Geometric Measures of Depth

Mathies Boile Christensen, 20104810
Thomas Sandholt, 20103476

Master's Thesis, Computer Science
July 2015
Advisors: Peyman Afshani and Gerth Stoelting Brodal

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

Modern technology has greatly promoted the collection of large scale multivariate data and analysing it is of interest in many statistical applications. Extending the univariate measures is insufficient as multivariate data lacks a natural order in Euclidean space. The concept of depth solves this problem by defining the centrality of a point relative to the input. Many notions of depth have been introduced of which the simplicial and halfspace depth are very prominent. In this thesis we will implement efficient algorithms for computing the depth of a point in $\mathbb{R}^2$, using the two notions of depth, and demonstrate that robust approximations can be achieved by uniformly sampling the input. We will also study the problem of finding a point that best describes the data in $\mathbb{R}^2$ with respect to the halfspace median and present an $O(n^{1+\epsilon})$ algorithm that use an array of simple algorithmic techniques.

Acknowledgements

We wish to express our sincere gratitude to our advisor Peyman Afshani for guiding us during our work on this thesis. He always took time to answer our questions and discuss new approaches and problems. Without his patience and immense knowledge, this thesis would have been near impossible.

We would also like to thank family and friends for moral support, especially Lukas Walther for contributing with constructive discussions and comments.

Mathies Boile Christensen,
Thomas Sandholt,
Aarhus, July 1, 2015.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Consider a country far far away with a population of 1000 citizens ruled by a vicious dictator. This dictator is accused by the U.N. of not paying his people enough to maintain a decent standard of living. The current distribution of salaries in the country is such that the dictator is paid 1.000.000.000 smackers while each citizen is paid 1 smack. When questioned by the U.N. the dictator simply presents the mean wage and argues that 1.000.000 smackers in average income is more than enough to survive. Thus, the U.N. concludes that the accusation is a lie and drops the case. However, upon further investigation by a diligent U.N. worker, the median wage of the country turns out to be 1 smack and the case reopens.

The above example illustrates how outliers can affect an estimator, which in the worst case may lead to faulty conclusions. Therefore it is important that an estimator is robust, i.e. it can handle noise and outliers. In this one-dimensional case, the problem is solved by using the median instead of the mean, which proves to be a more robust estimator.

More formally, given a set of points whose underlying probability distribution is unknown, or non-parametric and unimodal, the important data analytical task is to estimate a point which best describes the set. For any definition of such a point it is important to consider the issues of robustness and efficiency: How many outlying observations can an estimator tolerate before giving an incorrect result and can such an estimator be found efficiently? The presence of outliers also gives rise to the related problem of ranking and ordering points by determining their outlierness, or equivalently their depth. These two problems are of great relevance in any application where data analysis is paramount, but are computationally challenging given a multivariate setup. In this thesis we seek to solve these problems efficiently in $\mathbb{R}^2$ using the concept of data depth.

1.1 Background and related work

Data depth is an important concept and leads to the generalization of many univariate measures into a multivariate setting. One such measure is the estimator described above. The mean is fast to compute in any dimension and is straight forward to apply in practice. The major disadvantage is that it falls

short in terms of robustness because a single outlier has great influence on the result. To further reason about robustness we formalize the concept of the breakdown point.

Definition 1.1. The breakdown point is the proportion of data which must be moved to infinity so that the estimator will do the same.

The problem with the mean in $\mathbb{R}^1$ is that the breakdown point is $\frac{1}{n}$ because a single point at infinity will move the mean to infinity. A more robust alternative is the median which has a breakdown point of $\frac{1}{2}$. In fact, it has been show in [A17] that the maximum breakdown point for any estimator is $\frac{1}{2}$ meaning that the median is optimal in $\mathbb{R}^1$. Intuitively, the breakdown point cannot exceed $\frac{1}{2}$ because if more than half of the data points are outliers then it is not possible to distinguish between the underlying distribution and the outliers. Even though the median is a robust estimator it is only defined on ordered one-dimensional data, thus a generalized multivariate definition is needed for data in more than one dimension.

The generalization of the univariate median is an old research topic and not all solutions rely on the concept of depth. One of the first suggestions for generalizing the median was to combine several univariate medians along the different dimensions of the data. An example of this idea is the vector-of-medians from 1902 by Hayford [A7], but as was shown by Rousseeuw in [A16] the problem with this idea is that it is possible to generate a median outside the convex hull of the data points. Soon after Hayford's definition, Weber [A1] proposed to use the point which minimizes the sum of the Euclidean distances to all data points as an estimator. This estimator is referred to as the geometric median and coincides with the univariate median in one dimension. It has been show in [A16] that the breakdown point of the geometric median in $\mathbb{R}^d$ is $\frac{1}{2}$, hence it is optimal with respect to robustness. Unfortunately, the geometric median has a major drawback: It is unknown whether it can be computed exactly for $n > 3$ in more than one dimension. All known algorithms to this day approximate the geometric median and alternate definitions may be considered depending on the application.

Another definition that generalizes the univariate median is based on a popular depth measure, namely the halfspace depth. In 1929 Hotelling [A6] described the univariate median as the point which minimizes the maximum number of data points on one of its sides. This idea was generalized by Tukey [A11] into the halfspace depth and later into the halfspace median which has been subject to a lot of research. In order to define the halfspace median, we must first consider the depth function. The median follows directly as it is the point with the largest depth.

Definition 1.2. Given a set $P \subset \mathbb{R}^d$ of size n and a query point $q \in \mathbb{R}^d$, the halfspace depth $\mathrm{D}_{hs}(q)$ is defined as

$$\mathrm{D}_{hs}(P, q) = \min\{|P \cap \gamma| : \text{over all closed halfspaces } \gamma \text{ containing } q\}.$$

Definition 1.3. Given a set $P \subset \mathbb{R}^d$ of size n, the halfspace median $\mathrm{M}_{hs}(P)$ is defined as

$$\mathrm{M}_{hs}(P) = \arg\max_{q \in \mathbb{R}^d} \mathrm{D}_{hs}(P, q).$$

The halfspace median is generally not a unique point, but the set of all points at maximum depth is guaranteed to be a closed and bounded convex set. It has a breakdown point between $\frac{1}{d+1}$ and $\frac{1}{3}$ as show in [A3] and therefore serves as a robust estimator. Unfortunately, computing the halfspace median is a very time consuming task and efficient algorithms are either conceptually difficult or complex to implement. In order to emphasize the severity of the problem, a naive algorithm in $\mathbb{R}^2$ runs in $O(n^6)$. An improvement to this was made by Rousseeuw et al. in [A18] by calculating the depth of a single point in $O(n \log n)$ resulting in a $O(n^5 \log n)$ algorithm. Later they presented a more complicated $O(n^2 \log^2 n)$ algorithm along with an implementation [A19]. In 1991 Matoušek [A8] presented a $O(n \log^5 n)$ algorithm by showing how to compute some point with depth greater than some parameter k in $O(n \log^4 n)$ and combining it with binary search on k. The algorithm was later improved by Langerman et al. [A21] to find the median in $O(n \log^4 n)$. Finally, in 2004 Chan [A22] presented an $O(n \log n + n^{d-1})$ optimal randomized algorithm for computing the halfspace median in $\mathbb{R}^d$. This algorithm is the best known to date, but Chan perceives the problem from a purely theoretical point of view and suspects that the algorithm has very large constants. Consequently, he doubts its practical purpose and suggests that a simpler solution in $\mathbb{R}^2$ may not be as inefficient as one thinks.

A competing depth measure was first introduced by Liu in [A20] and stems from another generalization of the univariate median, namely the simplicial median. The basic idea extends the univariate median when seen as the point contained in the largest number of intervals constructed by the input. This naturally suggests to use the number of intervals a point is contained in as a depth measure. These intervals generalizes to simplicies in more than one dimension and leads to the following definition of simplicial depth.

Definition 1.4. Given a set $P \subset \mathbb{R}^d$ of size n and a query point $q \in \mathbb{R}^d$, the simplicial depth $\mathrm{D}_{sim}(P, q)$ is defined as the number of simplicies formed by $d + 1$ data points from P that contain q.

Again, the simplicial median follows directly from this definition as it is the point with the largest depth. Unfortunately, this definition of depth appears to be even more challenging than the halfspace depth when the task at hand is to compute an estimator. In $\mathbb{R}^2$ the best known algorithm was invented by Aloupis et al. in [A5] and runs in $O(n^4)$. Furthermore, the simplicial median has a worse breakdown point than the halfspace median which is shown by Chen in [A24]. One thing the two depth measures have in common is that the depth of a point can be calculated in $O(n \log n)$ using the algorithm described by Rousseeuw et al. in [A18]. Even though this is considered efficient, the process of ranking a set of points becomes a $O(n^2 \log n)$ algorithm, but can be improved by sampling the input.

In conclusion, the problem of finding a robust estimator using a depth measure is a computationally difficult challenge and efficient solutions are required to make use of it in practice. The notion of depth also makes it possible to rank data points in order to determine their outlierness. This task is less time consuming, but does not scale well for ranking an entire set of points.

1.2 Thesis statement

Our contribution and objective of this master thesis is divided into two parts. Firstly, we will describe, implement and analyze algorithms in $\mathbb{R}^2$ that compute the halfspace and simplicial depth of point and test their efficiency and how the depth measures are affected by uniformly sampling the input. Secondly, as suggested by Chan, we will describe, implement and analyze a simpler algorithm that computes the halfspace median in $\mathbb{R}^2$ and tests its efficiency in practice. To do this, we use an array of computational techniques of which we will describe and analyze the most extensive ones in great detail, in order to verify the final algorithm.

1.3 Computer specifications and source code

All experiments are executed on an Asus UX31A notebook with an Intel core i7-3517U processor with 2 cores (4 threads) and 4 GB memory. There is a single L1 instruction cache and L1 data cache per core, each of size 32 KB. There is also one unified L2 cache of size 256 KB per core. The unified L3 cache is shared by all cores and can contain 4096 KB. The page size is 4 KB and the cache line is 64 bytes. The L1 instruction TLB and L1 data TLB can contain 64 entries each and the unified L2 TLB can contain 512 entries. The code is compiled with `clang++` version 3.4-1ubuntu3 using the 03 optimization flag on Ubuntu 14.04 LTS.

The source depends on `GMP` which is a multiple precision library for doing arithmetic with large to infinite precision. The source also depends on `PAPI` which is a library for counting hardware events. Finally, the source includes `PEMPEK` which is a library that makes it easier to use assertions in production and test code. The assertion library is included and precompiled inside the package but `GMP` and `PAPI` needs to be installed on the host operating system. The source code can be downloaded at `https://www.dropbox.com/s/yzhxyk4898jrbo5/thesis.zip?dl=0`.

1.4 Overview

The thesis consists of 7 chapters whereas this introduction is the first. In Chapter 2 we present the relevant and necessary theory used in the remaining chapters. In Chapter 3 we study the first part of the thesis statement concerning approximations of the halfspace and simplicial depth of a query point. We study the second part of the thesis statement in Chapter 4 through 6. In Chapter 4 we present a data-structure for storing an arrangement of lines and in Chapter

5 we present two $\frac{1}{r}$-cutting algorithms which uses this data-structure. The material presented in these two chapters are targeted at verifying correctness, improving performance and sets the stage for the algorithm computing the halfspace median presented in Chapter 6. Finally, the thesis is concluded in Chapter 7.

Chapter 2

Preliminaries

In this chapter we present the relevant and necessary theory, along with some fundamental geometric functions, for implementing and analyzing the algorithms used in this thesis. In this and the following chapters, we assume the random-access model when analyzing algorithm complexities.

2.1 Duality and arrangements

In this section we describe the theory of duality and arrangements of lines from [A12]. This topic is relevant since the problem of calculating the halfspace median is defined on a set of points, but may be solved efficiently in dual space where the arrangement of lines is a necessary tool.

In $\mathbb{R}^2$, the dualization of a point $p = (x, y)$ is given by the line

$$l = (x, -y) \tag{2.1}$$

where x denotes the slope and $-y$ denotes the intersection with the y-axis. This transformation ensures that there exists no vertical lines in the dual plane and it is incidence and order preserving.

Definition 2.1. The duality transformation is a bijective mapping defined by equation 2.1 and has the following properties.

- A point cannot dualize into a vertical line.
- It is incidence preserving meaning that points on a line maps to lines that goes through the dual of the line.
- It is order preserving meaning that a point lies above a line if and only if the dual of the line lies above the dual of the point.

A set of lines induces a subdivision that consists of vertices, edges and faces and is referred to as an arrangement of lines. The complexity is defined in the number of these structures and it is quadratic in the worst case.

Theorem 2.1. The arrangement induced by a set L of n lines in $\mathbb{R}^2$ has the following space complexity.

- The number of vertices is at most $\frac{n(n-1)}{2}$.
- The number of edges is at most n^2.
- The number of faces is at most $\frac{n(n+1)}{2} + 1$.

Proof. The number of vertices is equal to the number of intersections between the lines of L and there are at most

$$\binom{n}{2} = \frac{n(n-1)}{2}$$

such intersections. In order to prove the bound on the edges, we know that each line can at most be cut into n edges by $n-1$ vertices. Given that there are n lines there can at most be n^2 edges. Finally, the number of faces is increased by the number of edges that are created when inserting a line in the arrangement, because an edge splits a face in two. Taken the initial face into consideration there are at most

$$\left(\sum_{i=1}^{n} i\right) + 1 = \frac{n(n+1)}{2} + 1$$

faces in the final arrangement. Note, that if there are parallel lines or more than two lines pass through the same point the number of vertices, edges and faces decreases and the bounds still hold. □

The preferred data-structure for representing an arrangement of lines is a doubly-connected edge list. This data-structure creates a record for each vertex, edge and face and stores them such that they can be traversed and modified efficiently. The faces are not needed for implementing the algorithms in this thesis so they are left out. Edges are doubly-connected, meaning that all edges are oriented and have a twin edge pointing in the opposite direction. An edge must at least store a pointer to the twin edge together with a pointer to the previous edge and the next edge along with a pointer to the vertex that it originates from. A vertex must at least store the coordinates of its position and a pointer to an incident edge originating from it. A section of a doubly-connected edge list is depicted in Figure 2.1.

