of the Min-Seg(k) problem. We start by noting that we can assume that the interval that contains the most points has a point on its left boundary. We sort the points of P and let $p_1, \ldots, p_n$ be such that p_i is the ith point in the sorted set. We begin with the interval $[p_1, p_1 + r]$ and we determine the number of points in it by binary searching the value $p_1 + r$ in the sorted set P. Let p_j be the largest point of P less than or equal to $p_1 + r$. Next, we consider the interval $[p_2, p_2 + r]$. We can now find the largest point of P less than or equal to $p_2 + r$ by iterating over the points of P in ascending order starting from p_j. Similarly, we can determine the number of points in the intervals $[p_3, p_3 + r], \ldots, [p_n, p_n + r]$. If we consider these intervals as a sliding interval, we see that the right side crosses each point in P at most once. So determining the number of points in the intervals $[p_1, p_1 + r]$ to $[p_n, p_n + r]$ can be done in $O(n)$ time after we have sorted the points. Finally, we report the interval that contained the most points. Again, sorting is the most expensive step, so the algorithm runs in $O(n \log n)$ time. □

We can even show that $\Theta(n \log n)$ time is a lower bound to solve the Min-Seg(k) problem for certain value of k. If $k = 2$, then we can reduce the element uniqueness problem to the Min-Seg(2) problem. The element uniqueness problem is the problem of determining whether there exists a duplicate in a list of numbers. It is

known that $\Theta(n \log n)$ is a lower bound for this problem [6]. Now consider a larger (but constant) value of k. We can reduce MIN-SEG(2) to MIN-SEG(k) by creating $\lceil k/2 \rceil$ copies of every point. Then we can solve MIN-SEG(2) by solving MIN-SEG(k) on this new problem. It follows that MIN-SEG(k) is lower bounded by $\Theta(n \log n)$ for a constant k.

The approach of Lemmas 4.0.1 and 4.0.2 imply directly a data structure variant that we can use. In both lemmas we start with sorting the point set P. This step is identical for every value of k and r in the problems MIN-SEG(k) and MAX-PTS-SEG(k) respectively. After sorting the point set, we only need linear time to consider the relevant intervals.

Corollary 4.0.3. *Let P be a set of n points in $\mathbb{R}^1$. We can preprocess the set P in $O(n \log n)$ time in an $O(n)$ space data structure such that we can answer the* MIN-SEG(k) *and* MAX-PTS-SEG(r) *problems in linear time for any integer k such that $1 \leq k \leq n$ and for any positive real r in $O(n)$ time.*

We might wonder if we can achieve a faster query time than the result of Corollary 4.0.3. One way to achieve this is to precalculate the solution of MIN-SEG(k) for all values of k and store those values in an array A. We can then report the solution in $O(1)$ time. Using Corollary 4.0.3, we can compute these values then in $O(n^2)$ time. We can also answer the problem MAX-PTS-SEG(r) in $O(\log n)$ time using a binary search on A.

Of course the above result is not very useful, but it does give us a goal for further improvements. A reasonable question is whether we can achieve a sublinear query time using subquadratic preprocessing time. Such a result might be useful when n is too large to allow for quadratic algorithms, but we still want fast query times. Although we do not have concrete results, we have some ideas and conditional results that we want to present.

4.1 Reductions of the Min-Seg(k) problem

We begin with introducing a different problem. Let P be a set of n points in $\mathbb{R}^1$, t a point in $\mathbb{R}^1$ and k an integer such that $1 \leq k \leq n$. We define MIN-SEG-WITH-POINT(k, t) as the problem of finding the smallest interval that contains k points of P while also containing the point t. We can show the following conditional theorem.

Theorem 4.1.1. *Let P be a set of n points in $\mathbb{R}^1$. Suppose we have an algorithm $\mathcal{A}$ that can compute for some $t \in \mathbb{R}^1$ the solution to* MIN-SEG-WITH-POINT(k, t) *for all values of $1 \leq k \leq n$ in $f(n)$ time where $f(n) = o(n^2)$. Then we can compute the solution to* MIN-SEG(k) *for all values of $1 \leq k \leq n$ in subquadratic time.*

Proof. We start by sorting the point set P from left to right and we denote the points in this order by $p_1, \ldots, p_n$. We use the following observation. For any point $t \in \mathbb{R}^1$ and value of k, there are three cases possible for the k-smallest interval:

1. the k-smallest interval lies completely to the left of t;
2. the k-smallest interval lies completely to the right of t;
3. the k-smallest interval contains the point t.

If we choose $t = \lfloor n/2 \rfloor$ we can use a divide-and-conquer algorithm. For the first two cases we compute recursively the solution to the MIN-SEG(k) problem for $1 \leq k \leq n/2$. We then get the smallest intervals containing only points to the left and to the right of t which we will denote by $L(k)$ and $R(k)$ for $1 \leq k \leq n/2$ respectively. Then we use algorithm $\mathcal{A}$ to compute smallest interval that contains t for $1 \leq k \leq n$. We denote these intervals by $C(k)$ for $1 \leq k \leq n$. For $k > n/2$ this is automatically the k-smallest interval as well. For $k \leq n/2$ we have that the k-smallest interval is equal to the smallest interval among $L(k)$, $R(k)$ and $C(k)$.

Now we consider how much time this algorithm uses. Let the running time of this algorithm be denoted by $T(n)$. We spend $2T(n/2)$ on the two recursive calls. Since $\mathcal{A}$ outputs n values, we know that $f(n) = \Omega(n)$. The rest of the algorithm uses linear time. We conclude that the running time meets the following recursive relation

$$T(n) = 2T(n/2) + f(n).$$

To analyse the runtime, we distinguish two cases. If $f(n) = O(n^{1+\varepsilon})$ for some constant $\varepsilon > 0$, then we know that $T(n) = O(n^{1+\varepsilon}) = o(n^2)$. If $f(n) = \Omega(n^{1+\varepsilon})$, there is polynomial difference between $n^{\log_2(2)}$ and $f(n)$ and we can apply the master method [13]. The running time is then $O(f(n)) = o(n)$. □

The above theorem helps us reduce the problem to a different problem that might be easier to solve. We want to mention that there is a reverse relation as well.

Theorem 4.1.2. *Suppose we have an algorithm $\mathcal{B}$ that can compute, for any point set P of n points in $\mathbb{R}^1$, the solution to* Min-Seg(k) *for all values of $1 \leq k \leq n$ in $f(n)$ time. Let P be a set of n points in $\mathbb{R}^1$ and $t \in \mathbb{R}^1$. We can compute the solution to* Min-Seg-With-Point(k, t) *for all values of $1 \leq k \leq n$ in $f(2n)$ time.*

Proof. We define the (multi)set T as $\{t, \ldots, t\}$ consisting of n duplicates of t. Let $P' = P \cup T$. If $\mathcal{B}$ requires a point set without duplicates, we can choose n distinct points in $[t - \varepsilon/2, t + \varepsilon/2]$ where $\varepsilon > 0$ is such that $|p - q| > \varepsilon$ for any $p, q \in P$. We know have a set P' of $2n$ points. If $k > n$, then it must contain at least one point of T and since all points of T lie inside an interval of size ε, we know that the smallest interval containing k points contains all points of T. It follows that the solution of Min-Seg(k) for $k > n$ contains the same points of P as the solution of Min-Seg-With-Point(k, t). □

4.2 Ideas for the Min-Seg-With-Point(k, t) problem

Now we discuss our idea for a data structure for Min-Seg(k, t). Specifically, for the variant where t splits P in two sets A and B such that $|A| = |B| = n/2$. For simplicity we assume that P contains an even number of points. Let a_i and b_i be the ith point from the sets A and B of points to the right and left respectively, starting from t and let $a_0 = b_0 = t$ by convention. We start by finding the optimal segments where we only allow the points of B. Note that this only works if $k \leq n/2$, but we ignore that for now. The idea is to iteratively add the points of A, starting from a_1, while maintaining the smallest segments for each k. We denote the solution to Min-Seg-With-Point(k, t) over the set $B \cup \{a_1, \ldots, a_i\}$ by $I_{i,k}$. Then $I_{n/2,k}$ equals the solution to Min-Seg-With-Point(k, t) over P. At the start, we have $I_{0,k} = [t, b_k]$. If we add a_1 to our point set, then some of the segments will 'shift' to the left as there is a new smallest segment. In other words, for those values of k, we have that $I_{1,k} \neq I_{0,k}$. More precisely, exactly those intervals whose final part is larger than $t - a_1$. The process is visualised in Figure 9. This immediately presents us with a problem as there is no structure that determines which segments shift, i.e. the k values do not form a connected interval for example. In fact, we can show the following.

Lemma 4.2.1. *Let $[j] = \{1, \ldots, j\}$ and $E \subseteq [n/2]$. Then we can find a point set P and $t \in \mathbb{R}$ as we described above such that $I_{0,k}$ shifts when we add a_1, if and only if $k \in E$. More precisely, we have that $I_{0,k} \neq I_{1,k}$ if and only if $k \in E$.*

Proof. Let $t = 0$. In accordance with the above notation we define P by means of A and B. Let $a_1 = -2$. The other elements of A are irrelevant, but for completion, we define $a_i = -2 - i$ for $2 \leq i \leq n/2$. We define $\Delta b_k = 3$ if $k \in E$ and $\Delta b_k = 1$ if $k \in [n/2] \setminus E$. Now let $b_k = t + \sum_{j=1}^{k} \Delta b_j$. At the start of the algorithm, we have that $I_{0,k} = [t, b_k]$. If we add a_1, then we shift those segments for which $[a_1, b_{k-1}]$ is a smaller segment than $[t, b_k]$. That is of course when $t - a_1 \leq b_k - b_{k-1} = \Delta_k$ which holds if and only if $k \in E$. □

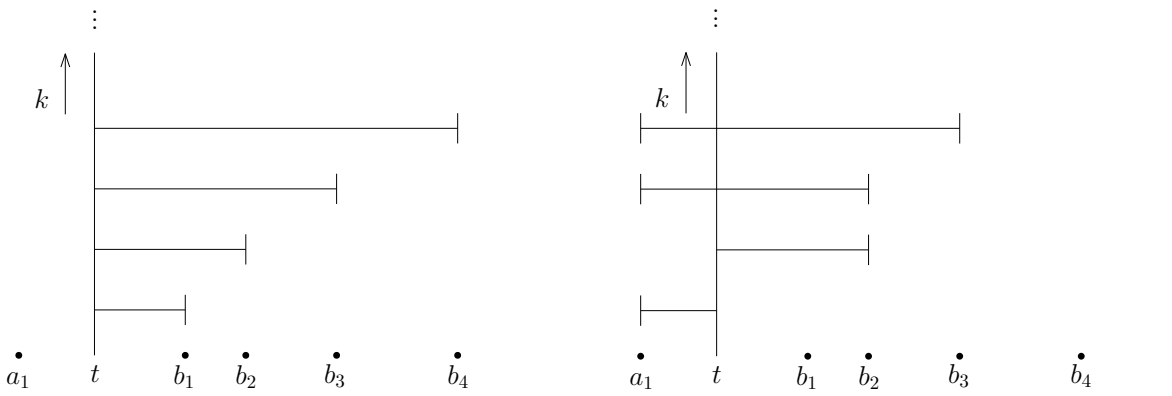
In accordance with the above notation, we define for sets A and B the following variables. Let $\Delta b_i = |b_i - b_{i-1}|$ and $\Delta a_i = |a_i - a_{i-1}|$. In that case it holds that $b_k = t + \sum_{j=1}^{k} \Delta b_k$ and $a_k = t - \sum_{j=1}^{k} \Delta a_k$. We will call these intervals between two points *subsegments*.