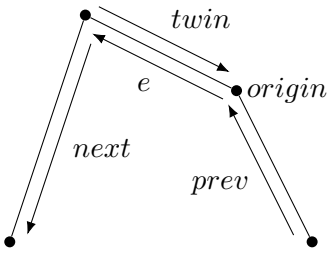


Figure 2.1: A section of a doubly-connected edge list.

Algorithms for constructing and traversing a doubly-connected edge list simply use these pointers. In order to analyze such algorithms we need to introduce the concept of zones. A zone is associated with a line l and consists

of the faces that the line intersects. An example is depicted in Figure 2.2. The complexity of a zone is given by the number of vertices and edges bounding the faces in the zone. It is not intuitive that its complexity is linear in the number of lines that forms the arrangement because vertices may be counted more than once. Nonetheless, as shown by the following theorem, it is true.

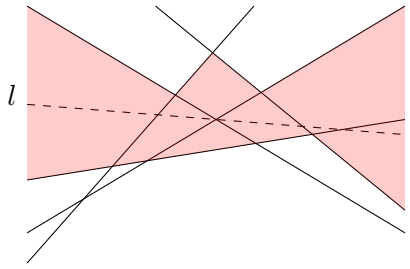


Figure 2.2: The zone of a line l in an arrangement of lines.

Theorem 2.2. Given an arrangement of n lines, the complexity of the zone associated with a line l is $O(n)$.

The formal induction proof of Theorem 2.2 can be found in the literature referred to in the beginning of this section and starts by defining left and right bounding edges. An edge is left bounding of the face that lies in the right halfspace of the line defining it. Symmetrically, an edge is right bounding if the opposite hold. Informally, the proof is carried by bounding the number of such edges by $10n$, i.e. by bounding left bounding edges by $5n$ and right bounding edges by $5n$. Considering left bounding edges, the induction step is proven by counting how many edges the rightmost edge l_1 that intersects l introduces. By convexity of faces this number is at most 5 and the theorem holds by induction.

2.2 Levels

In this section we present the relevant theory about levels in arrangements of lines as described in [A15]. The notion of levels is directly connected to computing the halfspace median in dual space as it constrains the maximum possible depth.

Given an arrangement of lines in general position, the upper level of a vertex is the number of lines strictly above it. The upper k-level is the closure of the set of vertices that have k lines strictly above them. Note that the upper k-level contains vertices with upper level k or $k-1$ in order for the closure to be well defined. Symmetrically, The lower level of a vertex is the number of lines strictly below it. The lower k-level is the closure of the set of vertices that have k lines strictly below them. Note that the lower k-level contains vertices with lower level k or $k-1$ in order for the closure to be well defined. An example of the lower 3-level vertices in an arrangement of lines is depicted in Figure 2.3.

Definition 2.2. Given an arrangement induced by n lines in general position we define the k-level as follows.

- The upper k-level is the closure of the set of vertices that have k lines strictly above them.
- The lower k-level is the closure of the set of vertices that have k lines strictly below them.

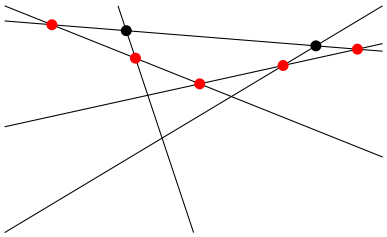


Figure 2.3: The lower 3-level in an arrangement of lines. The vertices with 2 lines strictly below them are red and the vertices with 3 lines strictly below them are black.

Unfortunately, the size of the k-level is super-linear and the best known upper bound is $O(nk^{1/3})$ [A23]. In practice this number is most likely smaller, but it remains an open problem whether a tighter bound exists.

Lemma 2.1. The number of vertices on the k-level is at most $O(nk^{1/3})$.

2.3 Range space, ϵ-approximations and ϵ-nets

In this section we recall the basic facts about range spaces, ϵ-approximations and ϵ-nets as described in [A2] and [A4]. The concept of ϵ-approximations is relevant for calculating the query depth, since the input space may be sampled, producing results with a tolerable error. Additionally, ϵ-nets are important in relation to calculating the maximum halfspace depth as the dual space may be partitioned into smaller problems by sampling lines from the input. In this case the theory provides on upper bound on the size of this partitioning.

A range space is a pair $(X, \mathcal{R})$ where X is a set of n objects and $\mathcal{R}$ is a set of subsets of X. We call the elements of $\mathcal{R}$ ranges and it is typically defined in terms of a geometric structure. Given a subset $Y \subseteq X$, we naturally get a sub space induced by Y - namely the range space $\mathcal{R}_{|Y} = (Y, \{Y \cap R : R \in \mathcal{R}\})$ which is obtained by intersecting Y with each $R \in \mathcal{R}$. Given the notion of range spaces we can define an ϵ-approximation and ϵ-net.

Definition 2.3. Given a range space $(X, \mathcal{R})$ and $\epsilon \in]0; 1[$, an ϵ-approximation is a subset $Y \subseteq X$ such that for each range $R \in \mathcal{R}$

$$\left| \frac{|R|}{|X|} - \frac{|Y \cap R|}{|Y|} \right| \leq \epsilon.$$

Definition 2.4. Given a range space $(X, \mathcal{R})$ and $\epsilon \in]0; 1[$, an ϵ-net is a subset $Y \subseteq X$ such that for each range $R \in \mathcal{R}$

$$|R| \geq \epsilon |X| \Rightarrow Y \cap R \neq \emptyset$$

The notion of ϵ-nets is weaker than that of ϵ-approximations because an ϵ-approximation is automatically an ϵ-net. To make any use of these definitions we need to define the concept of shattering and the VC-dimension. We say that a subset $Y \subseteq X$ is shattered if $|\mathcal{R}_{|Y}| = 2^Y$ and the range space $(X, \mathcal{R})$ is said to have VC-dimension d_{vc} if d_{vc} is the smallest integer such that no $d_{vc}+1$ size subset Y can be shattered. If no such limit exists the VC-dimension is infinite. A property about range spaces with finite VC-dimension is that there exists an upper bound on the size of ϵ-approximations and ϵ-nets that is independent of the size of X.

Lemma 2.2. For any finite range space $(X, \mathcal{R})$ with finite VC-dimension d_{vc} and $\epsilon \in]0;1[$, there exists an ϵ-net of size at most $O(\frac{d_{vc}}{\epsilon} \log \frac{d_{vc}}{\epsilon})$ and an ϵ-approximation of size at most $O(\frac{d_{vc}}{\epsilon^2} \log \frac{d_{vc}}{\epsilon})$.

With respect to ϵ-approximations, this lemma tells us that it is reasonable to approximate $\frac{|R|}{|X|}$ by taking a relatively small sample from X and computing $\frac{|Y \cap R|}{|Y|}$ instead. For ϵ-nets, the lemma simply gives an upper bound on its size. The results proven in [A4] also reveal that we can expect to create an ϵ-net within a constant probability by taking a random sample of size $O(\frac{d_{vc}}{\epsilon} \log \frac{d_{vc}}{\epsilon})$.

Lemma 2.3. For any finite range space $(X, \mathcal{R})$ with finite VC-dimension d_{vc} and $\epsilon, \delta \in]0;1[$, the subset $Y \subseteq X$ is an ϵ-net with probability $1 - \delta$ if Y is created by choosing

$$m = \max\left(\frac{4}{\epsilon} \log \frac{2}{\delta}, \frac{8d_{vc}}{\epsilon} \log \frac{8d_{vc}}{\epsilon}\right)$$

independent draws from X.

2.4 Query depth

We seek to apply the theory presented in Section 2.3 on the problems of finding the simplicial and halfspace depths of a point. In this section we will prove that the range spaces defining them have constant VC-dimension.

Theorem 2.3. The problem defined in Definition 1.4 (simplicial query depth) has finite VC-dimension.

Proof. The range space for the simplicial depth problem consists of X which is the set of all possible triangles constructed by the input points and $\mathcal{R}$ which is the set of sets of all unique triangle combinations containing a query point. Now, by the definition of shattering, $\mathcal{R}_{|Y}$ shatters $Y \subseteq X$ if it corresponds to the powerset of Y. Thus for any combination of triangles from Y it should be possible to place a query point inside only those triangles. An example of a set of triangles being shattered is given in Figure 2.4. Here the labels represent four different query points covering all the possible combinations of triangles. The main insight to gain from the example is that the different triangle combinations must correspond to a face in the arrangement constructed from the line segments of the triangles. This follows from the fact that for any given triangle

combination a query point being covered by only those triangles must lie inside a region covered by only those triangles. Armed with this it is now possible to prove that the simplicial depth problem has finite VC-dimension. We know, that in order for $\mathcal{R}_{|Y}$ to shatter Y it must contain $2^{|Y|}$ elements. This corresponds to saying that the arrangement created by the line segments of the triangles must contain $2^{|Y|}$ faces. From Theorem 2.1 we know that the maximum number of faces for any arrangement of n lines is bounded by $\frac{n(n+1)}{2} + 1 = O(n^2)$. Thus the simiplicial depth problem has finite VC-dimension, because if $|Y|$ is a large enough constant, then there will be fewer than $2^{|Y|}$ faces in the arrangement, meaning that Y is not shattered.

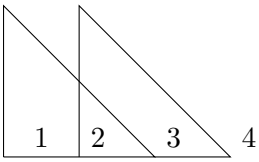


Figure 2.4: The possible simplicial depths of two triangles.

□

Theorem 2.3 states that the VC-dimension for the simplicial query depth problem is finite, but gives no exact bound on the VC-dimension. It is however possible to determine an upper bound using arrangement of lines theory. The actual bound on the number of faces for an arrangement of n lines is $\frac{n(n+1)}{2}+1$. Combining this with the fact that n triangles will generate $3n$ line segments, an upper bound for the number of faces in the arrangement created from n triangles is $\frac{3n(3n+1)}{2} + 1$. Hence, an upper bound of the VC-dimension is given by the equation

$$\frac{3n(3n+1)}{2} + 1 = 2^n$$

which has several solution of which $n \approx 8.35695$ is the largest. Thus we conclude that for the simplicial depth problem the VC-dimension is no more than 8 because $n = 9$ cannot be shattered.

Theorem 2.4. The problem defined in Definition 1.2 (halfspace query depth) has finite VC-dimension.

Proof. Instead of looking at the halfspace depth problem, we look at a generalized problem where the halfspace can be positioned anywhere in $\mathbb{R}^2$. The generalized problem consists of X which is a set of points in $\mathbb{R}^2$ and $\mathcal{R}$ which is the set of all possible sets of points contained in a halfspace. By the definition of shattering, $\mathcal{R}_{|Y}$ shatters $Y \subseteq X$ if it corresponds to the powerset of Y. Thus, we can prove that there exists a finite upper bound on the VC-dimension by showing that it is not possible to shatter a set of four points. Any set of four points can be partitioned into two subsets whose convex hulls intersect, hence there will always exist a point that no halfspace can separate from the rest of the points. Given four such points $p_1, p_2, p_3, p_4 \in \mathbb{R}^2$ there exists four

coefficients $a_1, a_2, a_3, a_4 \in \mathbb{R}$ that are not all zero such that the equations

$$\sum_{i=1}^{4} a_i p_i = 0 \tag{2.2}$$

$$\sum_{i=1}^{4} a_i = 0 \tag{2.3}$$

hold. Note that there are three equations and four unknown coefficients so the system of equations is underdetermined. Such a system is either consistent and has infinitely many solutions or inconsistent and has no solution. Inconsistencies occur when the left hand side equations are linearly dependent and the constants on the right hand side do not satisfy this dependency relation. In this case the right hand side is the zero vector, thus the dependency relation is always satisfied and the system is consistent. We now define the set I to contain the indices of the positive coefficients and the set J to contain the indices of the negative or zero coefficients. As a consequence of equation 2.3, and the fact that the coefficients are not all zero, neither I nor J are empty and the set of points they represent are convex because they contain between 1 and 3 indices. The claim is that there exists a point q in both of these convex sets such that they intersect. We can define q by rewriting equation 2.2 as below

$$q = \sum_{i \in I} \frac{a_i}{a} p_i = -\sum_{j \in J} \frac{a_j}{a} p_j$$

where $a > 0$ is defined by equation 2.3 as

$$a = \sum_{i \in I} a_i = -\sum_{j \in J} a_j.$$

Observe that q is a convex combination of both point sets represented by I and J because the coefficients are non-negative $\forall i \in I : \frac{a_i}{a} > 0$, $\forall j \in J : \frac{a_j}{a} > 0$ and the coefficients sum to one $\sum_{i \in I} \frac{a_i}{a} = \sum_{j \in J} \frac{a_j}{a} = 1$, hence q lies inside both sets by Proposition 7.1 and no set of four points can be shattered. □

In contrast to the proof of Theorem 2.3, the proof of Theorem 2.4 gives an exact upper bound on the VC-dimension. Thus, the VC-dimension of the halfspace query depth problem is no more than 3 because $n = 4$ cannot be shattered. As for the simplicial query depth problem, we do not need to determine the exact VC-dimension, but the fact that it is finite is sufficient in order to utilize Lemma 2.2 when approximating the depth in Chapter 3.

2.5 Cuttings

In this section we present the definition of and relevant theory about $\frac{1}{r}$-cuttings, which are used as a tool in a divide and conquer algorithm for computing the halfspace median, and prove an upper and lower bound on their size. This theory is closely connected to ϵ-nets presented in Section 2.3, thus we will also prove that the range space defining the problem has constant VC-dimension.

In general, a cutting is a disjoint partition of geometric objects. In this thesis we restrict our attention to the two-dimensional plane where the objects are lines. The partition that we are seeking is a set of open triangles with a certain property defined below.

Definition 2.5. Let L be a set of n lines in $\mathbb{R}^2$ and r a parameter where $n \geq r \geq 1$. A triangulation C of $\mathbb{R}^2$ is a $\frac{1}{r}$-cutting of L if each open triangle $\Delta \in C$ is crossed by at most $\frac{n}{r}$ lines from L and C covers the entire plane.

The size of a cutting is determined by the number of triangles that it creates. In relation to this, the size of a cutting may be bounded using ϵ-net theory, but in order to apply Lemma 2.2 we need to prove that the VC-dimension is finite.

Theorem 2.5. The problem defined in Definition 2.5 (cuttings) has finite VC-dimension.

Proof. We define the range space X as the set of n lines. Let s be an open segment and define R as the set of lines intersecting s. The ranges $\mathcal{R}$ is the set of R's over all open segments s. This range space generalizes the problem at hand because segments form the borders of any triangle. By the definition of shattering, $\mathcal{R}_{|Y}$ shatters $Y \subseteq X$ if it corresponds to the powerset of Y, i.e. there exists segments that intersect all combinations of lines from Y.

We claim that the number of sets in $\mathcal{R}_{|Y}$ is bounded by the number of face-face pairs formed by creating an arrangement of lines from Y. Clearly, any segment has to start in face and end in a face of the arrangement. If we can prove that all segments starting in the same face and ending in the same face intersects the same lines, the claim holds by Theorem 2.1. Assume for contradiction that there exists two segments s_1 and s_2 that start in the same face f_1 and ends in the same face f_2, but s_1 intersects a line l that s_2 does not. The line l then intersect either f_1 or f_2, otherwise it will intersect intersect s_2. Let f_1 be the face that l intersects and consequently cuts in half. The segments s_1 and s_2 now start in different faces contradicting the assumption, hence the claim holds true.

By Theorem 2.1 the number of faces is $O(n^2)$ meaning that there exists $O(\binom{n^2}{2})$ face-face pairs. Since this is bounded by $O(n^4)$ the VC-dimension is finite, because if $|Y|$ is a large enough constant, then there will be fewer than $2^{|Y|}$ face-face pairs in the arrangement, meaning that Y is not shattered. □

At this point, one could hope to build a cutting of n lines that is guaranteed to be smaller than the trivial $O(n^2)$. In fact, the smallest $\frac{1}{r}$-cutting one can achieve has size $\Omega(r^2)$ and is proven below.

Lemma 2.4. The size of a $\frac{1}{r}$-cutting of a set of n lines L in $\mathbb{R}^2$ is $\Omega(r^2)$.

Proof. Let C be a $\frac{1}{r}$-cutting of L. Each triangle in C is intersected by at most $\frac{n}{r}$ lines by Definition 2.5. This means that any triangle in C will contain $O((\frac{n}{r})^2)$ vertices since every intersecting line can intersect any other intersecting line. From Theorem 2.1 we know that for n lines in general position the number of

vertices in its arrangement is $\Omega(n^2)$. The number of needed triangles is therefore

$$\Omega\left(\frac{n^2}{(\frac{n}{r})^2}\right) = \Omega\left(n^2 \cdot \frac{r^2}{n^2}\right) = \Omega(r^2)$$

which proves the lower bound. □

In fact it is possible to prove that the upper bound is $O(r^2)$ as well. Instead, we will prove slightly weaker upper bound which is used in Chapter 5.

Lemma 2.5. There exists a $\frac{1}{r}$-cutting of a set of n lines L in $\mathbb{R}^2$ of size $O(r^2 \log^2 r)$.

Proof. We define the range space $(X, \mathcal{R})$ as in the proof of Theorem 2.5 and choose a $\frac{1}{2r}$-net denoted $Y \subseteq X$. By Theorem 2.5 we know that the VC-dimension is finite so the size of Y is

$$\begin{aligned} O(\frac{d_{vc}}{\frac{1}{2r}} \log \frac{d_{vc}}{\frac{1}{2r}}) &= O(2rd_{vc} \log 2rd_{vc}) \\ &= O(r(\log 2 + \log r + \log d_{vc})) \\ &= O(r \log 2 + r \log r + r \log d_{vc}) \\ &= O(r + r \log r + r) \\ &= O(r \log r) \end{aligned}$$

due to Lemma 2.2. We construct the arrangement induced by Y and triangulate it to obtain a cutting C. The triangulation is trivial since all faces are convex.

First we need to show that C is a valid cutting. Any edge e from C does not intersect any line from Y, hence e intersects at most $\frac{n}{2r}$ lines from X, which follows from Definition 2.4. Since a line must intersect exactly 2 edges in order to intersect a triangle, we can conclude that each triangle intersect at most $\frac{n}{r}$ lines, thus C is a valid cutting as the arrangement covers the entire plane.

Finally, we need to prove that the size of the cutting is $O(r^2 \log^2 r)$. Given that the size of Y is $O(r \log r)$ we know by Theorem 2.1 that there exists $O(r^2 \log^2 r)$ vertices in the arrangement of Y. The same thing holds for the cutting C, because the triangulation does not introduce new vertices. Since the number of vertices is an upper bound on the number of triangles the size of C is $O(r^2 \log^2 r)$ and the statement is proven. □

In Lemma 2.5 we claim that the number of vertices bounds the number of triangles. In order to support this claim we show that any triangulation of n points has $O(n)$ triangles. We know that each triangle has exactly 3 edges, each internal edge belongs to 2 triangles and each edge on the convex hull belongs to 1 triangle. This gives rise to the equation

$$3t = 2(e - k) + k = 2e - k \tag{2.4}$$

where t denotes the triangles, e denotes the internal edges and k denotes the edges on the convex hull. Using Euler's formula $2 = (t + 1) + n - e$ we can now

determine the number of edges as

$$\begin{aligned} 2 &= (t+1) + n - e \\ e &= t - 1 + n \\ 3e &= 3t - 3 + 3n = 2e - k - 3 + 3n \\ e &= 3n - k - 3, \end{aligned}$$

insert it into equation 2.4 we get

$$\begin{aligned} 3t = 2e - k &= 6n - 2k - 6 - k = 6n - 3k - 6 \\ t &= 2n - k - 1 \end{aligned}$$

and realize that $t = O(n)$ because k is equal to the number of vertices on the convex hull.

2.6 Geometric functions

In this section we present two basic geometric functions in two-dimensional space: One for determining the sidedness of a point with respect to a line and another for computing the angle between two points. These functions are used as subroutines for the implemented algorithms and are stated here for clarity.

The first problem consists of determining sidedness between points and boils down to computing the dot product given by equation 2.5. The equation is stated and proven below.

Proposition 2.1. Given points $p, q, r \in \mathbb{R}^2$, the sidedness of r with respect to the line going through p and q is determined by:

$$\mathrm{S}(p,q,r) = (q_x - p_x)(r_y - p_y) - (q_y - p_y)(r_x - p_x) \tag{2.5}$$

where a positive result means right, a negative result means left and zero means the points are colinear.

Proof. The correctness is a consequence of basic geometry, so we create the vectors $u = \vec{pq}$ and $v = \vec{pr}$ as depicted in Figure 2.5a and examine the cross product of these vectors in three dimensional space

$$\begin{aligned} u \times v &= \begin{pmatrix} u_2v_3 - u_3v_2 & u_3v_1 - u_1v_3 & u_1v_2 - u_2v_1 \end{pmatrix}^T \\ &= \begin{pmatrix} 0 & 0 & u_1v_2 - u_2v_1 \end{pmatrix}^T. \end{aligned}$$

Only the third entry of the cross product can be non-zero because the input is two-dimensional, i.e $u_3 = v_3 = 0$. Furthermore, the cross product is anti-commutative, i.e. $u \times v = -(v \times u)$, meaning that the sign of the cross product depends on the order of u and v as depicted in Figure 2.5b. Thus, the sign of $(u_1v_2 - u_2v_1)$ determines whether r is on the left or right side of the line going through p and q. When $u_1v_2 - u_2v_1 = 0$ we have that $u_1v_2 = u_2v_1$ which holds when v is a scaled version of u or vice verse. In that case the points are colinear. □

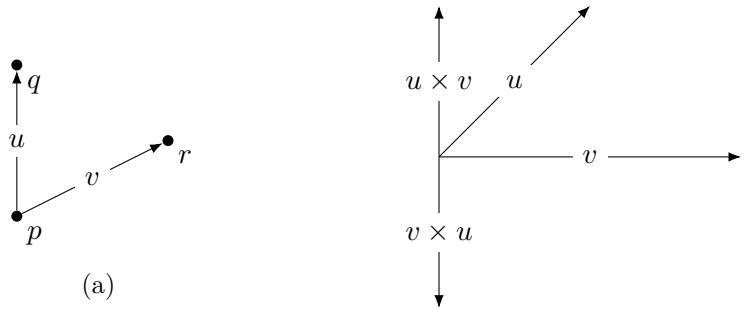


Figure 2.5: Geometric interpretation of computing the sidedness of a point r with respect to a line going through two points p and q.

In terms of robustness, equation 2.5 may lead to incorrect results due to round-off errors when the true cross product is near zero, provided that the coordinates are expressed using floating point precision. One solution is to use exact arithmetic using rational numbers but that comes with a significant slowdown. Another approach that uses floating point precision is presented by Shewchuk in [A10] who offers an effective and robust orientation test. This algorithm depends on the system architecture but could be employed given that the application at hand demands both speed and robustness.

The other function for determining the angle between two points is given by equation 2.6 and it also suffers from floating point errors because the vector between the two points is normalized using the square root and the result is based on the inverse sine and cosine functions. In order to remedy some of the errors we use the inverse sine and cosine functions whenever they are most accurate. For example, the inverse cosine function is not very accurate at $0°$ and $180°$, but is fairly accurate between $45°$ and $135°$, whereas the inverse sine function becomes inaccurate at $-90°$ and $90°$ but improves between $-45°$ and $45°$. The correctness is proven below and concludes the section.

Proposition 2.2. Given points $p, q \in \mathbb{R}^2$ and let $u = q - p$ be a vector, the angle θ in radians from p to q with respect to the line $y = 0$ is determined by

$$\mathrm{A}(p,q) = \begin{cases} \arccos(\hat{u}_x) & \text{if } q_y > p_y \text{ and } \hat{u}_x \leq \hat{u}_y \text{ ①} \\ 2\pi - \arccos(\hat{u}_x) & \text{if } q_y < p_y \text{ and } \hat{u}_x \leq \hat{u}_y \text{ ②} \\ \arcsin(\hat{u}_y) & \text{if } q_x > p_x \text{ and } q_y > p_y \text{ and } \hat{u}_x > \hat{u}_y \text{ ③} \\ 2\pi + \arcsin(\hat{u}_y) & \text{if } q_x > p_x \text{ and } q_y < p_y \text{ and } \hat{u}_x > \hat{u}_y \text{ ④} \\ \pi - \arcsin(\hat{u}_y) & \text{if } q_x < p_x \text{ and } \hat{u}_x > \hat{u}_y \text{ ⑤} \end{cases} \tag{2.6}$$

where $\hat{u}_x = \frac{u}{\|u\|}$ and $\|u\| = \sqrt{(u_x)^2 + (u_y)^2}$.

Proof. The angle that we seek to determine is exemplified in Figure 2.6a and we want to argue that each case of equation 2.6, which is associated with a region of Figure 2.6b, computes the correct angle. The correctness is a consequence of basic geometry, so we recall that given a triangle with edges a, b, c where a is the adjacent edge, b is the opposite edge and c is the hypotenuse, the angle θ can be

computed using the inverse sine function $\theta = \arcsin(b/c)$ or the inverse cosine function $\theta = \arccos(a/c)$. In this case where u is normalized, the hypotenuse is 1 so the angle can be computed solely from a or b. This means that the angle is correctly computed by $\arccos(\hat{u}_x)$ when q is in the upper half of the unit circle and by $2\pi - \arccos(\hat{u}_x)$ when q is in the lower half of the unit circle. Thus, region 1 an 2 are handled correctly. Furthermore, $\arcsin(\hat{u}_y)$ computes the correct angle in the right half of the unit circle except that the lower half results in negative angles, which is accounted for by adding 2π to the result. Thus, region 3 and 4 are handled correctly. Lastly, $\pi - \arcsin(\hat{u}_y)$ computes the correct angle in the left half of the unit circle because the lower half results in negative angles, hence region 5 is also handled correctly.

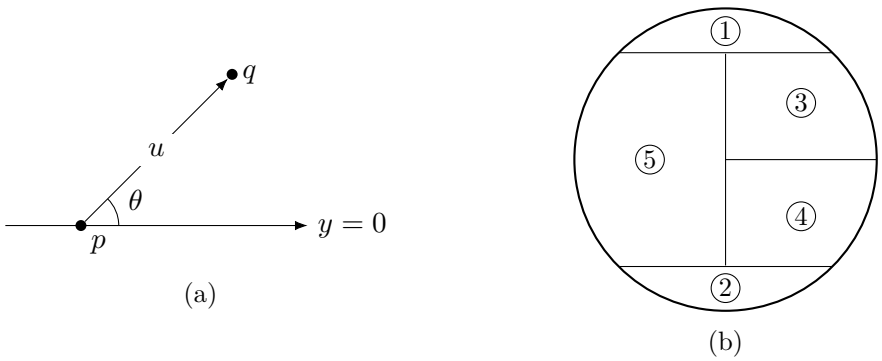


Figure 2.6: Geometric interpretation of computing the angle θ from p to q.

□

Chapter 3

Query depth

In this chapter we study the problem of ranking points with respect to the simplicial and halfspace depth in $\mathbb{R}^2$. We present a naive and sort based algorithm for computing the simplicial depth in Section 3.1, and a naive and sort based algorithm for computing the halfspace depth in Section 3.2. There is no general position assumption as all algorithms in this chapter handle coinciding points and more than two points on a line. In Section 3.3 we present a brief description of the tests performed and some considerations about robustness. In Section 3.4 we experiment with the efficiency of the algorithms and compare them with the theoretic bounds. We also experiment with the effects of approximating the depth in Section 3.5, using the theory from Section 2.3, in order to rank points and determine their outlierness. The chapter is concluded in Section 3.6.