So initially, there does not seem to be any structure that we can exploit to keep track of which segments have shifted. We also cannot move all intervals explicitly each step, since it is possible that this becomes too expensive. Note that if we must shift the kth interval when we add a_i, then this means that $I_{i,k} \neq I_{i-1,k}$. If we were to move each interval explicitly, then this would cost $\Omega(n^2)$ time which follows from the following lemma.

Lemma 4.2.2. *For any $n \in \mathbb{N}$, there exists a set P of n points in $\mathbb{R}^1$ and a $t \in \mathbb{R}$ such that the number of shifts over all segments is $\Theta(n^2)$. More precisely, we have that $|\mathcal{I}| = \Theta(n^2)$ where $\mathcal{I} = \{I_{i,k} \mid 0 \leq i \leq n/2, 1 \leq k \leq n\}$.*

Proof. Let $t = 0$. We define $a_i = -i$ and $b_i = i/2$ for $1 \leq i \leq n/2$. We see that $I_{i,k} = [a_i, b_{k-i}]$ if $k \geq i$. This immediately shows that $|\mathcal{I}| = \Theta(n^2)$. □

The above result prevents us from just shifting all segments each step explicitly. We did see that only those segments for which $\Delta b_k > \Delta a_1$. Our main idea is to store these Δb_k in the leaves of a binary search tree (BST) sorted by length. This way, we can store which segments are shifted by marking the highest nodes in the BST such that all leaves in that subtree are shifted, similarly to how segments are stored in a segment tree. When all points of A are added, we can determine how often a segment is shifted.



(a) At the start all intervals begin at t

(b) When adding p_1^l, some of the intervals shift to the left

Figure 9: Example of shifting intervals.

However, we encountered several problems that can occur. First of all, if a certain segment does not shift, when adding a_1, then it might still be the case that it shifts two steps, when adding a_2. This makes it more complicated and we can construct an example where this happens on a larger scale.

Example 4.2.3. We want to achieve that for some k we shift its segment one step when we add a_1 until a_T where $T = O(n)$. Then for $k+1$, we do not shift it when we add a_1, but we shift it two times, when adding a_2. From then it also shifts one step each time until a_T. This pattern continues until $k+T$ for which we do not shift its segment when adding a_1 up to a_{T-1}. Then when we add a_T, we shift it T steps at once. Such an example shows that segments can jump an arbitrary number of steps and that we can have a linear number of different step sizes.

For simplicity, we assume that n is a multiple of 4, but with some rounding it works for other values n as well. Let $\Delta b_k = \Omega - k$ for some large $\Omega \gg n$. Then we define $\Delta a_k = \Omega - n/4 - (k-1) + \varepsilon$ for some small but positive ε. Note that $\Delta a_1 > \Delta b_{n/4}$, but $\Delta a_1 < \Delta b_k$ for all $k > n/4$. This means that if we add a_1, then for $k = n/4$ we need to shift the segment to the left, but for larger k this does not happen. Similarly, if we add a_2, we get that $\Delta a_1 + \Delta a_2 > \Delta b_{n/4} + \Delta b_{n/4+1}$, so for $k = n/4 + 1$, we shift the segment two steps at once. For $k > n/4 + 1$ the segment does not move and for $k = n/4$ the segment moves another step. Table 2 shows the number of shifts for the intervals with $n/4 \leq k \leq n/2$ when we add the points a_1 to $a_{n/4}$. It is clear that this pattern continues, until $k = n/2$ that moves $n/4$ steps at once. △

k	a_1	a_2	a_3	a_4	…	$a_{n/4}$
$n/4$	1	1	1	1	…	1
$n/4+1$	0	2	1	1	…	1
$n/4+2$	0	0	3	1	…	1
…						
$n/2$	0	0	0	0	…	$n/4$

Table 2: The number of shifts we perform on the k-smallest intervals when we add a_i.

A different problem we have not addressed yet is that the above divide-and-conquer step only works for values of $k \leq n/2$. We could solve this by starting all intervals for $k > n/2$ completely to the right. The

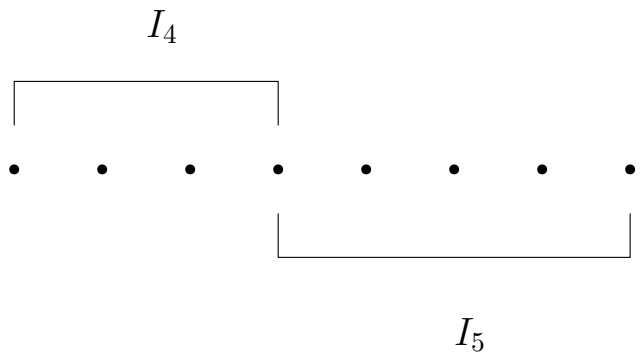


Figure 10: This situation cannot occur

intervals then all start at a unique point of A and each time we add a point from A, we add the corresponding interval to the data structure.

4.3 Interesting examples

Although it is difficult to prove, we expect it to be impossible to find a data structure that is able to answer the Min-Seg(k) problem for all $1 \leq k \leq n$ in subquadratic time. The main argument here is that there are $O(n^2)$ candidate intervals, meaning there exist $O(n^2)$ intervals that have a point of P on their left and right endpoint. Any such interval contains a certain number of points of P and could therefore be the smallest one. If there is an algorithm that can determine all k-smallest intervals in subquadratic time, it cannot consider all these candidates explicitly. One way to avoid this is to find a relation between the k-smallest intervals. For example, if, given a small set of the k-smallest intervals, reduces the number of possibilities of other k-smallest intervals, we might not need to consider each of the $O(n^2)$ candidate intervals. We have found one such relation that might be useful.

Lemma 4.3.1. *Let P be a set of n points in $\mathbb{R}^1$ and let k and k' be integers such that $1 \leq k, k' \leq n$ and $k \neq k'$. Let I_k and $I_{k'}$ be the k- and k'-smallest intervals respectively. If I_k and $I_{k'}$ are unique, then $|I_k \cap I_{k'} \cap P| \neq 1$. This means that I_k and $I_{k'}$ cannot share exactly one point of P. So either they are disjoint or they overlap on multiple points of P.*

Proof. Figure 10 shows an example of the situation that we claim impossible. We use a proof by contradiction. Without loss of generality we assume that $k < k'$ and that I_k lies to the left of $I_{k'}$. We denote the points of P contained by I_k from right to left by $p_1, \ldots, p_k$. Similarly, we denote the points of P contained by $I_{k'}$ from left to right by $q_1, \ldots, q_{k'}$. Note that $p_1 = q_1$. The length of I_k is now $|p_k - p_1|$ and the length of $I_{k'}$ is $|q_{k'} - q_1|$. Because I_k is the k-smallest interval and I_k is unique, we know that $|p_k - p_1| < |q_{k'} - q_{k'-k+1}|$. But then we have that

$$|p_k - q_{k'-k+1}| = |p_k - p_1| + |q_1 - q_{k'-k+1}| < |q_{k'} - q_{k'-k+1}| + |q_1 - q_{k'-k+1}| = |q_{k'} - q_1|.$$

It follows that the interval $[p_k, q_{k'-k+1}]$ is smaller than the the interval $[q_1, q_{k'}]$. This is of course a contradiction with the assumption that $I_{k'}$ was the unique k'-smallest interval.

□

Next, we want to show some examples that could be used to check other relations. We might suspect that many of the k-smallest intervals overlap with each other. Then if we know some of the k-smallest intervals, this could reduce the search for others. For example, if we have a set P' that contains k points of P that lie relatively close together compared to the other points of P. Then it is not unlikely that the k'-smallest intervals with $k' \leq k$ also contain points of P'. This is all quite informal and not meant to describe a specific relation between the k-smallest intervals. On the contrary, we now show some examples that illustrate that the above intuition does not always hold.

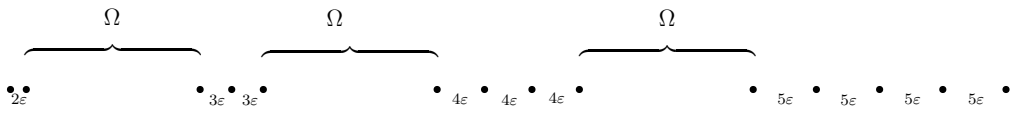


Figure 11: Illustration of Example 4.3.2.

Example 4.3.2. For any value of $n \in \mathbb{N}$, there exists a set P of n points in $\mathbb{R}^1$ such that the k-smallest intervals for $2 \leq k \leq \lfloor\sqrt{n}\rfloor$ are pairwise disjoint. The idea is to create for each $2 \leq k \leq \lfloor\sqrt{n}\rfloor$ a set P_k of k points such that if $p, q \in P_k$ for some k, then they lie extremely close, but if $p \in P_k$ and $q \in P_{k'}$ where $k \neq k'$, then lie far apart. For $2 \leq k \leq \lfloor\sqrt{n}\rfloor$, we define

$$P_k = \{k\Omega + jk\varepsilon \mid 1 \leq j \leq k\}$$

where ε is some small constant and $\Omega > n/\varepsilon$. Note that P_k contains k points such that two consecutive points are at a distance $k\varepsilon$. This ensures that the k-smallest interval with $2 \leq k \leq \lfloor\sqrt{n}\rfloor$ always encloses P_k and not some set of points of $P_{k'}$ with $k' > k$. To ensure that the k-smallest does not enclose multiple smaller $P_{k'}$, we place the groups at a huge distance of almost Ω. Figure 11 shows this construction for a small set of points. It follows that if we take $P' = \bigcup_{2\leq k\leq\lfloor\sqrt{n}\rfloor} P_k$, then the k-smallest intervals of the set P' for $2 \leq k \leq \lfloor\sqrt{n}\rfloor$ are disjoint and $|P'| \leq n$. Now we only need to ensure that we have a set of n points. Let $s = n - |P'|$. We define $P_1 = \{-j\Omega \mid 1 \leq j \leq s\}$ where Ω is defined in the same way as before. Now, let $P = P_1 \cup P'$. Neither of the k-smallest interval with $2 \leq k \leq \lfloor\sqrt{n}\rfloor$ contains any of the points of P_1, since the distance from a point in P_1 to any other point of P is always at least Ω. △

Example 4.3.3. For any $n \in \mathbb{N}$, there exists a set P of n points in $\mathbb{R}^1$ such that there are ρn values k such that the k-smallest interval is disjoint from the $(k+1)$-smallest interval. For simplicity we assume n is divisible by 12. Otherwise, we can place a few points very far away just as in Example 4.3.2. The example we will now describe is shown in Figure 12. We divide P in two sets P_1 and P_2 both containing $n/2$ points. The points of P_1 are divided into groups of two points such that the distance between two consecutive groups is some large constant Ω. The two points in such a group are placed at distance 1. The points of P_2 are divided into smaller groups of three points such that the distance between two consecutive groups is $\frac{3}{2}\Omega$. Again, the three points in a small group are placed at distance 1 between consecutive points. Lastly, the two sets P_1 and P_2 are placed at a distance Ω^2.

The large distance between P_1 and P_2 ensures that the k-smallest interval only contains points of either P_1 or P_2, but not both, if $k \leq n/2$. Let k be such that $k+1 < n/2$ is an odd multiple of 3. We can write $k+1 = 3(2j+1) = 6j+3$ for some $j \in \mathbb{N}$. Then $k = 6j+2$. Using only points in P_1, we can find an interval of length $3j\Omega + (3j+1)$ which contains $3j+1$ groups of two points. If we only can use points of P_2, then the interval needs to overlap with at least $2j+1$ groups of three points. From the rightmost group we only need to include two points. The interval has then length $\frac{3}{2}\Omega \cdot 2j + 2(2j+1) - 1 = 3j\Omega + 4j + 1$. For any $j \geq 1$ we have that the k-smallest interval only contains points of P_1.

Now we consider the smallest interval for $k+1$. If we can only points from P_1, then the smallest interval containing $k+1$ points covers $3j+1$ groups of two points entirely and one group partly. The length of this interval becomes $(3j+1)\Omega + (3j+1)$. If we only use points of P_2, then we can take an interval that covers exactly $2j+1$ groups of three points. The length of this interval is then $\frac{3}{2}\Omega \cdot 2j + 2(2j+1) = 3j\Omega + 4j + 2$. Thus, we conclude that the $(k+1)$-smallest interval only uses points of P_2. △

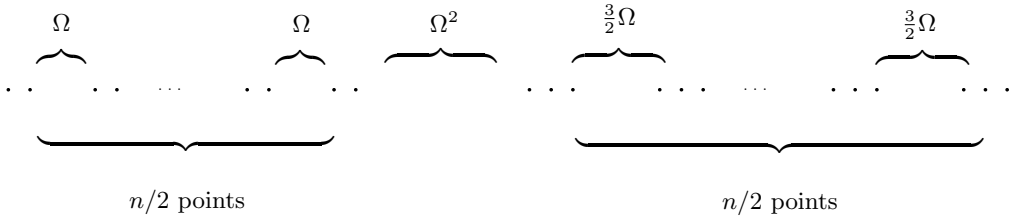


Figure 12: Illustration of Example 4.3.3.

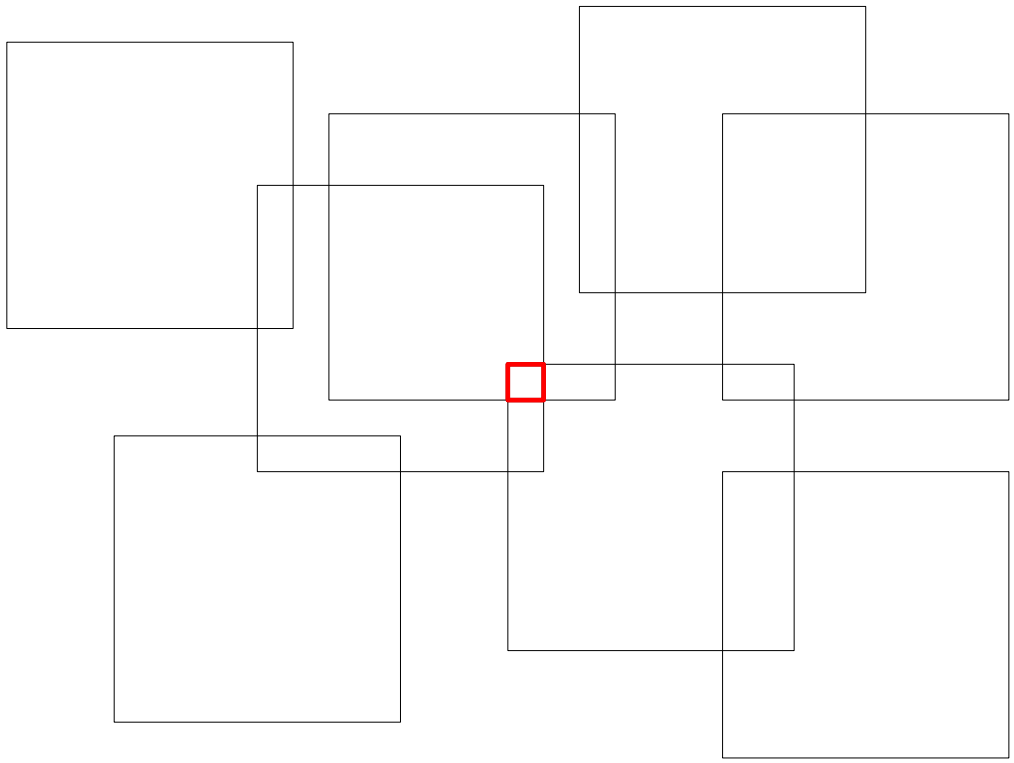


Figure 13: An arrangement of squares. The area marked in red has depth 3 which is the maximum in the arrangement.

Of course the above examples do not prove any sort of lower bound on the running time. It is only an indication that finding a relation between k-smallest intervals is difficult. Moreover, these examples can be useful to test claims that we might conjecture.

5 A data structure for the square

In this section we aim to construct a data structure that solves the problems Max-Pts-Square(r) and Min-Square(k). Let P be a set of n points in $\mathbb{R}^2$. We start by focusing on the Max-Pts-Square(r) problem. Before we describe this data structure, we revisit the algorithm that solves the problem directly in Section 5.1. Then in Section 5.2 we construct a data structure that allows for a faster query time than solving the problem. This requires extra assumptions on the computational model which we relieve in Section 5.3 by making an assumption on the input.

5.1 A direct algorithm for Max-Pts-Square(r)

As our algorithm and data structure are based on a specific algorithm for the Max-Pts-Square(r), we now explain it in great detail. Instead of finding a square of size r that encloses the maximum number of points, we place a square $Sq(p)$ around each point p of P. This results in an arrangement of squares for which we have to find its maximum depth, the maximum number of overlapping squares. The situation is shown in Figure 13. The algorithm to compute the depth of the arrangement is a sweep line algorithm that uses a dynamic segment tree as a status structure.

Let $I = \{[a_i, b_i] \mid 1 \leq i \leq m\}$ be a set of m intervals in $\mathbb{R}^1$. A segment tree [14, Chapter 10] is a binary tree data structure that can store this set I. If we only consider the endpoints of the intervals of I, then $\mathbb{R}^1$ is split into several smaller intervals, which we call *elementary segments* to avoid confusion with the intervals of I. These segments are stored in ascending order in the leaves of the segment tree. So the leftmost leaf corresponds to the leftmost elementary segment. Note that two leaves that share the same parent contain

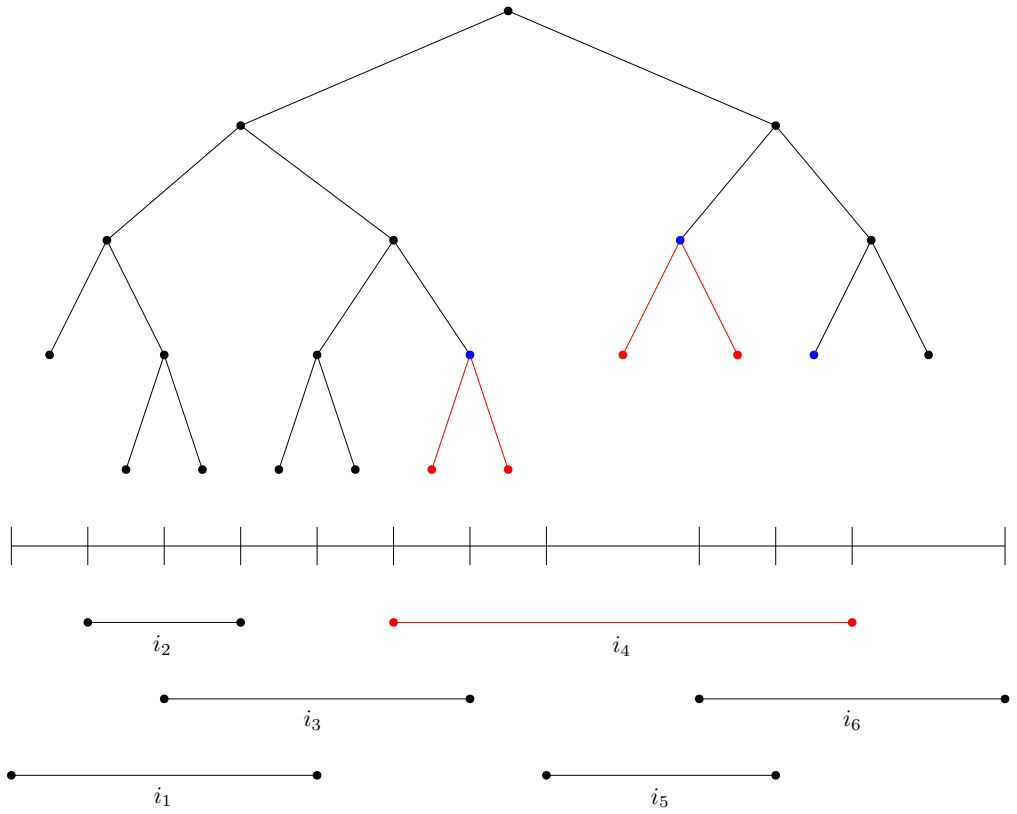


Figure 14: A segment tree on a set of six intervals. The interval i_4 is stored in three nodes.

segments that are adjacent. Also note that the jth endpoint is the left side of the jth leaf. An internal node stores a *segment* as well. This segment is defined as the union of the leaves in its subtree. To refer to the segment of a node v, we will simply use the notation v when it is clear we mean the segment stored at v. Figure 14 shows an example of a segment tree on a set of intervals. We can represent the interval i_4 using only three nodes of the segment tree. We chose the nodes whose segments are contained in i_4, but for which the parents are not. In general any interval i can be represented by $O(\log n)$ nodes this way [14, Lemma 10.10]. We call this the *canonical set* of i which we denote by $I(i)$.

In a static segment tree, we store each interval of I in the $O(\log n)$ nodes of its canonical subset. However, in a *dynamic* segment tree the tree is only built on the fixed set of endpoints of the intervals of I, but no intervals are stored yet. We can insert an interval in the segment tree by storing it in the nodes that form its canonical subset. We can also remove an inserted interval again. Both operations run in $O(\log n)$ time.

Now we go back to our sweep line algorithm. We move a horizontal sweep line from top to bottom over the squares. If we project the squares on the sweep line, we get a set of intervals. During the sweep we insert intervals in the segment tree, when we reach the top of a square and we remove it from the segment tree, when we hit the bottom. Because we know exactly where the top and bottom of each square is, we can sort these events in top-down order, before we start the sweep.

We now explain how the insertion and removal work. A small difference with the dynamic segment tree is that we do not store the intervals explicitly in each node. Instead, for a node v in the segment tree we store two values:

- n_v, the number of intervals that cover the segment of v, but not that of its parent;
- d_v, the maximum depth inside the segment of v of the arrangement induced by the intervals that do not cover v.

An insertion of an interval i then finds the canonical subset of i. For $v \in I(i)$ we increase n_v by 1. To

update the d_v, we use the following lemma.

Lemma 5.1.1. *Suppose we have a segment tree with a set of intervals I inserted. Suppose v is some internal node and let $l(v)$ and $r(v)$ be its left and right child respectively. Then $d_v = \max(d_{l(v)} + n_{l(v)}, d_{r(v)} + n_{r(v)})$.*

Proof. Let I' be the set of intervals in the segment tree that do not cover v. Then d_v equals the maximum depth of the arrangement of I' inside the segment v. Note that segment v is equal to the union of the segments $l(v)$ and $r(v)$. Then d_v equals the maximum of the maximum depths of the arrangement of I' inside the segments $l(v)$ and $r(v)$. Now we can determine the maximum depth of the arrangement of I' inside $l(v)$ as follows. Suppose the maximum is achieved at the point p somewhere in the segment $l(v)$. All intervals that cover $l(v)$ also cover p. Thus, if we only consider the intervals of I' that do not cover $l(v)$, then the maximum depth of the arrangement of this set intervals is also achieved at p. This depth is exactly the value $d_{l(v)}$. If we also include the intervals that do cover $l(v)$, i.e. I', we find that the maximum depth of the arrangement of I' inside the segment $l(v)$ is equal to $n_{l(v)} + d_{l(v)}$. An analogous reasoning holds for $r(v)$. Now it follows that $d_v = \max(d_{l(v)} + n_{l(v)}, d_{r(v)} + n_{r(v)})$. □

This lemma shows how we can compute the value d_v of an internal node v. Since the value n_v only changes if v is a node of the canonical subset $I(i)$ of an interval i, we only need to update the value d_v of a node v if its subtree contains a node of $I(i)$. We can 'bubble up' the information layer by layer starting from $I(i)$. We spend $O(1)$ time per node in the tree and since the canonical set of an interval i has size $O(\log n)$, the total time of an insertion equals $O(\log n)$.