3.1 Simplicial query depth algorithms

The simplicial depth of a point q is defined as the number of simplicies formed by the input points that contain q as formalized in Definition 1.4. In $\mathbb{R}^2$ these simplicies specializes into triangles. An example of a query with depth 4 is depicted in Figure 3.1 and naturally proposes a naive algorithm that counts all such triangles.

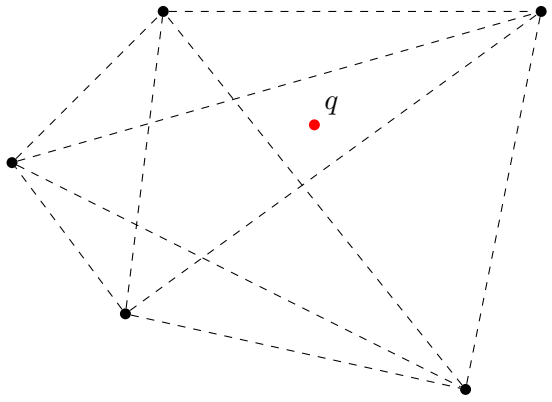
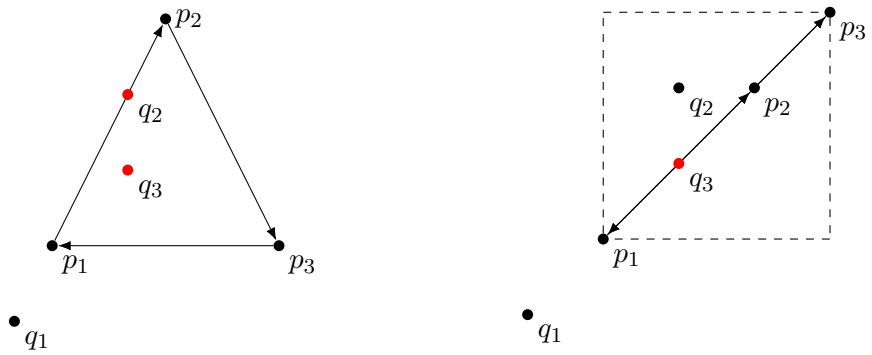


Figure 3.1: A query q with simplicial depth 4.

The naive algorithm is denoted `QUERY_SIM_NAIVE` and forms all possible triangles from P and increments the depth whenever the query q is contained in a triangle. The check is straightforward and consists of three sidedness tests given by equation 2.5, one for each segment bounding the triangle, and is carried out clockwise or counter-clockwise depending on the order in which the points are examined. If q is on the same side of the lines spanned by the boundaries of the triangle, the depth is incremented, which q_2 and q_3 in Figure 3.2a exemplifies. The sidedness test handles degenerate input with more than two colinear points as long as q is not colinear with all three points from the current triangle. In that case q is contained in the triangle if it is inside the bounding box of the triangle. The special case is depicted in Figure 3.2b where q_1 is on all three lines spanned by the segments of the triangle, but is discarded because it is outside of the bounding box. The correctness and complexity is proven below.



(a) A non-degenerate triangle.

(b) A degenerate triangle.

Figure 3.2: A non-degenerate and degenerate triangle.

Theorem 3.1. Given a set $P \subset \mathbb{R}^2$ of size n and a query point $q \in \mathbb{R}^2$, `QUERY_SIM_NAIVE` computes $\mathrm{D}_{sim}(P, q)$ in $O(n^3)$ time and uses $O(n)$ space.

Proof. Without loss of generality we look at a particular triangle $\Delta = \Delta(p_1, p_2, p_3)$ and assume that the order is clockwise. Ignoring the degenerate case for now, q is either outside or inside of the boundaries of Δ. When q is outside it is to the left of at least one of the boundaries and when q is inside it is to the right of all the boundaries, thus the depth is incremented correctly which follows from equation 2.5. The same logic holds in the degenerate case, except when p_1, p_2, p_3 and q are colinear. In this case, the depth is incremented when q_x is in between the largest and smallest x-coordinate of p_1, p_2 and p_3, and q_y is in between the largest and smallest y-coordinate of p_1, p_2 and p_3.

There exists $\binom{n}{3}$ triangles and `QUERY_SIM_NAIVE` looks at each triangle a constant number of times and executes a constant number of instructions every time. Thus, the running time is $O(n^3)$. The space complexity is clearly linear because the input has to be stored in memory at all times. □

The naive algorithm is simple to implement but the running time makes it useless in practice. An improved algorithm is described by Rousseeuw et al. in

[A18] which utilizes the order of the input points with respect to the query. The algorithm is denoted `QUERY_SIM_SORT` and computes the simplicial depth as the complement of the number of triangles that cannot contain q. Initially, all points $p_i \in P$ are sorted in ascending order on their angles α_i to q given by equation 2.6. Points coinciding with q are skipped and $\binom{n_{on}}{1}\binom{n_{off}}{2} + \binom{n_{on}}{2}\binom{n_{off}}{1} + \binom{n_{on}}{3}$ is added to the depth, where n_{on} is the number of points coinciding with q and n_{off} is the complement. If there exists a gap strictly larger than π between any two consecutive angles, the algorithm returns the calculated depth. Otherwise the smallest angle is subtracted from all other angles and the first occurring angle α_k strictly larger than π is remembered.

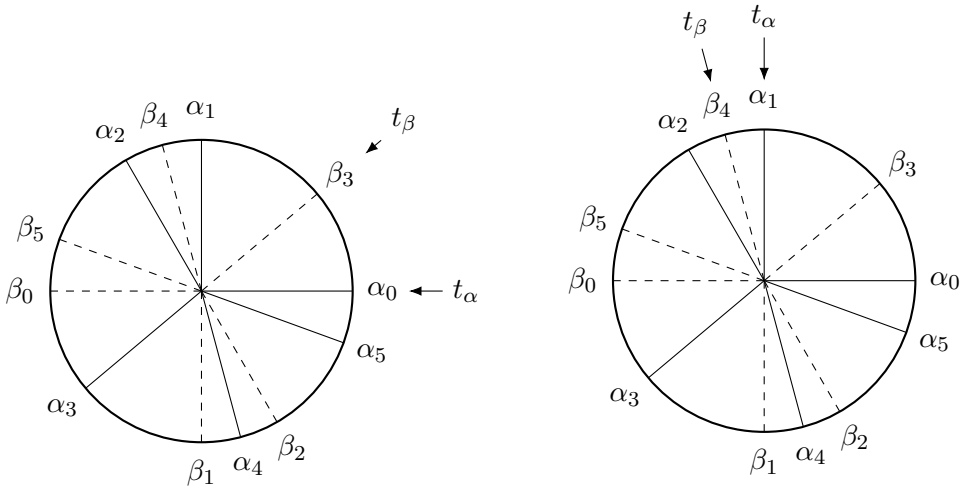
The final step of the algorithm consists of a scan through the α angles, with the antipodal angles β intertwined, in order to count the number of triangles that cannot contain q. An antipodal angle is given by $\beta_i = \alpha_i + \pi \mod 2\pi$ and the scan consists of traversing these $2n$ angles in sorted order. However, there is no need to add the antipodal β angles to the array - they can simply be calculated on the fly. During the scan two pointers are maintained: One for the current α angle denoted t_α and one for the current β angle denoted t_β. Initially t_α points to α_0 while t_β points to the antipodal angle of α_k. Furthermore, two counters c_1 and c_2 are also maintained during the scan. The counter c_1 denotes the number of angles in between t_α and the antipodal angle of t_β when traversing the angles in counter clockwise order from the antipodal angle of t_β to t_α. The counter c_2 is the current number of triangles that cannot contain q. Initially $c_1 = n_{off} - k - 1$ and $c_2 = 0$. The setup before the scan begins is exemplified in Figure 3.3a.

If $t_\alpha = \alpha_i < \beta_j = t_\beta$ then c_1 is incremented and t_α is updated to point to α_{i+1}. Otherwise c_2 is incremented by the number of triangles $\binom{c_1}{2}$, c_1 is decremented and t_β is updated to point to β_{j+1}. The state after the first two steps is exemplified in Figure 3.3b where $c_2 = \binom{3}{2} = 3$ because there can be made 3 triangles when fixing α_3 and choosing from $\alpha_4, \alpha_5, \alpha_0$ that does not contain q. The scan stops when all α and β angles have been examined and the resulting depth is incremented by $\binom{n_{off}}{3} - c_2$. The correctness and complexity is proven below.

Theorem 3.2. Given a set $P \subset \mathbb{R}^2$ of size n and a query point $q \in \mathbb{R}^2$, `QUERY_SIM_SORT` computes $\mathrm{D}_{sim}(P, q)$ in $O(n \log n)$ time and uses $O(n)$ space.

Proof. Clearly, the number of triangles that contain q is the complement of the triangles that do not contain q, so we want to prove that the algorithm counts all such triangles and computes the complement. For now we ignore the points coinciding with q and note that all indices are modulo n.

Let $\Delta(p_1, p_2, p_3)$ be a triangle and let the angles from q to p_1, p_2 and p_3 be denoted by α_1, α_2 and α_3. Clearly, the triangle cannot contain q if there exists an interval strictly smaller than π such that α_1, α_2 and α_3 are contained in the interval. We now define the root of all such triangles to be the angle furthest in the clockwise direction. Coinciding angles have some fixed arbitrary order. This ordering is well defined since all intervals are strictly smaller than π. The set of all triangles that do not contain q can now be partitioned into disjoint subsets of triangles, one for each root. Observe that for any root α_i, the



(a) The setup before the scan.

(b) The setup after two steps into the scan.

Figure 3.3: Illustrations during the scan of `QUERY_SIM_SORT`.

corresponding set consists of $\binom{m_i}{2}$ triangles where m_i is the number of angles in between α_i and $\alpha_i + \pi$ when traversing the angles in counter clockwise order from α_i to $\alpha_i + \pi$. Angles coinciding with α_i are included when deemed larger than α_i determined by the fixed arbitrary order. Thus, we need to prove that c_2 is the size of the union of all these sets when the scan of the α and β angles is complete.

Each time the algorithm updates t_β the size of the set defined by the root α_i, corresponding to the antipodal angle of t_β, is added to c_2. Hence, c_1 must be equal to m_i every time t_β is updated. This holds for the first root α_k because c_1 is initialized to be the number of angles in between α_k and $t_\alpha = \alpha_0$ when traversing the angles in counter clockwise order from α_k to t_α and c_1 is incremented once for each α angle until $t_\alpha \geq t_\beta$.

When t_β is updated, c_1 is correctly decremented because α_k cannot be in between α_{k+1} and β_{k+1} in the counter clockwise direction from α_{k+1} to β_{k+1} since that would require α_k and α_{k+1} to be strictly more than π apart, meaning that the algorithm would already have returned before the scan. In general, this holds and ensures that c_1 is always equal to m_i when t_β is updated because t_β marks the boundary such that any remaining α angles are included by updating t_α.

The correctness now follows from Proposition 2.2 and the standard sorting algorithm. Thus c_2 is the number of triangles that do not contain q when the scan is completed because each α angle is considered as root exactly once. The depth is correctly incremented because $\binom{n_{off}}{3}$ corresponds to the number of triangles that exist, ignoring points coinciding with q.

Finally, we need to argue for the equation that accounts for the points that

coincide with q. These points introduce three kinds of triangles where the first are triangles with one point coinciding with q, the second kind has two points coinciding with q and the third kind has three points coinciding with q, which is exactly what the equation calculates.

With respect to space and time complexity the algorithm computes the angles from q to any point in P and sorts them. This takes linear time plus the sorting bound $O(n \log n)$. The scan loops over $2n$ angles and does constant work for each angle, hence the total running time is bounded by $O(n \log n)$. Note that the binomial coefficient is only calculated for $\binom{k}{1}$, $\binom{k}{2}$ and $\binom{k}{3}$ ensuring that it takes constant time to compute. The algorithm clearly uses linear space as the n angles must be kept in memory at all times. □

3.2 Halfspace query depth algorithms

The halfspace depth of a point q is defined as the minimum number of points on one side of a halfspace going through q as formalized in Definition 1.2. An example of a query with depth 1 is depicted in Figure 3.4.

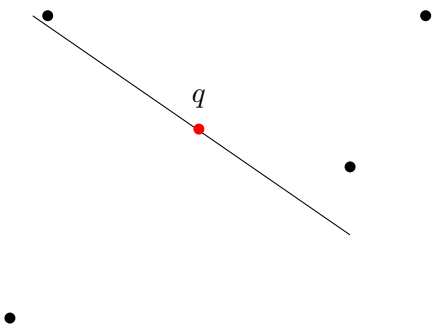


Figure 3.4: A query q with halfspace depth 1 using the halfspace.

The example give rise to a naive algorithm that creates all interesting halfspaces and counts the number of points on either side of the halfspace. The naive algorithm is denoted `QUERY_HS_NAIVE` and iteratively creates a halfspace between q and a point $p_i \in P$ and updates the depth as the minimum of the depth from the previous iteration and the minimum number of points on either side of the halfspace. Checking the sidedness of a point with respect to the halfspace is determined by equation 2.5. Special care is taken for points on the border of the halfspace, which is the case for p_i, p_j and p_k in Figure 3.5. They are considered to be on both sides, but the occurrences of points above and below q are also counted separately, e.g p_i and p_j are above and p_k is below. Points coinciding with q are not considered. When updating the depth, the number of points on the left side of the halfspace is subtracted by the maximum of the points above and below q that are on the border of the halfspace. The same thing goes for the number of points to the right side of the halfs-

pace. In the case where the halfspace is horizontal the points on the border of the halfspace is considered above when the x-coordinate is larger than q_x, and vice verse. Intuitively, this corresponds to rotating the halfspace as depicted in Figure 3.5. Finally, if p coincides with q the above routine is skipped and the depth is simply incremented.

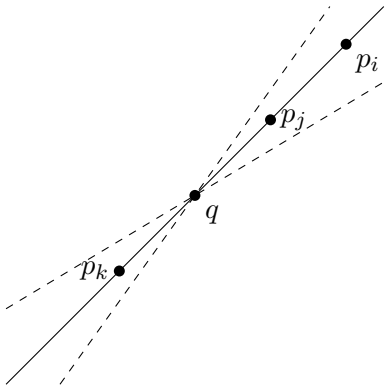


Figure 3.5: Rotating the halfspace going through q and p_i.