The removal of an interval works in a similar fashion and also takes $O(\log n)$ time. Each square contributes only two operations, one insertion and one removal, so the total time of this algorithm is $O(n \log n)$.

One final step we have glossed over so far is how we determine the maximum depth in the arrangement of squares. We keep track of the maximum depth D in the area above the sweep line. Note that the removal can never increase the depth, so we only check if we need to update D after an insertion. We determine the depth of the total arrangement of the intervals that are currently in the tree. This is equal to the maximum depth of the arrangement of the set of squares that intersect the sweep line. We can find this value at d_ρ where ρ is the root of the tree. Because no interval covers the segment of ρ, we have that $n_\rho = 0$ and that d_ρ considers every interval that is currently in the tree. If $d_\rho > D$, we update the current value of D. After the sweep the value D contains the maximum depth of the arrangement of the squares.

5.2 Finding a data structure for Max-Pts-Square(r)

Now we will augment the above algorithm, so we can answer Max-Pts-Square(r) faster. Specifically, we alter the segment tree, so insertions and removals can be done slightly faster, albeit under some assumptions.

The first important change is that we use a B-tree instead of a binary tree. A *B-tree* of order f is a tree data structure where each non-leaf has f children [5]. We use the B-tree as a segment tree, which we call a *segment B-tree*. This means that each node still corresponds to a segment. The only difference with a regular segment tree is that each segment is now split in f smaller segments instead of just two. Just as in a regular segment tree, a node stores the segments in left-to-right order. If the number of elementary segments is not a power of f, we add segments on the right side, until it is. Each internal node then has exactly f children and the depth of the tree equals $O(\log_f(n))$. We define $C(v)$ to be the set of children of a node v. We augment a node v with the following data:

- N_v, an array of size f that stores for each child w the value n_w;
- D_v, an array of size f that stores for each child w the value d_w.

The value d_ρ of the root ρ (remember that n_ρ is always 0) is stored separately in the root, so the root has an additional value.

Lemma 5.2.1. *Suppose we have a segment tree with a set of intervals I inserted. For any internal node v, let d_v and n_v be the same as in Section 5.1. Let u be an arbitrary internal node of the segment tree. Then it holds that*

$$d_u = \max_{w \in C(u)} (N_u(w) + D_u(w)).$$

Proof. For each child w of u we have that $N_u(w) + D_u(w) = n_w + d_w$ by definition. Note that d_u is the maximum of $n_w + d_w$ over all children w of u following the same argument as the proof of Lemma 5.1.1. The lemma then follows directly. □

Now we have described how the data structure works. Before we consider the preprocessing step and how the queries work, we choose the order f of the B-tree. We choose $f = \log n$. In that case N_v and D_v become arrays of size $\log n$. The height of the tree is equal to $O(\log_f(n)) = O(\log n / \log\log(n))$.

We start with our first result which we then improve in later steps.

Theorem 5.2.2. *Let P be a point set in $\mathbb{R}^2$. We can preprocess these points in a data structure using $O(n)$ space in terms of number of words in $O(n \log n)$ time such that, for any real positive r, we can answer* Max-Pts-Square(r) *in $O(n \log^2(n)/\log\log(n))$ time under the following assumptions:*

- *we can increment and decrement any contiguous (part of an) array of at most f numbers by 1 in $O(1)$ time;*
- *we can determine the maximum over f numbers of $\log(n)$ bits in $O(1)$ time.*

Proof. We start with the preprocessing step. The first problem we solve is how the segment B-tree works if the intervals are not known beforehand. Note that the exact length of the intervals is in fact not relevant for the segment tree. If we know in which order the endpoints occur, then we can find the canonical subset of each interval. However, the order of the endpoints can also differ for different values of r. If r is extremely small, then the squares will not overlap and the resulting intervals will also be disjoint. However, for larger r this is not the case. But note that the structure of our B-tree is identical for any order of endpoints. Since the number of endpoints does not change, it is always a tree of the same depth where each internal node has f children. We only need to know the order of the endpoints when we are inserting the intervals in the tree. We can determine the order of the intervals in $O(n)$ time at the start of a query as follows. First, note that for a point $p = (p_x, p_y) \in P$ the corresponding interval equals $[p_x - r/2, p_x + r/2]$. The midpoint of this interval is always p_x independent of r. It follows that if we sort P by x-coordinate, we also get the order of the left sides of the corresponding intervals. The same holds for the right side of the intervals. This sorting step takes $O(n \log n)$ time and can be done in the preprocess step. Then, at the start of a query, we can determine the actual position of the endpoints and compute the order of all endpoints in $O(n)$ by using the merge step from the merge sort algorithm.

The next preprocess step is to determine the order of the insertions and removals. This situation is very similar to finding the order of the endpoints. The top of a square of a point $p = (p_x, p_y) \in P$ is at $y = p_y + r/2$ and its bottom is at $y = p_y - r/2$. So the order of the tops of the squares is independent of r. In the preprocess step we sort the points of P by y-coordinate (in a separate array of course). Again, at the start of a query we can determine the order of insertions and removals in $O(n)$ time using the merge step of the merge sort algorithm.

The final preprocess step is constructing the B-tree itself. We build the structure and the arrays N_v and D_v are initialised to 0 for each internal node v. Note that the size of the data structure is only $O(n)$ in terms of the number of words. However, each entry of arrays N_v and D_v can range from 0 to n, needing $\log(n) + 1$ bits. More importantly, although each internal node store $\log(n)$ values, for each child, in both N_v and D_v, per child only two values are stored. As each node only has one parent (except the root), in total the combined space usage of all N_v and D_v arrays is $O(n)$ in terms of the number of words.

Now we discuss the insertion and removal operations. We insert some interval i in the segment B-tree. Suppose we are in a node v of the B-tree. Let $v_1, \ldots, v_f$ be the children of v in left-to-right order. Now i covers some part of v, but not v completely. Otherwise, we would not visit v. Then v contains either the left or the right endpoint or both of them. We assume for now it only contains both the left and right endpoint, but the other two cases are analogous. We denote the two children that contain the left and right endpoints by v_l and v_r respectively. If they happen to fall on the border of two child nodes, then we choose the nodes that are contained in v.

The interval i contains all children in the (possibly empty) set $A = \{v_{l+1}, \ldots, v_{r-1}\}$. For each child $w \in A$ we want to increment the value $N_v(w)$ by 1, since i is an interval that covers w, but not its parent

v. Because A is a contiguous interval, we can, with our assumption, perform this increment in $O(1)$ time. Next we consider the children v_l and v_r. If i covers either v_l or v_r entirely, we increment $N_v(v_l)$ or $N_v(v_r)$ respectively. Otherwise, we traverse to v_l and v_r where we follow the same steps again. Eventually we arrive in the leaves from where we can start updating the D_v arrays. We can use Lemma 5.2.1 for this. Since the value $D_v(w)$ is equal to d_w, a node w can compute its own d_w and then update the corresponding value in the array of its parent. We can then, in the same way as in Section 5.1 bubble up the information to update the D_v arrays in the entire tree.

Let us consider the cost of an insertion. In the top-down traversal we spend $O(\log(f))$ time in a node, since we must perform a binary search to find which children contain the left and right boundary of i. The increment of part of the N_v array is then done in $O(1)$ time. In the bottom-up traversal, we must perform f additions to compute $N_v(w) + d_v(w)$ for each child w of v. This takes $O(f)$ time. Then we can take the maximum over all values in $O(1)$ time. In total we spend $O(f)$ time in each node and we traverse $O(\log_f(n))$ nodes. Thus, an insertion costs $O(f \cdot \log_f(n)) = O(\log^2(n)/\log\log(n))$ time.

The removal of an interval works in an analogous way and has the same cost. Instead of incrementing a part of the array, we now decrement it by 1. We perform n insertions and n removals. We already spent $O(n)$ time to find the order of the endpoints and the order of the insertions and removals. In total the runtime of a query becomes $O(n + n\log^2(n)/\log\log(n)) = O(n\log^2(n)/\log\log(n))$. □

The above result is slower than the result of Lemma 5.1.1. In the next theorem we improve the steps, so that we only spend $O(1)$ time per node.

Theorem 5.2.3. *Let P be a point set in $\mathbb{R}^2$. We can preprocess these points in a data structure using $O(n)$ space in terms of number of words in $O(n\log n)$ time such that, for any real positive r, we can answer* Max-Pts-Square(r) *in $O(n\log(n)/\log\log(n))$ time under the following assumptions:*

- *we can increment and decrement any contiguous (part of an) array of at most f numbers by 1 in $O(1)$ time;*
- *we can determine the maximum over f numbers of $\log(n)$ bits in $O(1)$ time.*

Proof. In the algorithm of Theorem 5.2.2, whenever we insert an interval i in our segment B-tree, there are two steps where we spend more than $O(1)$ time per node in the tree. The first is during our top-down traversal. We have to find which children contain the left and right endpoint of the interval which can be done with a binary search on the children. However, we can determine directly which child contains these endpoints. Suppose the left endpoint of the interval is the jth endpoint altogether. Since the subtree of every child of a node contains the same number of leaves, we can calculate directly which subtree contains the jth leave and therefore the jth endpoint. In the same way we can find the right endpoint.

The second step where we spend more than $O(1)$ time is when a node w calculates the new value d_w to update $D_v(w)$ of its parent v. We did not assume we can add two arrays pointwise together. However, we can avoid doing this addition altogether. In addition to N_v and D_v, we augment an internal node with an extra array E_v of size $\log n$ where $E_v(w) := N_v(w) + D_v(w)$ for every child w of v. Whenever we increment a contiguous part of the array of N_v, we also increment the same part of E_v. Similarly, whenever we update a value $D_v(w)$, we check whether $D_v(w)$ has increased by 1 (note that during one insertion this value can either be incremented with 1 or it remains the same). If this is the case we also increment $E_v(w)$. The node v can now update $D_u(v)$ (and $E_u(v)$) of its parent u by calculating the maximum value of the array E_v. By our assumptions, we can perform this calculation in $O(1)$ time. Note that the array N_v has now become redundant. The array D_v is not redundant, since it is not guaranteed that $D_v(w)$ (and $E_v(w)$) need to be incremented and we only know whether this happens by comparing it to the old value of $D_v(w)$.

During an insertion we now only spend $O(1)$ time per node, so the total cost of an insertion becomes $O(\log_f(n)) = O(\log n/\log\log n)$. The removal operation works again in the same way. With this increased runtime, the total query time becomes $O(n\log(n)/\log\log(n))$. □

5.3 Relieving the assumptions

Now it is time to address the assumptions we made and see if we can relieve them in some way. As a start we want to mention that we do not necessarily have to pick $f = \log(n)$. The same argument works for any

$f = O(n)$. For example, if we choose $f = \log\log(n)$, then our assumption is weaker, but the running time becomes larger. If we choose $f = O(1)$, we make no extra assumptions and we get the original running time of $O(\log(n))$ per insertion/deletion.

Next, we consider the problem Max-Pts-Square(r) under the assumption that k, the maximum number of points that can be contained by a square of size r, is bounded by some function $g(n)$. By making this extra assumption, we can remove the previous assumptions and design an algorithm entirely in the word RAM model.[3] Before we prove this statement, we need a small lemma.