Theorem 3.3. Given a set $P \subset \mathbb{R}^2$ of size n and a query point $q \in \mathbb{R}^2$, `QUERY_HS_NAIVE` computes $\mathrm{D}_{hs}(P, q)$ in $O(n^2)$ time and uses $O(n)$ space.

Proof. Clearly, the interesting halfspaces are those going through q and each point $p_i \in P$, since it is the only place the depth can change. Let k_i be the number of points on the side of the halfspace going through q and p_i containing the minimum number of points. The halfspace depth of q is then found by minimizing k_i. Without loss of generality we look at a particular point p_i and seek to determine k_i. Disregarding the points that coincide with q, let l be the number of points that are to the left or on the boundary of the halfspace, r be the number of points that are to the right or on the boundary of the halfspace, a be the number of points on the boundary of the halfspace on one side of q and b be the number of points on the boundary of the halfspace on the other side of q. Note that the halfspace can be rotated an infinitely small amount in both directions, such that all points are either to the left or right of the halfspace. Thus, it is possible to subtract either a or b from l and symmetrically for r. The minimum number of points k_i is then given by

$$k_i = \min(l - \max(a, b), r - \max(a, b)).$$

The points coinciding with q are ignored. They will inevitable be included in the halfspace and are simply added to the final depth. The correctness now follows from equation 2.5.

The query point q is connected to n points and computes order n instructions each time. Thus, the running time is $O(n^2)$. The space complexity is clearly linear because the input has to be stored in memory at all times. □

As for the simplicial depth, the naive algorithm is simple to implement but the running time makes it useless in practice even though it is a factor

of n faster. The improved algorithm is also inspired by Rousseeuw et al. in [A18] and is denoted `QUERY_HS_SORT`. Initially, all points $p_i \in P$ are sorted in ascending order given by equation 2.6. Points coinciding with q are skipped and added to the depth. If there exists a gap strictly larger than π between any two consecutive angles, the algorithm returns the calculated depth. Otherwise the smallest angle is subtracted from all other angles and the first occurring angle α_k strictly larger than π is remembered.

The remaining part of the algorithm consists of two scans: One scan through the α angles with the β angles intertwined and another scan solely with the α angles. The β angles in the first scan are defined in the same way as in the description of `QUERY_SIM_SORT`. During the first scan, two pointers t_α and t_β are maintained together with an array F and a counter c_1. The counter c_1 denotes the number of angles strictly less than $t_\alpha + \pi$ meaning that the value of c_1 can be at most $2n$. The i'th entry of F denotes the value of the counter for α_i. Initially $c_1 = n_{off}$ where n_{off} is the number of points not coinciding with q. The setup before the first scan begins is exemplified in Figure 3.6a.

If $t_\alpha = \alpha_i < \beta_j = t_\beta$ then c_1 is incremented and t_α is updated to point to α_{i+1}. Otherwise the counter c_1 is written into F at index j and t_β is updated to point to β_{j+1}. In the case where t_β is updated from β_{n-1} to β_0 the value n_{off} is subtracted from c_1 after writing into F. A state during the first scan is exemplified in Figure 3.6b.

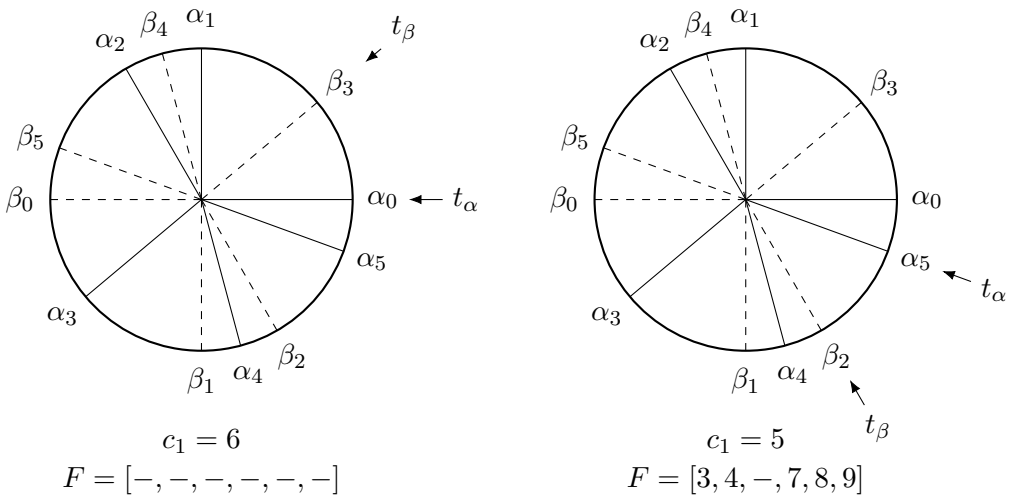
The second scan uses the information stored in F by scanning through the α angles and maintaining a counter c_2 that denotes the number of angles strictly less than α_i in order to minimize

$$d = \min_i \{\min(F_i - c_2, n_{off} - F_i - c_2)\}. \tag{3.1}$$

Initially the minimum value is $d = \min(F_0, n_{off} - F_0)$. If α_i coincides with the previous angle α_{i-1} the counter c_2 is left untouched. In this case a separate counter c_3 is incremented to keep track of the number of coinciding angles. Otherwise c_2 is incremented once, plus the potential number of angles saved in c_3. When the second scan is complete the depth is incremented by d and returned. The correctness and complexity is proven below.

Theorem 3.4. Given a set $P \subset \mathbb{R}^2$ of size n and a query point $q \in \mathbb{R}^2$, `QUERY_HS_SORT` computes $\mathrm{D}_{hs}(P, q)$ in $O(n \log n)$ time and uses $O(n)$ space.

Proof. We need to prove that computing equation 3.1 results in the correct halfspace depth and that the minimization is solved correctly by the algorithm. We ignore the points coinciding with q, because each such point clearly increases the depth by one. By definition, the halfspace depth is the minimum number of points on one side of a halfspace going through q, meaning that both sides of the halfspace needs to be considered which is what the min operator ensures. We just need need to argue that $F_i - c_2$ equals the number of points on one side of the halfspace going through q and a point $p_i \in P$. Let $F_i = |\{j : 0 \leq \alpha_j < \alpha_i + \pi\}|$ and $G_i = |\{j : 0 \leq \alpha_j < \alpha_i\}|$. The value $F_i - G_i = |\{j : \alpha_i \leq \alpha_j < \alpha_i + \pi\}|$ is equal to the number of points on one side of the halfspace and is what the algorithm computes since c_1 takes the value of F_i when writing to the array,



(a) The setup before the first scan.

(b) During the scan.

Figure 3.6: Illustrations during the first scan of `QUERY_HS_SORT`.

following a similar analysis as for c_1 in the proof of Theorem 3.2, and c_2 takes the value of G_i when doing the minimization.

The minimization is performed once for each $p_i \in P$ where the border of the halfspace is on one side of p_i. The case where the border of the halfspace is on the other side of p_i is considered when handling points on the opposing side of the unit circle. Furthermore, introducing c_3 ensures that coinciding α angles are always considered to be on the same side of the halfspace, since c_2 is not incremented until all coinciding α angles have been processed. The correctness now follows from Proposition 2.2 and the standard sorting algorithm.

With respect to space and time complexity, the algorithm computes the angles from q to any point in P and sorts them. This takes linear time plus the sorting bound $O(n \log n)$. The first scan loops over $2n$ angles and the second scan loops over n angles and both do constant work for each angle, hence the total running time is $O(n \log n)$. The algorithm clearly uses linear space as the n angles and F must be kept in memory at all times. □

3.3 Testing and robustness

In addition to the correctness proofs of the query depth algorithms we also implemented an extensive test suite which asserts the correct result on different kinds of degenerate and non-degenerate inputs. The algorithms are also compared pairwise on randomly generated input, though they sometimes proves to disagree due to the robustness issues described in Section 2.6. The problem is that equations 2.5 and equation 2.6 are biased towards different kinds of floating point errors. In an attempt to remedy this, all algorithms are generic such that the data type can be replaced by rational numbers or floating point num-

bers with more precision provided by the multiple precision arithmetic library GMP. Unfortunately, the angle computation does not apply for rational numbers because the square root and the inverse sine and cosine functions are not supported in the library. The algorithms are therefore compared using doubles and floating point numbers from GMP with very large precision. This introduces another problem because the implementation uses an approximation of π with a finite number of decimals which can cause errors. Extensions of GMP introduce functions that can calculate π with many decimals which could be used instead. We considered this out of scope and settled with comparing the algorithms using doubles and floating point numbers from GMP - a setting in which we have yet to see the algorithms fail in.

3.4 Efficiency of calculating the query depth

In terms of running time, the improved algorithms should in theory outperform the naive algorithms. In Table 3.1 we present a selection of measured running times for each of the four algorithms to get an impression of the gained speed. The input consists of n normal distributed random points generated with standard mean 0 and variance 500 where the query point is located at $(0, 0)$. It should be mentioned that the random number generator is seeded with the same number across all algorithms making it possible to compare the result. This is the case for all experiments in this chapter.

n	QUERY_HS_NAIVE	QUERY_HS_SORT	QUERY_SIM_NAIVE	QUERY_SIM_SORT
4	0.0016	0.0004	0.0030	0.0004
8	0.0032	0.0009	0.0263	0.0008
16	0.0101	0.0019	0.2307	0.0018
32	0.0309	0.0037	1.2869	0.0035
64	0.1988	0.0075	7.0981	0.0071
128	0.2274	0.0146	48.8211	0.0144
256	1.6356	0.0303	374.3440	0.0296
2^{20}		174.0910		170.8180

Table 3.1: The query time in ms when calculating the depth of a query point in the center.

It is clear that the sort bounded versions are much faster than their counterparts, justifying the relative complexity their implementations demand. The constants also appear to be relatively low given that it takes around 170 ms to compute the depth of 2^{20} points. The naive algorithms are several orders of magnitude slower. In fact, QUERY_SIM_NAIVE made it infeasible to run the experiment for larger inputs than shown in Table 3.1, making it completely useless in a practical setting. The only obvious use of the naive algorithms

is that they serve as a nice tool for asserting the correct result for the more complicated algorithms.

Though we already proved the complexity of the sort bounded algorithms, the next experiment gives empirical evidence that the bound actually holds in practice. The experiment is executed in the same setting as the timings in Table 3.1 and Figure 3.7 shows the results.

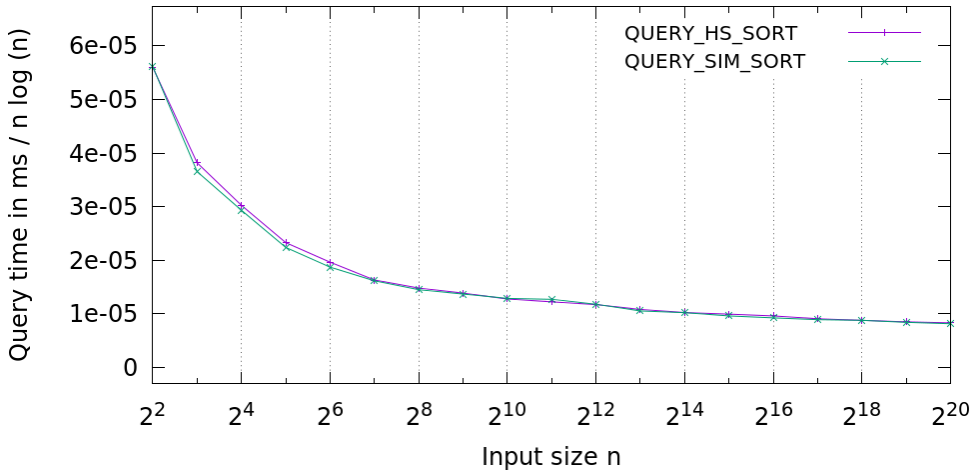


Figure 3.7: The query time in ms when calculating the depth of a query point in the center.

The measured running time is divided by $n \log n$ in order to emphasize that the algorithms run in $O(n \log n)$, because both algorithms converges towards a constant. There seems to be no difference between the two algorithms, meaning that the extra scan in `QUERY_HS_SORT` has a negligible impact on the overall running time, which is bounded by the standard sorting algorithm. The experiment also displays the trivial fact that the running time increases as the input gets bigger. In conclusion, the presented experiments of the efficiency is in line with what is expected and supports the theory.

3.5 Approximating the query depth

Computing the depth of a point in $O(n \log n)$ time will turn the process of ranking a set of size n into an $O(n^2 \log n)$ algorithm. Such a task will be very time consuming for large inputs, but Lemma 2.2 offers hope for improving the running time by ensuring an upper bound on the subset of Definition 2.3. In other words, it should be possible to improve the running time by uniformly sampling a random subset of the input without altering the result much. This section contains experiments that examine the effects of sampling with respect to the accuracy of the depth. Note that the algorithms are sampling the input points uniformly throughout this section.

3.5.1 A query in the center

The first experiment illustrates the effects of sampling the input and calculating the depth of a point close to the center. The input P consists of $n = 2^{20}$ normal distributed random points generated with standard mean 0 and variance 500 where the query point is located at $(0, 0)$. For each sample, multiple repeats are performed and the average depth over all repeats is calculated. The normalized error is obtained by rescaling the average depth based on the sampling probability, subtracting it from the true depth at full sampling size n and dividing it with the true depth. The rescaling factor is computed using Definition 2.3 by isolating the true depth $|R|$ in the following manner

$$\begin{aligned}
\left|\frac{|R|}{|X|} - \frac{|Y \cap R|}{|Y|}\right| &\leq \epsilon \\
-\epsilon \leq \frac{|R|}{|X|} - \frac{|Y \cap R|}{|Y|} &\leq \epsilon \\
-\epsilon|X| + \frac{|Y \cap R| \cdot |X|}{|Y|} \leq |R| &\leq \epsilon|X| + \frac{|Y \cap R| \cdot |X|}{|Y|}.
\end{aligned}$$

Ignoring the error terms the approximation becomes

$$|R| \approx \frac{|Y \cap R| \cdot |X|}{|Y|}$$

where $|Y \cap R|$ denotes the depth and $\frac{|X|}{|Y|}$ denotes the rescaling factor. When approximating the halfspace depth we are given a query point q and a subset $Q \subseteq P$ of size k and compute $\mathrm{D}_{hs}(Q, q)$. In this case $|X| = n$ and $|Y| = k$ meaning that the calculated depth $\mathrm{D}_{hs}(Q, q)$ is multiplied with $\frac{n}{k}$ in order to approximate $\mathrm{D}_{hs}(P, q)$. For the simplicial depth $\mathrm{D}_{sim}(Q, q)$ we have that $|X| = \binom{n}{3}$ and $|Y| = \binom{k}{3}$ resulting in a rescaling factor of $\frac{\binom{n}{3}}{\binom{k}{3}}$. The result of this experiment is shown in Figure 3.8.