Lemma 5.3.1. *For any positive real m and α such that $\alpha < 1$. We have that*

$$\log^m(n) \cdot 2^{\log^\alpha(n)} = o(n).$$

Proof. We will prove this by showing that

$$\lim_{n\to\infty} \frac{\log^m(n) \cdot 2^{\log^\alpha(n)}}{n} = 0.$$

We show that for any $k > 0$, there exists an $N \in \mathbb{N}$ such that for all $n > N$, it holds that

$$\frac{2^{\log^\alpha(n)}}{n} < k.$$

We start by noting that $\log(n) - \log^\alpha(n) - m\log\log(n)$ goes to infinity as $n \to \infty$. Therefore, there exists an $N \in \mathbb{N}$ such that $\log^\alpha(n) + \log\log^m(n) = \log^\alpha(n) + m\log\log(n) < \log(k) + \log(n) = \log(kn)$ for all $n > N$. Since the function $f(x) = 2^x$ is strictly monotone increasing, we have that $\log^m(n) \cdot 2^{\log^\alpha(n)} < kn$ for all $n > N$ from which it immediately follows that

$$\frac{\log^m(n) \cdot 2^{\log^\alpha(n)}}{n} < k.$$

□

Using this result we can now prove the following lemma.

Lemma 5.3.2. *Suppose we have a variable n and positive real constants c, ρ and ε with $\varepsilon + \rho < 1$. We can preprocess a data structure that uses $o(n)$ bits space that contains an ordered set A of $f := \log^\varepsilon(n)$ integers such that we can increase or decrease any contiguous part of A by 1 in $O(1)$ time and retrieve the maximum value at any time in $O(1)$ time under the assumption that the stored values are always less than $2^{c\log^\rho(n)}$.*

Proof. If the values of A are all less than $2^{c\log^\rho(n)}$, we can represent them with $w := c\log^\rho(n)$ bits. The set A can then be represented by a number containing $wf = c\log^{\rho+\varepsilon}$ bits. We call this number the *state* of A. The idea for the data structure is to construct a large table where each row corresponds with one of the possible states of A. The columns of the table correspond with the possible edits we can perform on A. This is either incrementing a contiguous part of A or decrementing it. As there are f values in A, there are $O(f^2)$ possible edits. The table then stores for each state and edit what the resulting state is, if we apply the edit to that state. Now if we want to increment or decrement a contiguous part of A, we can simply look up what the resulting state becomes in $O(1)$ time. To find the maximum of A, we add one extra column to our table that stores for each state the maximum value. Again, we can just look up this value in $O(1)$ time. A visualisation of the table can be seen in Table 3. For simplicity we left out the removal edit.

Next, we discuss how we can construct this table. Suppose we are considering a certain state E that consists of f numbers of w bits and we want to determine the state of some edit. Let us denote this edit by $\{i, j\}$ where we want to increase the range of values between the i-th and j-th index (inclusive). We can increment all values individually, starting from the ith one, but we observe that we then also determine the states for edits $\{i, i\}, \{i, i+1\}, \ldots, \{i, j-1\}$. So we can start with the edit $\{1, 1\}$. From there we determine $\{1, 2\}$, $\{1, 3\}$ etc. After that we start with $\{2, 2\}$ and continue to $\{2, f\}$. Each edit can be determined in $O(1)$ time from the previous edit. We can find the maximum of a state in simply $O(f)$ time by considering the

[3] This is only after the discretisation step which of course happens in the real RAM model.

f numbers of w bits	$\frac{f(f+1)}{2}$ edits				maximum
$0, 0, \ldots, 0$	$0, 0, \ldots, 0, 1$	$0, 0, \ldots, 1, 1$	$\ldots$	$1, 0, \ldots, 0, 0$	0
$0, 0, \ldots, 1$	$0, 0, \ldots, 0, 2$	$0, 0, \ldots, 1, 2$	$\ldots$	$1, 0, \ldots, 0, 1$	1
$0, 0, \ldots, 2$	$0, 0, \ldots, 0, 3$	$0, 0, \ldots, 1, 3$	$\ldots$	$1, 0, \ldots, 0, 2$	2
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
$2^w - 1, 2^w - 1, \ldots, 2^w - 1$	-	-	-	-	$2^w - 1$

(2^{fw} states)

Table 3: The large table that will store all possible state edits

f values. Since we also spend $O(f^2)$ time to construct the row for this state, it does not increase the complexity of the construction time. We conclude that the table can be built in linear time in the size of the table.

Each entry in the table consists of f numbers of w bits. This costs $wf = c\log^{\rho+\varepsilon}(n)$ bits. We have a row for each possible state. Since a state is represented by wf bits, we have at most $2^{wf} = 2^{c\log^{\rho+\varepsilon}(n)}$ rows. Finally the number of rows is $O(f^2) = O(\log^{2\varepsilon}(n))$. Because $\rho + \varepsilon < 1$, we can apply Lemma 5.3.1 and the total number of bits of the table then becomes

$$\begin{aligned} 2^{c\log^{\rho+\varepsilon}(n)} \cdot c\log^{\rho+\varepsilon}(n) \cdot O\left(\log^{2\varepsilon}(n)\right) &= 2^c 2^{\log^{\rho+\varepsilon}(n)} O\left(\log^{\rho+3\varepsilon}(n)\right) \\ &= o(n). \end{aligned}$$

□

Using this lemma, we can now show the following theorem on the Max-Pts-Square(r) problem.

Theorem 5.3.3. *Suppose we have some point set $P \in \mathbb{R}^2$. Then we can construct a data structure in $O(n \log n)$ time using $O(n)$ space in terms of the number of words such that, given a real value r, we can answer* Max-Pts-Square(r) *in $O(n\log(n)/\log\log(n))$ time under the assumption that the maximum number of points enclosed by a square of size r is at most $2^{c\log^{\rho}(n)}$ where c and ρ are positive constants with $\rho < 1$.*

Proof. Let ε be a constant such that $\varepsilon < 1-\rho$. We start by choosing $f = \log^{\varepsilon}(n)$ as the order of our segment B-tree. Each internal node then has $\log^{\varepsilon}(n)$ children. The height of the tree becomes $O(\log(n)/\log(\log^{\varepsilon}(n))) = O(\log(n)/\log\log(n))$ which is equal to the height of the tree of Theorem 5.2.3. Under our assumption, for any node v no entry of E_v becomes more than k. Thus, we can use Lemma 5.3.2. It follows that we can increment any contiguous portion of E_v and calculate the maximum in $O(1)$ time. Note that we only need to preprocess the large table once, since we can use it for every node in the B-tree. Constructing this data structure is done faster than the $O(n \log n)$ sorting step, so the total preprocessing time remains the same. Moreover, the table uses $o(n)$ number of bits to store, so it definitely uses less than $O(n)$ in terms of number of words. The state of a node uses less than $\log(n)$ bits as well which fits in one word. The total memory cost of the data structure then becomes $O(n)$ in terms of the numbers of words stored. □

5.4 The result applied to other problems

We can use Theorem 5.3.3 not only for the Max-Pts-Square(r) problem. Unfortunately, we cannot apply it to the Min-Square(k) problem. Recall that the k-smallest square always had either a point of P on both its top and bottom side or on its left and right side. It followed that we could reduce the possible sizes of the k-smallest intervals to a set of size $O(n^2)$. In Section 3.1 we discussed an algorithm by Eppstein and Erickson [17, Lemma 5.7] where the direct algorithm of the Max-Pts-Square(r) was invoked $O(\log n)$ times by applying a binary search on all these $O(n^2)$ possible sizes for the square size. Determining the next size required $O(n \log n)$ time using a search technique by Frederickson and Johnson [19]. This $O(n \log n)$ overshadows the $O(n\log(n)/\log\log(n))$ query time we have in 5.3.3.

Fortunately, we can apply Theorem 5.3.3 to the Max-Pts-Rect(l, w) problem.

Theorem 5.4.1. *Suppose we have some point set $P \in \mathbb{R}^2$. Then we can construct a data structure in $O(n \log n)$ time using $O(n)$ space in terms of the number of words such that, given real values l and w, we can answer* Max-Pts-Rect(l, w) *in $O(n \log(n)/\log\log(n))$ time under the assumption that the maximum number of points enclosed by a rectangle of size l by w is at most $2^{c \log^{\rho}(n)}$ where c and ρ are positive constants with $\rho < 1$.*

Proof. This problem is very similar to the Max-Pts-Square(r) problem. We use the same sweep line algorithm where each rectangle is mapped to an interval on the sweep line. Note that all insertions events are still in the same order as the order of P, sorted by y-coordinate and the same holds for the removal events. Thus, we can determine the order of all events in $O(n)$ time by merging the insertions and the removals. In a similar way, we can determine the order of the endpoints of the intervals. The rest of the algorithm is the exact same. □

6 A data structure for the circle

In this section we consider the Min-Circle(k) and the Max-Pts-Circle(r) problems and describe a data structure for both of them. We will solve these problems using the lifting transformation of Section 2.6. Suppose we have a point set P in $\mathbb{R}^2$. We assume P to be in general position, meaning there are no four co-circular points. We want to find a smallest circle containing k points which we denote by C_k. The following lemma reduces the number of candidate circles for C_k significantly.

Lemma 6.0.1. *Suppose P is a point set in $\mathbb{R}^2$ in general position. For any value of k, then for the smallest circle containing k points, C_k, there are two possible cases:*

1. *the boundary of C_k contains three points of P;*
2. *the boundary of C_k contains two points that form a diagonal of C_k.*

Proof. Suppose we have some value k for which C_k has neither of the two stated properties. By the general position assumption we know that C_k has at most one point of P on its border. Then we can shrink it by a small amount while still containing the same k points. So C_k is not the smallest circle containing k points and by contradiction the lemma holds. An example where $k = 8$ is shown in Figure 15. □

Suppose we have a circle C that has three points p, q and r of P on its boundary. We apply the lifting transformation to these circle and points. As a short reminder, a point p in P becomes a plane h_p in $\mathbb{R}^3$ in the projective space and a circle C becomes a point p_C in $\mathbb{R}^3$. Then p lies inside C if and only if h_p lies above p_C. In the second dual space these become planes h_p, h_q and h_r respectively. By Lemma 2.6.3 we have that the transformation of C, p_C, must lie on each of these three planes in the projected space. It follows that p_C coincides with the unique intersection point $s = (s_x, s_y, s_z)$ of the three planes. Note that s is a vertex in the arrangement of $H(P)$. Since each circle corresponds to a vertex in the arrangement in the projected space, we can consider each possible circle that has three points of P on its border by visiting all vertices in the arrangement of $H(P)$. For each vertex s we can determine the radius of the corresponding circle by reversing the transformation of a circle. If $s = (s_x, s_y, s_z)$, then the radius equals $\sqrt{s_x^2 + s_y^2 - s_z}$.

We will not use the lifting transformation for the circles whose boundary contains two points on a diagonal. For completion we do explain their transformation and to which points in the projected space these circles are mapped. Suppose we have a circle C with two points p and q on a diagonal of its boundary. Then the centre of this circle is the midpoint of the line segment $\overline{pq}$. The transformation p_C lies somewhere on the line that is the intersection of h_p and h_q. Note that if we map this line on the $z = 0$ plane, we get the bisector of p and q, since each point on the bisector is equidistant to p and q. Each of these points defines a circle that goes through p and q, but we are looking for the one where p and q lie on a diagonal. That is the circle with the smallest radius. This means that we are looking for the point on the line that is the intersection of h_p and h_q that has the smallest vertical distance to the unit paraboloid.

Since we only use the lifting transformation for circles that have three points of P on its boundary, we give this problem a specific name. We define Min-SubCircle(k) as the problem of finding the smallest circle

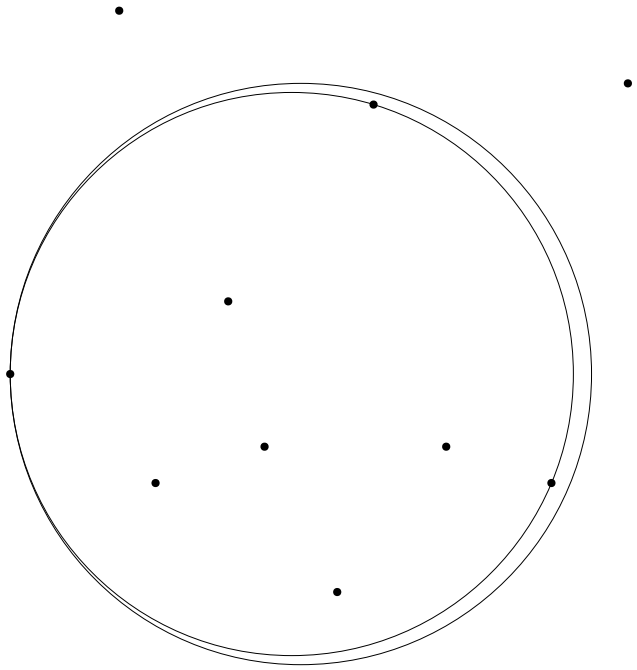


Figure 15: We shrink C_8, until its border contains three points.

that contains $k \geq 3$ points of P of which three lie on its border. We observed that a circle C that is defined by three points of P actually corresponds to a point p_C in $\mathbb{R}^3$ that is the intersection of three planes of $H(P)$. Given a circle C, we can compute how many points lie in the interior of C by counting the number of planes lying strictly above s. In the literature, this is often called the *level* of a point in an arrangement [3]. Now we see that the problem Min-SubCircle(k) is equivalent to finding a vertex of level $k-3$ in the arrangement of planes in the projected space that minimises some function $f : \mathbb{R}^3 \to \mathbb{R}$. The function f, in this case, computes the radius of the corresponding circle and we can define it as $f(x, y, z) = \sqrt{x^2 + y^2 - z}$. We search for a vertex of level $k-3$, since the level of a point excludes planes that go through the point. Note that we can look for exact levels and not for levels at least $k-3$, since we assumed the points are in general position. If C is the k-smallest subcircle, we know that the j-smallest subcircle, with $j < k$, has radius strict smaller than C, since we can shrink C by a small amount to exclude exactly one point. Of course this problem uses a specific function of f, but there might be other problems that use a different function. As we will present an algorithm that does not use the specific f, we can formulate this as a generalised version of the problem.

Definition 6.0.2. Let H be a set of n hyperplanes in $\mathbb{R}^3$. Let also $f : \mathbb{R}^3 \to \mathbb{R}$ be a function and let j be an integer. Then we define Min-Level(j, f) to be the problem of finding the vertex q of H of level j that minimises f. We use the term *j-smallest level (of f)* to refer to the solution of this problem.

Lemma 6.0.3. *The* Min-SubCircle(k) *problem is a special case of the* Min-Level(j, f) *problem.*

Proof. We transform the set of points P to $H(P)$. Then we choose $f(x, y, z) = \sqrt{x^2 + y^2 - z}$ and $j = k - 3$. □

6.1 Finding all j-smallest levels

Before we describe a data structure for the Min-Circle(k) problem, we consider the problem of computing all j-smallest levels for a given function $f : \mathbb{R}^3 \to \mathbb{R}$. That is, we are given a set H of planes in $\mathbb{R}^3$ and a function $f : \mathbb{R}^3 \to \mathbb{R}$. Then for each value j we find a vertex in the arrangement of H of level j such that f is minimised. We will discuss four algorithms in total. We start with the following algorithm presented in Algorithm 1. We just fully construct the entire arrangement of planes. Then we consider the graph consisting of just the vertices and edges of the arrangement. So two vertices v and w are connected if there are two

Algorithm 1: NaiveAlgorithm(H, f)

Input: A set of n planes H in $\mathbb{R}^3$ and a function $f : \mathbb{R}^3 \to \mathbb{R}$
Result: The solutions of Min-Level(j, f) for $1 \leq j \leq n$ in an array A
1. Construct the arrangement $\mathcal{A}(H)$ of the planes.
2. Construct the graph G that consists of the vertices and edges of $\mathcal{A}(H)$.
3. Select random vertex v of G and determine its level j:
   3a. Determine $f(v)$ and update $A[j]$.
   3b. Select random unvisited neighbour w of v.
   3c. Compute level of w from j.
   3d. Set $v := w$.
3. Return A.

planes going through both and no other plane intersects the line segment $\overline{vw}$. We perform a *breadth-first search* (BFS) on this graph. During the BFS we store a reference to the j-smallest level for each value of j. We can start from any vertex v for which we determine its level in the arrangement. We just iterate over all planes and count how many lie above v. When we go to a neighbouring vertex w, we can compute its level from the level of v using the following lemma.

Lemma 6.1.1. *Given an arrangement of a set H of planes in $\mathbb{R}^3$. We assume general position, so in particular no four planes share a common point. Then for any two vertices v and w that are adjacent in the arrangement, we can determine the level of w, given the level of v, in $O(1)$ time.*

Proof. We start by noting that if v and w are adjacent in the arrangement, then there are two planes, h_1 and h_2 of H that contain both v and w. Then there are two more planes, h_v and h_w, such that v is the intersection point of h_1, h_2 and h_v and w is the intersection point of h_1, h_2 and h_w. All other planes either pass below or above $\overline{vw}$. If this is not the case, then some plane h must intersect $\overline{vw}$. It cannot go through v and w by the general position assumption. Suppose h intersects the interior of $\overline{vw}$. Then the planes h, h_1 and h_2 intersect each other in a point on $\overline{vw}$, so v and w are not adjacent.

Let l_v be the level of v. We start by setting the level l_w of w to l_v. The only difference between the level of v and the level of w can be caused by h_v or h_w, since all other planes pass above, below or through both v and w. If h_v passes above w, then there is an additional plane passing above w and we increment l_w by 1. Similarly, if h_w passes above v, then we decrement h_w by 1. Now we have the level of w. □

For each vertex v we calculate $f(v)$ (slight abuse of notation) and its level j and check whether $f(v)$ is smaller than the one we had stored level j. We will call this algorithm NaiveAlgorithm. Since the arrangement has complexity $O(n^3)$, we need both $O(n^3)$ space and time.

We want to avoid using cubic space. An intuitive approach is to generate all $O(n^3)$ intersections in sequence and query how many planes lie above it. This is the Vertical Ray Stabbing problem. This approach can more easily be solved by considering the dual problem we described in Section 2.5. In $\mathbb{R}^3$ each plane h is transformed to a point h^* and a point p is transformed to a plane p^*. By applying this transformation we end up with the following problem. Given a set of points $P \in \mathbb{R}^3$ and a plane h, how many points of P lie above h. We already saw in Section 2 that this is called the Halfspace Range counting problem. If we can solve this problem, we can also count the number of planes that lie above a certain point. If we want to avoid large space usage, then we can use the following result by Matoušek [25]. He constructs a data structure of size $O(n)$ in $O(n^{1+\delta})$ where δ is some arbitrary small positive constant. A query then costs $O(n^{2/3})$ time. This algorithm is actually used to solve the Simplex Range Counting problem, but with Lemma 2.3.2 this also solves the Halfspace Range counting problem. Unfortunately, if we query each of the $O(n^3)$ vertices of the arrangement, we get a total running time of $O(n^{11/3})$ algorithm.

Finally, we show an algorithm that computes all j-smallest levels in $O(n^3)$ time using $O(n)$ space. Before we consider this algorithm however, we revisit Chazelle's algorithm for the $(1/r)$-cutting. Although the cutting itself only uses $O(r^d)$ space, the algorithm that computes it uses $O(nr^{d-1})$ workspace. Because our aim is to achieve linear space usage, we must alter the algorithm, so it uses less space. Choosing $r = O(1)$

does not work as it does not reduce the number of hyperplanes that intersect a simplex enough. At the cost of some time we reduce the space usage in the following lemma.

Lemma 6.1.2. *Given a set H of n hyperplanes in $\mathbb{R}^d$. We can find a $(1/r)$-cutting of size $O(r^d)$ in $O(nr^d)$ time using $O(n + r^d)$ space.*

Proof. Recall that Chazelle's algorithm computes a $(1/r)$-cutting by iteratively splitting the simplices in smaller ones, until each simplex is only intersected by n/r hyperplanes. The intermediate cuttings were labeled C_k. In order to construct C_{k+1} from C_k, Chazelle's algorithm explicitly stored the conflict lists $H_{|s}$ for all simplices s in C_k. In the final step this resulted in $O((n/r) \cdot r^d) = O(nr^{d-1})$ space. Given the conflict list $H_{|s}$ of some simplex s, the algorithm splits s for the next cutting in $O(|H_{|s}|)$ time. It follows that we need at most $O(|H_{|s}|)$ additional space for this step. Now instead of storing the sets $H_{|s}$ explicitly, we only determine $H_{|s}$ when we are splitting simplex s. We can achieve this simply by iterating over all hyperplanes. This increases the time used per simplex from $O(|H_{|s}|) = O(n/r_0^k)$ to $O(n)$. Then the runtime of the algorithm becomes

$$\begin{aligned}\sum_{0\le k\le \lceil \log_{r_0} r\rceil} O(n)|C_k| &\le \sum_{0\le k\le \lceil \log_{r_0} r\rceil} O\left(nr_0^{kd+d}\right)\\ &= O\left(n(r_0^d)^{\lceil \log_{r_0} r\rceil}\right)\\ &= O(nr^d)\end{aligned}$$

The space usage now becomes $O(n + r^d)$, since we only need to store $H_{|s}$ whenever we split simplex s. After this is done, we can reuse this space for the next simplex. The cutting itself still has complexity $O(r^d)$. □

With this lemma we can greatly improve the space usage for the algorithm to compute all j-smallest levels.

Theorem 6.1.3. *Given a set H of n planes in $\mathbb{R}^3$ and some function $f : \mathbb{R}^3 \to \mathbb{R}$, we can calculate all j-smallest levels of f in $O(n^3)$ time using $O(n^{3/2})$ space.*

Proof. The algorithm we describe is presented in Algorithm 2. For each value of j we store a reference to the j-smallest level. We then start by computing a $(1/r)$-cutting of H as described in Lemma 6.1.2. Before we choose r, we describe first our algorithm. For each simplex s of the cutting we perform two steps. First, we count the number t of planes that lie entirely above s and simultaneously we determine the conflict list $H_{|s}$. This can be done simply in $O(n)$ time by iterating over the planes. Note that the number of planes that pass above a vertex v in s equals the number of planes passing above s plus the number of planes of $H_{|s}$ passing above v. The second step is to execute the NaiveAlgorithm on $H_{|s}$. This gives us the j'-smallest levels on $H_{|s}$ with $0 \le j' \le |H_{|s}|$. This allows us to update the reference to the j-smallest levels for $t \le j \le t + |H_{|s}|$. The NaiveAlgorithm costs $O(|H_{|s}|^3) = O((n/r)^3)$ time and also uses $O((n/r)^3)$ space. If we perform these steps for every simplex in the cutting, we visit all vertices of our arrangement at least once. Note that it may happen that we see some vertices more than once, since it is possible that a certain set of three hyperplanes intersects more than one simplex. Then their intersection point occurs in the arrangement for multiple simplices.