Based on Lemma 2.2 we expect the error to decrease rapidly given that the function $\frac{1}{\epsilon^2} \log \frac{1}{\epsilon}$ also decreases rapidly, which is evident from Figure 3.8. Both algorithms quickly stabilize towards a small normalized error because the approximated depth does not deviate much from the actual depth, thus the technique works really well for points deep inside the data cloud.

An interesting observation from Figure 3.8 is that the simplicial depth has an error of 1 when the sampling probability is 2^{-18}, corresponding to 4 points. This follows from two facts. Firstly, the maximal simplicial depth converges towards $\frac{1}{4}$ of the total number of possible triangles. The intuition behind this fact can be seen by looking at the implementation of `QUERY_SIM_SORT`. The main insight is that the input points and their angles will be distributed evenly around the point in the center. Hence, any root α_i angle will get $m_i = \frac{n}{2}$ during the execution of the algorithm. Each of the n angles can therfore form $\binom{\frac{n}{2}}{2}$ triangles that does not contain q which corresponds to a factor of

$$\frac{n\binom{\frac{n}{2}}{2}}{\binom{n}{3}} = 0.75.$$

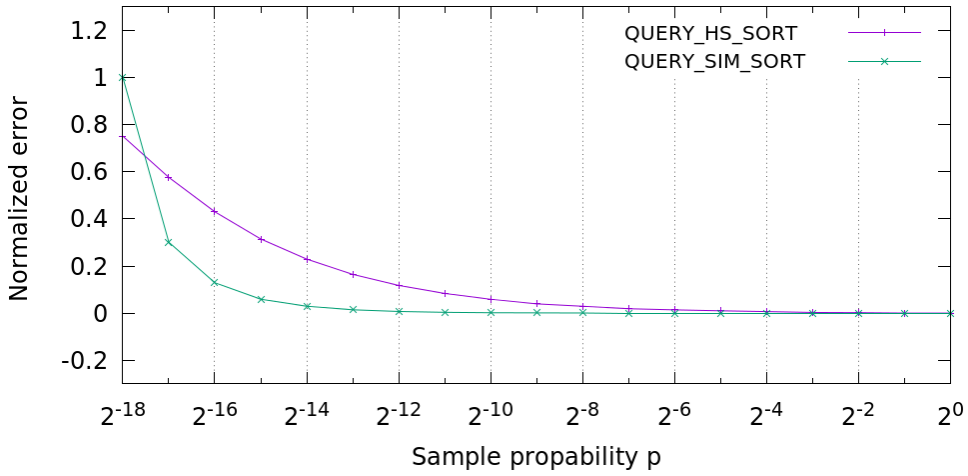


Figure 3.8: The normalized error when approximating the depth of a query point in the center.

The second fact is that the simplicial depth of 4 points will be either 0, 2 or 4. Clearly, a depth of 0 is always possible. A depth of 2 is possible since for any point inside a single triangle spanned by 3 of the 4 points, it must also be inside at least one more triangle formed by substituting one of the 3 points with the fourth. Clearly, the edge on the opposing side of the fourth point can be used to generate a new triangle also containing the query point. Lastly, a depth of 4 is possible if the query point lies on the intersection of all the 4 possible triangles, which is highly unlikely.

With these two facts in hand, it is now possible to explain the error of 1 for the sampling probability of 2^{-18}. The problem is that the true depth is equal to the rescaling factor. To clarify, the true depth is approximately $0.25 \cdot \binom{2^{20}}{3}$ by the first fact and since $\binom{4}{3} = 4$ the rescaling factor also becomes $\frac{\binom{2^{20}}{3}}{4} = 0.25 \cdot \binom{2^{20}}{3}$. The approximated depth is 0, 2 or the highly unlikely 4 by the second fact, hence the corresponding error after rescaling becomes 1 because the approximation is either zero or two times the the real depth.

3.5.2 A query near an outlier

Given that the goal is to rank data points and mark potential outliers it makes sense to see how the approximation approach behaves for outliers, hence the second experiment attempts to show this effect. The setup is almost identical to the first experiment and the only difference is that the query point is now positioned very close to an outlier. The depth is still calculated by random sampling and rescaling depending on the sampling probability. The results of the second experiment is depicted in Figure 3.9.

Figure 3.9 illustrates two main problems with the approximation approach when used on a point near an outlier. The first problem is that only a few input points actually affect the depth calculation. This means that whenever

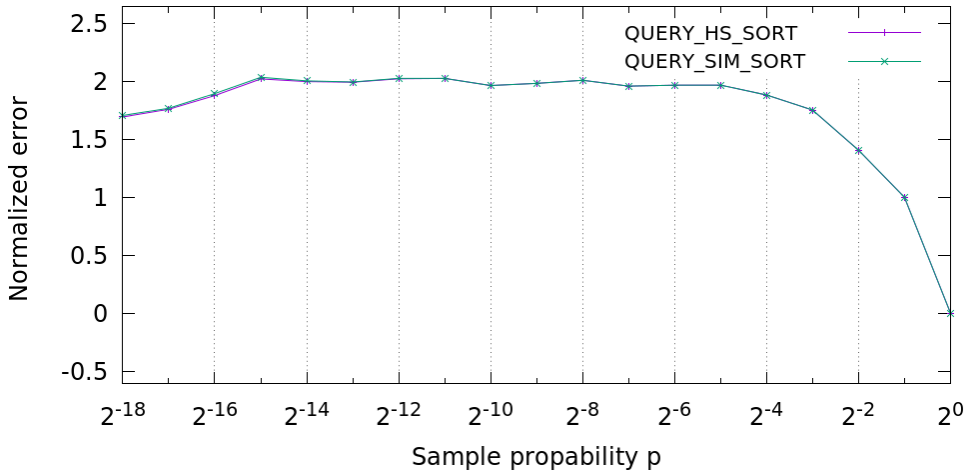


Figure 3.9: The normalized error when approximating the depth of a query point near an outlier.

the outlier is not sampled the depth will simply be zero. The second problem is caused by the rescaling performed on the sampled depth. The rescaling assumes that the random sample represents the true distribution of the entire input P, meaning that the calculated depth represents the true relative depth of the query point. For example, given a point with a true halfspace depth of 1 caused by a given outlier p_i, if all points except p_i are included in the sample the depth is zero. Otherwise, if p_i is included in the sample the depth will be scaled by the scaling factor, resulting in a depth larger than 1. Thus, for small sample sizes there is a very high probability that p_i is not sampled and the normalized error is 1. On the other hand, if p_i is sampled the normalized error will be huge since the small sampling probability will scale the result by a large factor.

Even though Figure 3.9 displays the normalized error, the curves converges towards an error of 2 for small sampling probabilities which is a consequence of how the sampling probability effects the rescaling. Take the halfspace depth for example, assuming a single outlier p_i affects the depth calculation and the sampling probability is s, then with probability s, p_i is sampled and the approximated depth is $\frac{1}{s}$. Thus the normalized error is

$$\frac{\frac{1}{s} - 1}{1} = \frac{1}{s} - 1.$$

Likewise, with probability $1 - s$ the depth is 0 and the normalized error is 1. Suppose S is the random variable taking the value $\frac{1}{s} - 1$ with probability s and the value 1 with probability $1 - s$, it is now possible to calculate the expected value of S

$$E[S] = s \cdot \left(\frac{1}{s} - 1\right) + (1 - s) \cdot 1 = 1 - s + 1 - s = 2 - 2s.$$

Applying this to the experiment with a sampling probability of 2^{-10} we get an expected normalized error of $2 - 2 \cdot 2^{-10} \approx 2$. This means the halfspace depth

error will be 2 most of the time as long as the sampling probability is low. A similar argument can be made for the simplicial depth error, the only difference is the normalized error calculated once p_i is sampled. It gives rise to a more complex equation since the approximated depth does not solely depend on the sampling probability, i.e. adding more points will always increase the depth.

3.5.3 Approximations on average

Based on the result from the previous experiment the approximation approach does not seem to be particularly good at estimating the depth on outliers because the error is relatively large. However, the first experiment shows the opposite for query points near the center. This raises a natural questions: Is the approximation approach viable at all?

The focus of the third experiment is to answer this question. The setup again consists of 2^{20} normal distributed random points with mean 0 and standard deviation 500. Unlike the first and second experiments, this experiment does not use a fixed query point, but picks a new random query point a number of times. To elaborate, the experiment repeats the depth calculation for a query point with respect to a given sample size, then picks a new random query point and does the same. This is repeated a number of times before moving on to the subsequent sample size. The experiment uses the same normalized error calculation as the two first experiments. Worth noting is that the random query point is generated using the same distribution as the input points. Thus, the objective of the third experiment it to see the effects of the approximation approach on average.

The first experiment showed that the approximation approach for points close to the center achieves good results. The second experiment showed the opposite for outliers. The problem with points at low depth is that the true depth can almost be calculated using a sample which causes the rescaling to overshoot the real depth. But the closer the point is to center the less this problem occurs. Furthermore, assuming the input data contains a natural center, points at high depth will be more likely than points a low depth. Based on this, it makes sense to expect the approximation approach to produce decent results on average for random input data, assuming the query points follow the input distribution. The results of the third experiment is shown in Figure 3.10.

Indeed this is evident in Figure 3.10. Although the algorithms converge slower than in the first experiment, they clearly converge towards a very low normalized error. Based on this there is some merit in using a random sampling approach to approximate the real depth of a query point.

Given the error on average we also try to verify the theory by inserting into Lemma 2.2. We know that the VC-dimension of the simplicial depth problem is bounded by 8. Consider a sampling probability of 2^{-12} and observe that the experiment measures a normalized error of 28.3851% for the simplicial depth. Using Lemma 2.2 the expected theoretical upper bound on the sampling size is

$$\frac{8}{0.283851^2} \log \frac{8}{0.283851} \approx 478.$$

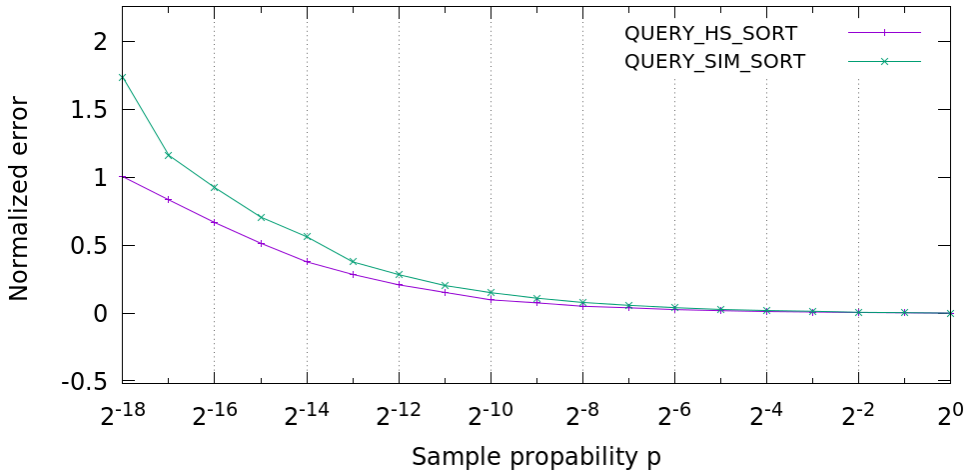


Figure 3.10: The normalized error when approximating the depth of multiple random query points.

A sampling probability of 2^{-12} corresponds to sampling 256 points, thus the experimental result is well within the theoretical upper bound. Since the difference between the two sampling sizes is relative high, this does indicate that the true VC-dimension of the simplicial depth problem is smaller and a tighter bound may exist. For completeness, we make the same computation for the halfspace depth problem where theory reveals that the VC-dimesion is 3 or less. For the sampling probability of 2^{-12} the experiment measures a normalized error of 20.88% resulting in the upper bound

$$\frac{3}{0.2088^2}\log\frac{3}{0.2088} \approx 264.$$

This proves to be a much tighter bound than for simplicial depth since the difference is less than 10 points and emphasizes that the theory is sound.

Unfortunately, there is a problem with the simplicial depth and our method of sampling. The approximation is based on the fact that any subset $Y \subseteq X$ of the range space $(X, \mathcal{R})$ can be generated as a possible sample. This is the case for the halfspace depth since X consists of points and the algorithm samples points. For the simplicial depth however, X consists of triangles but the algorithm samples points. Thus, the algorithm cannot sample X independently and some subsets are impossible to create. The most obvious example of an impossible sample is two disjoint triangles. Such a pair cannot be generated by sampling points, since the triangles generated by the points will always be connected. Hence, the question is how these two sampling approaches relate?

The fourth experiment seeks to answer this question. The setup is the same as in the previous experiment except in this experiment `QUERY_SIM_SORT` is compared to a new sampling algorithm that actually samples uniformly from all possible input triangles. One thing worth noting is that the uniform sampling algorithm's running time is $O(n^3)$ since n points can generate $\binom{n}{3}$ possible triangles. Because of this, the fourth experiment is not executed for all sampling

probabilities, it simply takes too much time. The results of the experiment can be seen in Figure 3.11

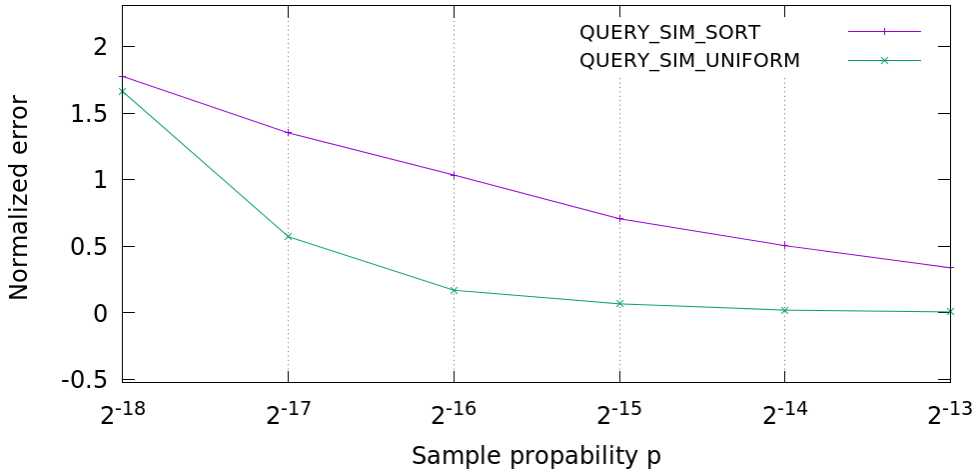


Figure 3.11: The normalized error when approximating the depth of multiple random query points.

From Figure 3.11 it is evident that approximating by uniformly sampling from all possible triangles is better than our standard sampling approach for simplicial depth. However, approximating by sampling points is within the theoretic bound and not much worse, leading to the conclusion that approximating by sampling points is decent even though it does not follow directly from the theory.

3.5.4 Ranking points in practice

Seeing that approximating the depth can be calculated with a tolerable error, it is interesting to see how the depth measure works for ranking points in practice. Specifically, we want to see if outlier detection is possible while not sampling all points? In order to answer this question, a slightly different depth calculation is needed. The current problem is that random sampling and rescaling is only viable for outliers if the sample probability is very high. The solution is to approximate the depth relative to the maximal possible depth, i.e. the depth is normalized. The only change needed for the approximation to work is to divide the approximate depth with the theoretical max depth. Thus the new depth approximation for halfspace is given by

$$\mathrm{D}_{hs}(Q,q) \cdot \frac{n}{k} \cdot \frac{1}{\lceil \frac{n}{2} \rceil}.$$

For simplicial depth the approximation becomes

$$\mathrm{D}_{sim}(Q,q) \cdot \frac{\binom{n}{3}}{\binom{k}{3}} \cdot \frac{1}{\binom{n}{3}}.$$

The final experiment illustrates the effects of this approach. The experiment consists of rerunning the first and second experiment using the new depth approximation. Instead of calculating the normalized error this experiment calculates the average normalized depth of the query point for a given sample. The results can be seen in Figure 3.12 and Figure 3.13.

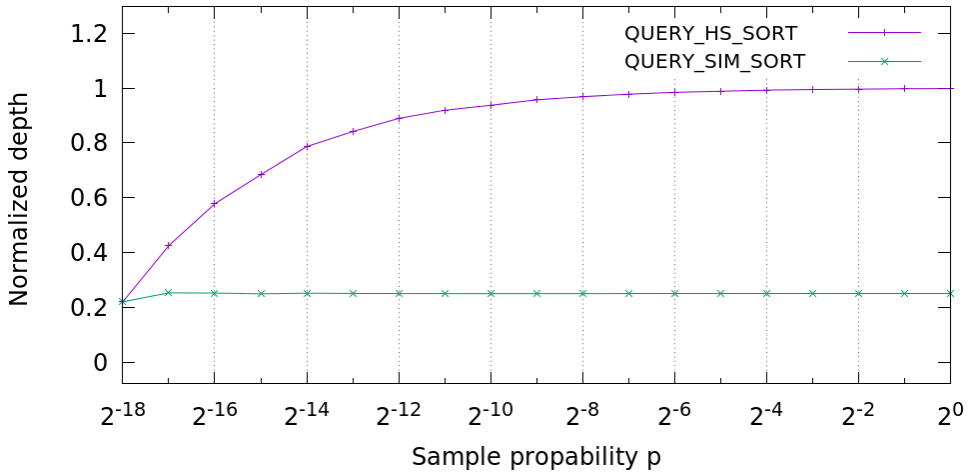


Figure 3.12: The approximate normalized depth of a query point in the center.

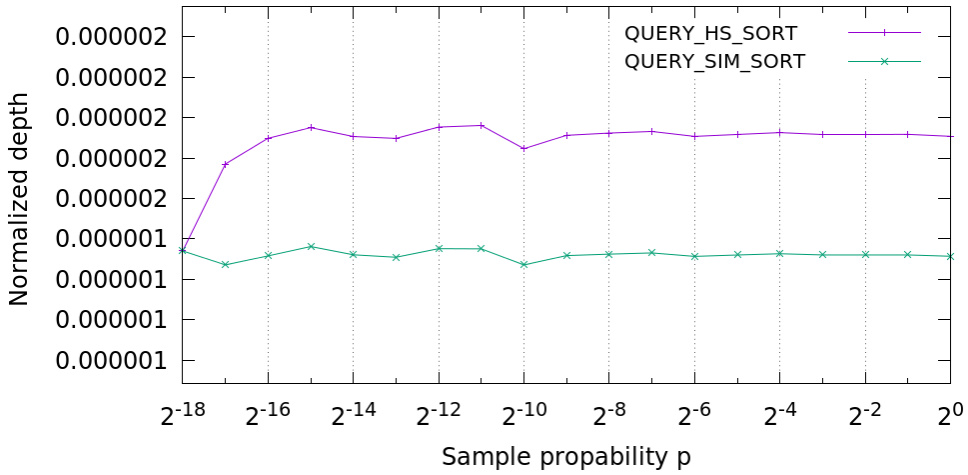


Figure 3.13: The approximate normalized depth of a query point near an outlier.

It is evident from Figure 3.12 that the normalized depth of a query point close to max depth quickly stabilizes at a high value. Likewise, it is evident from Figure 3.13 that the relative depth of an outlier is constantly very low. In conclusion, we have established that outlier detection is indeed possible and reliable when uniformly sampling the input, hence the process of ranking an entire set of n points can be done within a tolerable error in $O(nk \log k)$ where k is the sample size and $k << n$.

Note that the experiments are conducted on normal distributed data. Therefore, taking uniform random samples can pose technical difficulties when applied to specific types of input as presented by Afshani in [A14], where the simplicial depth of a point deep in the data cloud has a large approximation error. We will not pursue this problem further, but merely make it clear that this simple method has certain disadvantages.

3.6 Conclusion

In this chapter we sought to describe, implement, analyze and experiment with four different algorithms for ranking a point relative to the halfspace and simplicial depth measures, in order to reason about efficiency and approximations. The naive implementations are easy to implement but prove useless in practice due to their running time. More efficient algorithms can be achieved by sorting the input and utilizing the order of the points such that the halfspace and simplicial depth can be computed in $O(n \log n)$ time. Though ranking is relatively fast for a few points, the process of ranking a set of size n becomes an $O(n^2 \log n)$ algorithm which may be infeasible for large inputs. As a solution, approximate results can be achieved by uniformly sampling the input with a tolerable error. The approximation error of outlying points are significantly larger than points deep in the data cloud, but the error is very low on average even for a very small sample. In practice the approximation method works very well and can improve the algorithm for ranking a set of n points to $O(nk \log k)$ where $k << n$. The approximation can be performed with respect to both depth measures and the results is consistent with the theory on the subject.

Chapter 4

Arrangements

In this chapter we seek to describe and experiment with a versatile, efficient and robust implementation of an arrangement of lines in $\mathbb{R}^2$. The main motivation for this study is that the halfspace median may be calculated efficiently using a geometric divide and conquer procedure in dual space. For that purpose we need to split the plane into cells by cutting the input into triangles, which requires a robust and efficient arrangement algorithm. Note that the algorithm does not handle vertical lines as the input is a dualization of points, in which vertical lines cannot occur. The algorithm is tedious to implement, explain and imposes a lot of special case handling, hence it is presented in Section 4.1 through Section 4.5 for the sake of understanding. In Section 4.6 we present a brief description of the tests performed and some considerations about robustness. The main focus of the experiments presented in Section 4.7 and Section 4.8 is to verify the theoretic space and time complexity of the arrangement in order to emphasize the correctness of the algorithm. The chapter is concluded in Section 4.9.

4.1 The bounded arrangement

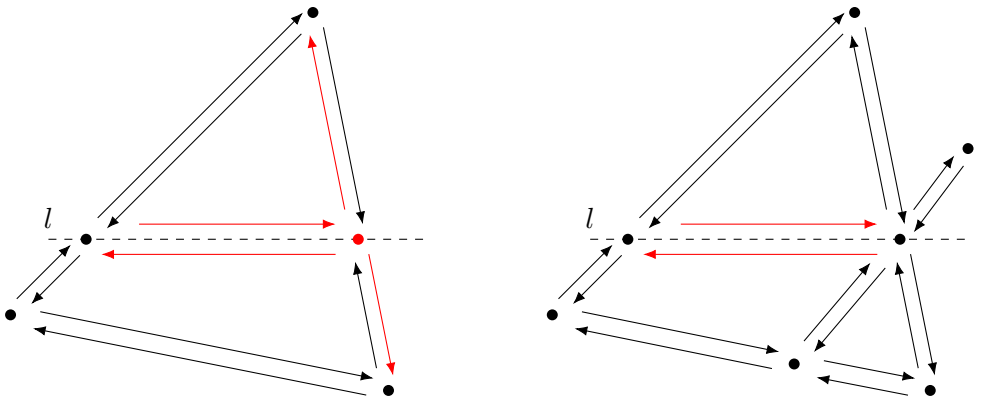
The algorithm for constructing an arrangement of lines is incremental and based on a doubly-connected edge list as described in Section 2.1. The arrangement implemented in the source code extends the arrangement presented in this section and it is slightly more complicated. We will return to these extension in Section 4.3 and Section 4.5.

The algorithm is denoted `ARRANGEMENT` and starts out by computing a bounding box of the input lines L in $\mathbb{R}^2$. This is done by comparing all pairs of lines and updating two points, i.e. the point in the top left corner with the maximum y-coordinate and minimum x-coordinate and the point in the bottom right corner with the minimum y-coordinate and maximum x-coordinate. The lines are inserted one after another by finding the leftmost intersection with the bounding box and traversing the interior of the arrangement from left to right. Finding the next intersection is done by traversing the current face using the next pointers of the edges. Intersections are always calculated between the line that is inserted $l_1 = (a_1, b_1)$ and the segment s spanned by two vertices $v_1 = (x_1, y_1)$ and $v_2 = (x_2, y_2)$, which is done by determining the line

$l_2 = (a_2, b_2)$ going through v_1 and v_2 and computing the intersection

$$\begin{aligned} x &= \frac{b_2 - b_1}{a_1 - a_2} \\ y &= a_1 \cdot x + b_1. \end{aligned} \tag{4.1}$$

The line l_1 intersects the segment s if and only if $x \in [\min(x_1, x_2); \max(x_1, x_2)]$ and $y \in [\min(y_1, y_2); \max(y_1, y_2)]$. There are two intersect events to consider: Either the line intersects an edge e or a vertex v. In the first case a new vertex is created on the intersection point along with four edges. The first two edges are inserted where the existing edge e is split. The remaining two are crossing edges that are connected to the vertex from the previous event. This event is depicted in Figure 4.1a where the red edges and red vertex are created. In the second case, the line intersects a vertex and the faces around the vertex are searched to find the one where the line continues in. This event is depicted in Figure 4.1b where the two red edges are created. No vertices are created in this case and crossing edges are added whenever the line is not coinciding with an existing edge, which happens if there exists duplicate lines. The first intersection with the bounding box is a special case as no crossing edges are created. The algorithm terminates when the outer side of the bounding box is reached. An example of a bounded arrangement can be seen in Figure 4.8 where the bounding box is substituted with at bounding triangle. The reason for changing the bounding object will be explained later.



(a) Handling an edge intersect event.

(b) Handling a vertex intersect event.

Figure 4.1: Illustrations of handling intersect events.

Theorem 4.1. Given a set L of n lines in general position in $\mathbb{R}^2$, `ARRANGEMENT` computes an arrangement of lines in $O(n^2)$ time and uses $O(n^2)$ space.

Proof. We need to prove that the doubly-connected edge list represents the subdivision induced by L. Clearly there is no need to represent the subdivision outside of the smallest bounding box having all intersections in its interior, thus we need to shown that this interior is generated correctly.

Without loss of generality we look at a single line l_i and let $A_{i-1}(L)$ denote the doubly-connected edge list before inserting l_i. Observe that the edge

list is updated correctly if all intersect events between l_i and $l_1, \ldots, l_{i-1}$ are encountered during the insertion of l_i because these events are clearly handled correctly as illustrated by Figure 4.1a and Figure 4.1b. The subsequent face that l_i continues in after an intersect event, is traversed in the counter-clockwise direction using the next pointers of the edges. By convexity of faces, this will correctly determine the next intersection given that $A_{i-1}(L)$ is generated correctly, because a line enters and leaves a face in exactly two unique points. Thus all intersection events are encountered exactly once and $A_k(L)$ is correct. Note that there may be missing intersections between l_i and $l_{i+1}, \ldots, l_n$ that needs to be added to the edge list. These will be added by the same argument when $l_{i+1}, \ldots, l_n$ are inserted. Hence, $A_n(L)$ represents the subdivision induced by L.