Let us analyse the running time and storage. Constructing the cutting can be done in $O(nr^3)$ time and $O(n + r^3)$ space using Lemma 6.1.2. Per cell we spend $O(n + (n/r)^3)$ time and space and there are $O(r^3)$ cells. In total we have an algorithm that runs in $O\left(nr^3 + r^3 \cdot (n + (n/r)^3)\right) = O(n^3)$ time and uses $O(n + r^3 + (n/r)^3)$ space. If we choose $r = \sqrt{n}$, then it becomes an $O(n^{3/2})$ space algorithm.

□

The $O(n^{3/2})$ space is already an improvement, but we can improve further. The NaiveAlgorithm, that we use as subroutine, requires cubic space. Instead, we can use CuttingAlgorithm itself as a subroutine. Since this algorithm uses less space, the total space usage also becomes smaller. If we choose $r = n^{1/3}$, the space usage then becomes $O(n + r^3 + (n/r)^{3/2}) = O\left(n + \left(n^{1/3}\right)^3 + \left(n/n^{1/3}\right)^{3/2}\right) = O(n)$. We have come to the following theorem.

Algorithm 2: CuttingAlgorithm(H, f)

Input: A set of n planes H in $\mathbb{R}^3$ and a function $f : \mathbb{R}^3 \to \mathbb{R}$
Result: The solutions of Min-Level(j, f) for $1 \leq j \leq n$ in an array A
1. Construct a $(1/\sqrt{n})$-cutting on H.
2. For each simplex s:
   2a. Count the number of planes t that lie entirely above s.
   2b. Find the set of planes R that intersect s.
   2c. Set $A' =$ NaiveAlgorithm(R, f).
   2d. Update A with A' and t.
3. Return A.

Theorem 6.1.4. *Given a set H of n planes in $\mathbb{R}^3$ and some function $f : \mathbb{R}^3 \to \mathbb{R}$, we can calculate all j-smallest levels of f in $O(n^3)$ time using $O(n)$ space.*

The corresponding algorithm is presented in Algorithm 3.

Algorithm 3: ImprovedCuttingAlgorithm(H, f)

Input: A set of n planes H in $\mathbb{R}^3$ and a function $f : \mathbb{R}^3 \to \mathbb{R}$
Result: The solutions of Min-Level(j, f) for $1 \leq j \leq n$ in an array A
1. Construct a $(1/n^{1/3})$-cutting on H.
2. For each simplex s:
   2a. Count the number of planes t that lie entirely above s.
   2b. Find the set of planes R that intersect s.
   2c. Set $A' =$ CuttingAlgorithm(R, f).
   2d. Update A with A' and t.
3. Return A.

6.2 Finding all k-smallest circles

Finally, we show how we can use these results to come to an efficient data structure for the Min-Circle(k). We want this data structure to be able to report which point lies in the k-smallest circle whenever we query the value k. Ideally, we use linear space and use $O(k)$ time to report the points in the k-smallest circle. This is in fact possible using only linear workspace during the construction of the data structure.

Theorem 6.2.1. *Given a set P of n points in $\mathbb{R}^2$. We can construct a linear space data structure in $O(n^3)$ time and $O(n)$ space that, given an integer k, answers the* Min-Circle(k) *problem in $O(1)$ time and reports the corresponding k points in $O(k)$ time.*

Proof. We start by computing all k-smallest subcircles using Theorem 6.1.4. To compute the k-smallest circles we only need to check the circles where two points of P lie on the diagonal. Per circle we simply count the number of points, say k, it contains. If we keep track per value of k of the smallest circle, we can determine the k-smallest circle by combining the results with the k-smallest subcircle. This step takes $O(n^3)$ time as there are $O(n^2)$ circles we need to check and we spend $O(n)$ time per circle. We also need only linear space.

So we can determine the k-smallest circles in $O(n^3)$ time and $O(n)$ space. The only step left is reporting the k corresponding points in $O(k)$ time. We start by noting that we cannot store these points explicitly, since that uses $O(n^2)$ space. However, we can store the points that lie in the k-smallest circles for $1 \leq k \leq \sqrt{n}$, as this uses $O(n)$ space. Furthermore, we discuss a result by Afshani and Chan [1]. They consider the variant of the Halfspace Range Reporting problem where we also report the points. In their paper they build an $O(n)$ space data structure on a point set P in $O(n \log n)$ time. Using this data structure, they can query planes and report the points that lie above the plane in $O(\log n + k)$ time. After we apply the transformation described in Section 2.6 to the set P and the k-smallest circles, we can query the k-smallest

circles for $1 \leq k \leq \sqrt{n}$ and store the reported points explicitly. We also store the data structure of Afshani and Chan. Now we can report the k points in the k-smallest circle as follows. If $k \leq \sqrt{n}$, we have stored the points explicitly, so we can report them in $O(k)$ time. Otherwise, we query the data structure of Afshani and Chan using $O(\log n + k) = O(k)$ time. □

We get a similar result for MAX-PTS-CIRCLE(r) problem.

Corollary 6.2.2. *Given a set P of n points in $\mathbb{R}^2$. We can construct a linear space data structure in $O(n^3)$ time and $O(n)$ space that, given a positive real number r, answers the* MAX-PTS-CIRCLE(r) *problem in $O(\log n)$ time and reports the corresponding points in $O(k)$ time where k is the maximum number of points contained by a circle with diameter r.*

Proof. We build the same data structure as in Theorem 6.2.1. A query then consists of a binary search to find the solution k and report the k smallest points in the exact same way as in Theorem 6.2.1. □

6.3 Extra result on Chazelle's algorithm

In Lemma 6.1.2 we showed that we can reduce the space usage of Chazelle's algorithm to $O(n + r^d)$ instead of the $O(nr^{d-1})$ at the cost of some extra computation time. We can actually achieve a better result than what we showed there. The extra cost originated in finding the conflict lists $H_{|s}$ every time we needed to split a simplex s. This takes $O(n)$ time instead of the $O(|H_{|s}|)$ time we need to split the simplex. However, we can find the relevant conflict list faster than in $O(n)$ time.

Lemma 6.3.1. *Let H be a set of n $(d-1)$-dimensional hyperplanes in $\mathbb{R}^d$ and $\delta > 0$. We can preprocess these hyperplanes in a linear space data structure using $O(n^{1+\delta})$ time and space such that we can report the set $H_{|s}$ of hyperplanes that intersect some d-dimensional simplex s in $O(n^{1-1/d} + k)$ time where $k = |H_{|s}|$.*

Proof. Before we show this general lemma, we prove it for the case where $d = 2$. The H set of hyperplanes, which is now a set of lines, we will call L. The simplex s becomes then a triangle. Let s_0, s_1, s_2 be the vertices of s. We apply the standard duality transformation to the lines of L that we described in Section 2.5. Then each line ℓ becomes a point $\ell^* \in \mathbb{R}^2$ and we define $L^* := \{\ell^* \mid \ell \in L\}$. We also transform s_0, s_1 and s_2 into the lines s_0^*, s_1^* and s_2^* respectively. Now, if a line ℓ does not cross the triangle, then it either must pass below all its vertices or over them. It follows that ℓ^* then lies above s_0^*, s_1^* and s_2^* or below them all. Let us call these regions A (above) and B (below) respectively. It follows that we must find the points of L^* that lie outside A and B. As A and B are defined by a constant number of lines, their complexity is constant as well. So, we can subdivide $\mathbb{R}^2 \setminus (A \cup B)$ in a finite number of triangles. Now we have reduced the problem to a finite number of instances of the SIMPLEX RANGE REPORTING problem.

In d dimensions the same logic applies. The simplex s is now defined by d points, so the complexity of the regions A and B will only depend on d. Again A and B have constant complexity. Now all us rests is to find an appropriate algorithm that solves the SIMPLEX RANGE REPORTING problem. For this we can use a result by Matoušek [25]. He constructs a linear space data structure that can answer simplex range queries in $O(n^{1-1/d})$ time. The preprocess time is $O(n^{1+\delta})$ where δ is some small positive constant. The algorithm also uses $O(n^{1+\delta})$ space during the construction of the data structure. □

We use the above algorithm to get the following result on the $(1/r)$-cutting.

Corollary 6.3.2. *Given a set H of n hyperplanes in $\mathbb{R}^d$ and $\delta > 0$. We can find a $(1/r)$-cutting of size $O(r^d)$ in $O\left(n^{1-1/d}r^d + nr^{d-1}\right)$ using $O(n^{1+\delta} + r^d)$ space.*

Proof. We combine Lemmas 6.1.2 and 6.3.1. Recall that in Chazelle's algorithm in iteration k we had $|H_{|s}| \leq n/r_0^k$. Using Lemma 6.3.1 we can find $H_{|s}$ in $O\left(n^{1-1/d} + n/r_0^k\right)$ time. The total running time then becomes

$$\begin{aligned}\sum_{0 \leq k \leq \lceil \log_{r_0} r \rceil} O\left(n^{1-1/d} + n/r_0^k\right)|C_k| &\leq \sum_{0 \leq k \leq \lceil \log_{r_0} r \rceil} O\left(n^{1-1/d} r_0^{kd+d} + n r_0^{k(d-1)+d}\right) \\ &= O\left(n^{1-1/d}(r_0^d)^{\lceil \log_{r_0} r \rceil} + n(r_0^{d-1})^{\lceil \log_{r_0} r \rceil}\right) \\ &= O\left(n^{1-1/d} r^d + nr^{d-1}\right)\end{aligned}$$

We use $O(n^{1+\delta})$ space from Lemma 6.3.1 and $O(r^d)$ to store the total cutting. □

7 Future research and ideas

Finally, we want to discuss some open questions and ideas that can further improve the results that we have discussed in this thesis. An important open problem is whether we can, given a set of point in $\mathbb{R}^1$, find the k-smallest intervals for $1 \leq k \leq n$ in subquadratic time.

In Section 5.4 we discussed whether we could use the results of Section 5 to Min-Square(k). We observed that we could not speed up the binary search of the $O(n^2)$ possible squares. Although the call to the algorithm for Max-Pts-Circle(r) has a speedup, determining the next value of r in the binary search costs $O(n \log n)$. There might be a different way for finding this next value of r that does not require $O(n \log n)$ time. In that case we also find results for the Min-Square(k) problem.

In Section 6 our idea basically comes down to iterating over each circle that has exactly three points of a set P on its boundary in a particular order. Although we achieved minimal space usage, in some situations a cubic algorithm might not be feasible. A different approach is then to look for a data structure that can be built in less time at the cost of a higher query time. Using Lemma 3.1.1 from Chan we could take the following approach. Instead of precalculating all solutions for all values of k, we visit the simplices of the $(1/r)$-cutting during the query of a specific value k. Chan's lemma uses the fact that checking whether the solution of a subproblem is smaller than a specific value is easier than computing the solution itself. In our case we can check whether a simplex s of the $(1/r)$-cutting contains a vertex of the arrangement of hyperplanes such that the corresponding circle contains k points and is smaller than a value t. We denote this problem by Dec(s, k, t). If we can solve this problem faster than the $O((n/r)^3)$ time that we spend in our algorithm, we save time, since we only need to calculate the true solution for a small number of simplices. In Chan's lemma we observed that the expected number of subproblems that we have to solve completely is $O(\log m)$ where m is the number of subproblems in total. In our case this is only $O(r^3)$, so we expect to only need to compute $O(\log r)$ subproblems explicitly. Of course, to make a fair comparison, we need to compare the query time to the direct algorithms for the Min-Circle(k) problem. In Section 3 we saw that all known results achieve a $O(n^2 \cdot \text{polylog}(n))$ for $k = \Theta(n)$. Now suppose we have an oracle $\mathcal{A}$ that can solve Dec(s, k, t) in $f(n)$ where n is the number of planes that intersect s. We then spend $f(n/r)$ time per simplex. On top of that, for $O(\log r)$ simplices we must spend more time. The total running time then becomes $O((n/r)^3 \log(r) + f(n/r)r^3)$. Whether such an algorithm has a query time that is faster than a direct algorithm depends on the running time of the oracle. A future study could try to determine how we can preprocess the cutting further to get a sufficient fast enough query time.

A different continuation is to find what other problems we could solve using Theorem 6.1.4. We considered a specific choice of the function f which reduced the problem to finding the k-smallest circles. One possible generalisation is that of semialgebraic sets, which are regions that are bounded by a constant number of bounded degree polynomials [2]. For example, algorithms for $(1/r)$-cuttings have been generalised to these semialgebraic sets under some conditions [2, Lemma 3.1]. In our case we considered the special case where our semialgebraic sets where halfspaces, bounded by one hyperplane. Combined with the fact that we can freely choose our function f, we might be able to achieve similar results for shapes different from circles.

Finally, in Section 6.3 we improved our result on Chazelle's algorithm while remaining small space usage. However, in Lemma 6.1.2 we achieved a space usage of $O(n + r^d)$ which is truly linear if $r = O(n^{1/d})$. In Corollary 6.3.2 we achieved a faster running time, but the space usage became $O(n^{1+\delta} + r^d)$. The question remains how fast we can compute an $(1/r)$-cutting while retaining minimal space.

In general, we could try to achieve better results by making assumptions on the input. In Section 5 we already made an assumption on the maximum number of points a square could enclose. However, a different common assumption is that on the point set P. For example, we can assume a bounded spread. This means that points cannot lie arbitrarily close or far apart from each other. In Section 3.2 we discussed two paper that made this assumption.

References

[1] Peyman Afshani and Timothy M. Chan. Optimal halfspace range reporting in three dimensions. In Claire Mathieu, editor, *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2009, New York, NY, USA, January 4-6, 2009*, pages 180–186. SIAM, 2009. URL http://dl.acm.org/citation.cfm?id=1496770.1496791.

[2] Pankaj K. Agarwal and Jirí Matousek. On range searching with semialgebraic sets. *Discret. Comput. Geom.*, 11:393–418, 1994. doi: 10.1007/BF02574015. URL https://doi.org/10.1007/BF02574015.

[3] Pankaj K. Agarwal, Mark de Berg, Jirí Matousek, and Otfried Schwarzkopf. Constructing levels in arrangements and higher order voronoi diagrams. *SIAM J. Comput.*, 27(3):654–667, 1998. doi: 10.1137/S0097539795281840. URL https://doi.org/10.1137/S0097539795281840.

[4] Alok Aggarwal, Hiroshi Imai, Naoki Katoh, and Subhash Suri. Finding k points with minimum diameter and related problems. *J. Algorithms*, 12(1):38–56, 1991. doi: 10.1016/0196-6774(91)90022-Q. URL https://doi.org/10.1016/0196-6774(91)90022-Q.

[5] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972. doi: 10.1007/BF00288683. URL https://doi.org/10.1007/BF00288683.

[6] Michael Ben-Or. Lower bounds for algebraic computation trees (preliminary report). In David S. Johnson, Ronald Fagin, Michael L. Fredman, David Harel, Richard M. Karp, Nancy A. Lynch, Christos H. Papadimitriou, Ronald L. Rivest, Walter L. Ruzzo, and Joel I. Seiferas, editors, *Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 25-27 April, 1983, Boston, Massachusetts, USA*, pages 80–86. ACM, 1983. doi: 10.1145/800061.808735. URL https://doi.org/10.1145/800061.808735.

[7] Jon Louis Bentley and Michael Ian Shamos. Divide-and-conquer in multidimensional space. In Ashok K. Chandra, Detlef Wotschke, Emily P. Friedman, and Michael A. Harrison, editors, *Proceedings of the 8th Annual ACM Symposium on Theory of Computing, May 3-5, 1976, Hershey, Pennsylvania, USA*, pages 220–230. ACM, 1976. doi: 10.1145/800113.803652. URL https://doi.org/10.1145/800113.803652.

[8] Jan Brandts, Sergey Korotov, Michal Křížek, et al. *Simplicial Partitions with Applications to the Finite Element Method*. Springer, 2020.

[9] Timothy M. Chan. Geometric applications of a randomized optimization technique. *Discret. Comput. Geom.*, 22(4):547–567, 1999. doi: 10.1007/PL00009478. URL https://doi.org/10.1007/PL00009478.

[10] Timothy M. Chan and Sariel Har-Peled. Smallest k-enclosing rectangle revisited. *CoRR*, abs/1903.06785, 2019. URL http://arxiv.org/abs/1903.06785.

[11] Bernard Chazelle. Cutting hyperplanes for divide-and-conquer. *Discret. Comput. Geom.*, 9:145–158, 1993. doi: 10.1007/BF02189314. URL https://doi.org/10.1007/BF02189314.

[12] Bernard Chazelle and Joel Friedman. A deterministic view of random sampling and its use in geometry. *Comb.*, 10(3):229–249, 1990. doi: 10.1007/BF02122778. URL https://doi.org/10.1007/BF02122778.

[13] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. ISBN 978-0-262-03384-8. URL http://mitpress.mit.edu/books/introduction-algorithms.

[14] Mark de Berg, Otfried Cheong, Marc J. van Kreveld, and Mark H. Overmars. *Computational geometry: algorithms and applications, 3rd Edition*. Springer, 2008. ISBN 9783540779735. URL https://www.worldcat.org/oclc/227584184.

[15] Mark de Berg, Sergio Cabello, Otfried Cheong, David Eppstein, and Christian Knauer. Covering many points with a small-area box. *J. Comput. Geom.*, 10(1):207–222, 2019. doi: 10.20382/jocg.v10i1a8. URL https://doi.org/10.20382/jocg.v10i1a8.

[16] Alon Efrat, Micha Sharir, and Alon Ziv. Computing the smallest k-enclosing circle and related problems. *Comput. Geom.*, 4:119–136, 1994. doi: 10.1016/0925-7721(94)90003-5. URL https://doi.org/10.1016/0925-7721(94)90003-5.

[17] David Eppstein and Jeff Erickson. Iterated nearest neighbors and finding minimal polytopes. *Discret. Comput. Geom.*, 11:321–350, 1994. doi: 10.1007/BF02574012. URL https://doi.org/10.1007/BF02574012.

[18] Steven Fortune and John E. Hopcroft. A note on rabin's nearest-neighbor algorithm. *Inf. Process. Lett.*, 8 (1):20–23, 1979. doi: 10.1016/0020-0190(79)90085-1. URL https://doi.org/10.1016/0020-0190(79)90085-1.

[19] Greg N. Frederickson and Donald B. Johnson. The complexity of selection and ranking in X+Y and matrices with sorted columns. *J. Comput. Syst. Sci.*, 24(2):197–208, 1982. doi: 10.1016/0022-0000(82)90048-4. URL https://doi.org/10.1016/0022-0000(82)90048-4.

[20] Raymond Greenlaw and H James Hoover. *Fundamentals of the Theory of Computation: Principles and Practice*. Morgan Kaufmann, 1998.

[21] Sariel Har-Peled and Soham Mazumdar. Fast algorithms for computing the smallest k-enclosing circle. *Algorithmica*, 41(3):147–157, 2005. doi: 10.1007/s00453-004-1123-0. URL https://doi.org/10.1007/s00453-004-1123-0.

[22] Haim Kaplan, Sasanka Roy, and Micha Sharir. Finding axis-parallel rectangles of fixed perimeter or area containing the largest number of points. *Comput. Geom.*, 81:1–11, 2019. doi: 10.1016/j.comgeo.2019.01.007. URL https://doi.org/10.1016/j.comgeo.2019.01.007.

[23] Der-Tsai Lee. On k-nearest neighbor voronoi diagrams in the plane. *IEEE Trans. Computers*, 31(6):478–487, 1982. doi: 10.1109/TC.1982.1676031. URL https://doi.org/10.1109/TC.1982.1676031.

[24] Priya Ranjan Sinha Mahapatra, Arindam Karmakar, Sandip Das, and Partha P. Goswami. k-enclosing axis-parallel square. In Beniamino Murgante, Osvaldo Gervasi, Andrés Iglesias, David Taniar, and Bernady O. Apduhan, editors, *Computational Science and Its Applications - ICCSA 2011 - International Conference, Santander, Spain, June 20-23, 2011. Proceedings, Part III*, volume 6784 of *Lecture Notes in Computer Science*, pages 84–93. Springer, 2011. doi: 10.1007/978-3-642-21931-3_7. URL https://doi.org/10.1007/978-3-642-21931-3_7.

[25] Jirí Matousek. Range searching with efficient hierarchical cuttings. In David Avis, editor, *Proceedings of the Eighth Annual Symposium on Computational Geometry, Berlin, Germany, June 10-12, 1992*, pages 276–285. ACM, 1992. doi: 10.1145/142675.142732. URL https://doi.org/10.1145/142675.142732.

[26] Jirí Matousek. Approximations and optimal geometric divide-an-conquer. *J. Comput. Syst. Sci.*, 50(2):203–208, 1995. doi: 10.1006/jcss.1995.1018. URL https://doi.org/10.1006/jcss.1995.1018.

[27] Jirí Matousek. On enclosing k points by a circle. *Inf. Process. Lett.*, 53(4):217–221, 1995. doi: 10.1016/0020-0190(94)00190-A. URL https://doi.org/10.1016/0020-0190(94)00190-A.

[28] Nimrod Megiddo. Applying parallel computation algorithms in the design of serial algorithms. *J. ACM*, 30(4):852–865, 1983. doi: 10.1145/2157.322410. URL https://doi.org/10.1145/2157.322410.

[29] Mark H. Overmars and Chee-Keng Yap. New upper bounds in klee's measure problem. *SIAM J. Comput.*, 20(6):1034–1045, 1991. doi: 10.1137/0220065. URL https://doi.org/10.1137/0220065.

[30] Franco P Preparata and Michael I Shamos. *Computational geometry: an introduction*. Springer Science & Business Media, 2012.

[31] Michael O. Rabin. Probabilistic algorithms in finite fields. *SIAM J. Comput.*, 9(2):273–280, 1980. doi: 10.1137/0209024. URL https://doi.org/10.1137/0209024.

[32] Arnold Schönhage. On the power of random access machines. In Hermann A. Maurer, editor, *Automata, Languages and Programming, 6th Colloquium, Graz, Austria, July 16-20, 1979, Proceedings*, volume 71 of *Lecture Notes in Computer Science*, pages 520–529. Springer, 1979. doi: 10.1007/3-540-09510-1_42. URL https://doi.org/10.1007/3-540-09510-1_42.

[33] Michael Segal and Klara Kedem. Enclosing k points in the smallest axis parallel rectangle. *Inf. Process. Lett.*, 65(2):95–99, 1998. doi: 10.1016/S0020-0190(97)00212-3. URL https://doi.org/10.1016/S0020-0190(97)00212-3.

[34] Michael Ian Shamos and Dan Hoey. Closest-point problems. In *16th Annual Symposium on Foundations of Computer Science, Berkeley, California, USA, October 13-15, 1975*, pages 151–162. IEEE Computer Society, 1975. doi: 10.1109/SFCS.1975.8. URL https://doi.org/10.1109/SFCS.1975.8.

[35] Micha Sharir. On k-sets in arrangement of curves and surfaces. *Discret. Comput. Geom.*, 6:593–613, 1991. doi: 10.1007/BF02574706. URL https://doi.org/10.1007/BF02574706.