Computing the bounding box takes $O(n^2)$ times because all intersections between all pairs of lines are calculated. When inserting l_i the algorithm does work proportional to the complexity of the zone associated with the line, which is $O(i)$ by Theorem 2.2, and the time spend inserting all lines is bounded by

$$\sum_{i=1}^{n} O(i) = O\left(\frac{n(n+1)}{2}\right) = O(n^2).$$

Hence the total running time is $O(n^2)$. With respect to space complexity Theorem 2.1 tells us that there exists $O(n^2)$ edges and $O(n^2)$ vertices but it does not account for the the records on the bounding box. Fortunately, there are at most $2n = O(n)$ edges and vertices on the bounding box so the bound still holds. Finally, the algorithm creates the double amount of edges, but that has no influence asymptotically, thus the total space used is $O(n^2)$. □

4.2 Details of the bounded arrangement

The are numerous implementation details when implementing the arrangement. One such detail is how to determine what face the line continues in when handling a vertex intersect event. This is handled by evaluating a point on the line and checking whether it is inside the faces incident to the vertex. The face that contains the point is the face the line continues in. Note that the algorithm traverses the arrangement from left to right, so this point always has a larger x-coordinate than the vertex of the intersect event. Performing the check reduces to doing two sidedness tests using equation 2.5 because the point must be to the left of the lines spanned by the edges bounding the face. During implementation we discovered that this can cause precision error because the line may intersect vertices that are not exactly on the line because the intersection routine recreates a line from two vertices. This is exemplified in Figure 4.2 where the line $l = (a, b)$ intersects vertex v due to numeric instability. If the algorithm aimlessly evaluates the point p the next face to consider is f_1 which has already been visited. The problem is solved by translating the point in the y-direction such that it looks like l actually goes through v. The distance that p is moved is determined by

$$v_y - (a \cdot v_x + b)$$

which is the distance between v and l in the y-direction.

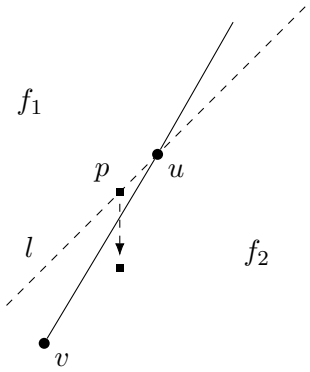


Figure 4.2: Translating the point p to determine the correct face.

Another interesting implementation detail when coding in a language without a garbage collector is how to avoid memory leaks. The data-structure itself uses a lot of memory, which we will return to in Section 4.7, and memory may become an issue when using it in a recursive algorithm, thus it needs to be freed. Freeing the memory obtained by a doubly-connected edge list requires that the edge list can be traversed. We do this by a standard breadth-first search using a queue of vertices. The root vertex is inserted into the queue and marked visited by setting a bit on the vertex. The search procedure iteratively removes a vertex from the queue, marks it visited and inserts all adjacent non visited vertices in the queue. The search terminates when the queue is empty. When a vertex is removed from the queue it is also inserted into a local list along with all its incident edges. This ensures that all vertices and edges are considered because the edge list is connected, meaning that all vertices can be reached from the root, and each edge has exactly on originating vertex. The vertices and edges are then deleted when the breadth-first search terminates. The drawback of this delete procedure is that we need store a pointer for each vertex and edge that forms the edge list. This cannot be avoided because deleting edges and vertices during the breadth-first search causes invalid reads of unallocated memory, e.g. when a vertex reads the visited bit of an adjacent deleted vertex or when incident edges of a vertex is located using deleted edges that originates from another vertex.

A final detail is that edges are equipped with a bit that marks whether it is on the inside or outside of the arrangement. This notion will be used extensively in the unbounded arrangement algorithm, but for now it is only used to determine when the outer side of the bounding box is reached.

4.3 The unbounded arrangement

As described in Section 4.1, the algorithm implemented in the source code is slightly more complicated than explained so far. One of its extensions is that the concept of the bounding box is changed, thus becoming an unbounded arrangement without increasing the complexity with respect to space and time.

The problem is that the arrangement is used to create a cutting in an algorithm that runs faster than $O(n^2)$ time. Creating a cutting is done by sampling $k \leq n$ lines, building the corresponding arrangement and triangulating it. For this to work using the bounded arrangement the bounding box has to be built in less than $O(n^2)$, because a bounding box of the sample does not provide a subdivision of the lines outside of the box. The current algorithm computes the bounding box in $O(n^2)$ by comparing all pairs of lines, but this could be done in $O(n \log n)$ time by sorting the lines on their slopes. Instead, our implementation is inspired by the idea of making a conceptual bounding box at infinity by constructing infinity vertices representing the input lines. This is more efficient because the lines does not need to be sorted. The concept is depicted in Figure 4.3 where the arrangement consists of the lines $l_1 = (a_1, b_1)$ and $l_2 = (a_2, b_2)$.

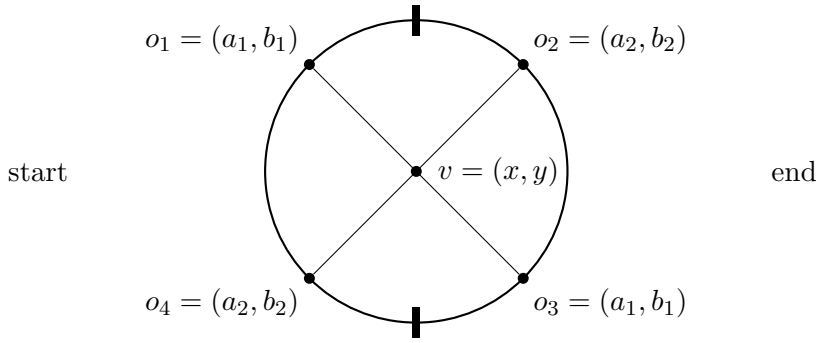


Figure 4.3: An arrangement with a conceptual bounding box at infinity.

Vertices and edges on the bounding box are referred to as outer vertices and outer edges. The edges that connects inner and outer vertices are also special in a sense and are referred to as hybrid edges. The lines are still inserted one after another by finding the leftmost intersection with the bounding box. The term intersection is a little misleading in this case because outer vertices contains line parameters, thus the starting point is determined by the parameters of the line. The outer vertices are sorted on these parameters such that vertices with a larger slope are below vertices with a smaller slope. This only holds for the left side of the arrangement, which is denoted start in Figure 4.3. The reverse sorting holds on the right side of the arrangement. Hence, the initial step of inserting the line $l_1 = (a_1, b_1)$ consists of finding the spot among outer vertices on the starting side, which is done by following the next or previous pointers of outer edges. When the search stops the algorithm handles the intersection as a normal edge intersect event without creating the crossing edges and marks the vertex as being outer. The parameters $o_s = (a_s, b_s)$ and $o_l = (a_l, b_l)$ from the two surrounding outer vertices are recorded, because they indicate where the line intersects the outer vertices on the ending side, just in reverse order. This stopping criteria becomes slightly more complicated when adding parallel lines, but we will return to that in Section 4.4.

In addition to the standard edge and vertex intersect events as described in Section 4.1, the line can now intersect hybrid edges and outer edges. Hybrid edges are easy because the parameters of the other line is given by the outer

vertex $o = (a_2, b_2)$ so equation 4.1 can be used to determine the intersection point directly. The intersection is handled as a normal edge intersect event. Outer edges are also easy to handle since the line intersects such an edge if and only if the two surrounding outer vertices have parameters equal to o_s and o_l. The intersection is also handled as a normal edge intersect event and the vertex is marked outer. With respect to vertex intersect events, the line can now intersect outer vertices, but that just means that a coinciding line has already been inserted so the algorithm can return safely. We will provide no formal proof, but the statement from Theorem 4.1 still holds for the this unbounded arrangement algorithm.

4.4 Details of the unbounded arrangement

As for the bounded arrangement there are many details to consider when implementing the algorithm. Firstly, the stopping criteria on the ending side of the arrangement is not necessarily determined by the two surrounding outer vertices $o_s = (a_s, b_s)$ and $o_l = (a_l, b_l)$ as described in Section 4.3 when there exists parallel lines. The problem is that parallel lines do not swap places inside the arrangement because they do not intersect each other which is depicted in Figure 4.4.

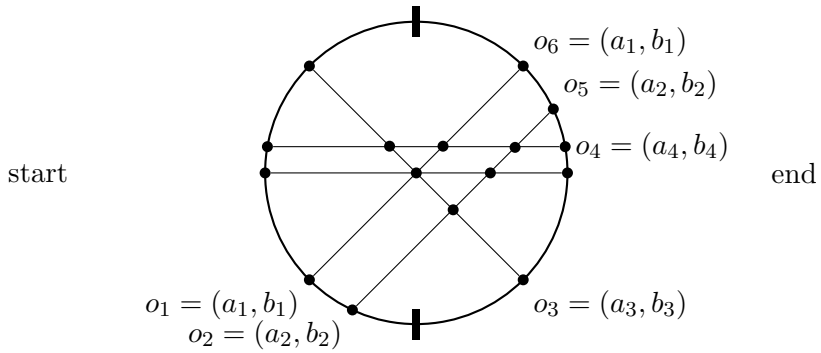


Figure 4.4: Handling the stopping criteria with parallel lines.

Take o_2 for example, the surrounding outer vertices on the starting side are $o_1 = (a_1, b_1)$ and $o_3 = (a_3, b_3)$, but the line ends between $o_6 = (a_1, b_1)$ and $o_4 = (a_4, b_4)$. In order to remedy this, the algorithm issues a search among the outer vertices on the ending side of the arrangement instead of recording the surrounding vertices on the starting side. This fixes the problem, but parallel lines actually impose another headache. Remember that the order of the outer vertices are reversed when going from the starting to the ending side, i.e. the vertex with the largest slope is lowest on the starting side and highest on the ending side. That order does not hold for parallel lines that are sorted on the intersection with the y-axis, which can be seen in Figure 4.4 where o_1 is above o_2 on the starting side and o_6 is above o_5 on the ending side. Consequently, the algorithm must issue two slightly different searches to determine the starting position and the stopping criteria. This does not change the bounds as both

searches takes $O(n)$ time in the worst case.

Another interesting implementation detail imposed by the unbounded arrangement occurs during a vertex intersect event. Recall from Section 4.4 that we needed to determine what face the line continues in, which reduced to two sidedness tests using equation 2.5. If one of the edges bounding the face is a hybrid edge, the algorithm needs to evaluate a point on the hybrid edge in order to use equation 2.5. It is not possible to use the outer vertex because it stores the line parameters that created the edge. The solution is however straightforward since the hybrid edge is connected to this outer vertex. The only question is whether the hybrid edge is bounded in the positive or negative direction with respect to the x-axis, because the point needs to be on the edge. In order to determine this, we added a bit on all outer vertices marking whether it belongs to the starting or ending side. If the outer vertex is on the starting side the point has a smaller x-coordinate than the vertex of the intersect event, otherwise the x-coordinate is larger.

4.5 The combined arrangement

The arrangements described so far are either bounded or unbounded, but the one implemented actually incorporates the best of both worlds. The combined arrangement can either be unbounded, semi-bounded or bounded. The reason is that the algorithms using the arrangement needs to build new arrangements inside existing faces that are shaped as triangles. This does not pose a problem for the bounded arrangement as the bounding object may just as well be a bounding triangle, the building procedure is identical. For this combined algorithm to work correctly, each line is checked against the current arrangement to see if it starts in an unbounded region, ends in an unbounded region, both or neither. Assuming that a line intersects the arrangement these cases are simple to determine. If the arrangement has zero outer vertices or nothing but outer vertices on the boundary, we are in one of the two last cases and the algorithm continues using the bounded or unbounded algorithm respectively. Otherwise, the arrangement is semi-bounded with two hybrid edges. In this case the line may intersect both hybrid edges and the algorithm continues using the bounded algorithm. If not, the line starts or ends in the unbounded region depending on the cases depicted in Figure 4.5. Note that there cannot be more than π radians between the interior of the semi-bounded arrangement because it is built from straight lines. Thus, each of these cases can be uniquely determined by checking whether the outer vertices belongs to the starting or ending region.

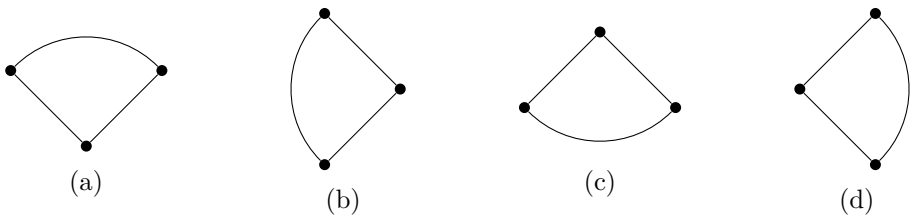


Figure 4.5: The four cases of a semi-bounded arrangement.

In case 4.5a the line starts in the unbounded region if it has a smaller slope than the leftmost hybrid edge. Similarly, the line ends in the unbounded region if it has a larger slope than the rightmost hybrid edge. The slopes are checked using the parameters saved in the associated outer vertices. The intersection with the y-axis is checked on equality. Clearly, the result of one slope test determines the other, which results in a minor optimization. Handling the remaining three cases is a simple case analysis and is left out.

These extensions do not affect the space and time bounds stated in Theorem 4.1 as the combined arrangement algorithm merely combines the procedures described in Section 4.1 and Section 4.3. For completeness, Figure 4.6 through Figure 4.8 shows examples of these three kinds of arrangements produces by the algorithm.

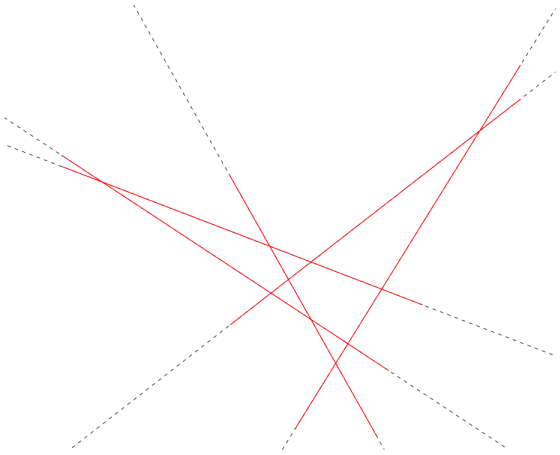


Figure 4.6: An example of an unbounded arrangement.

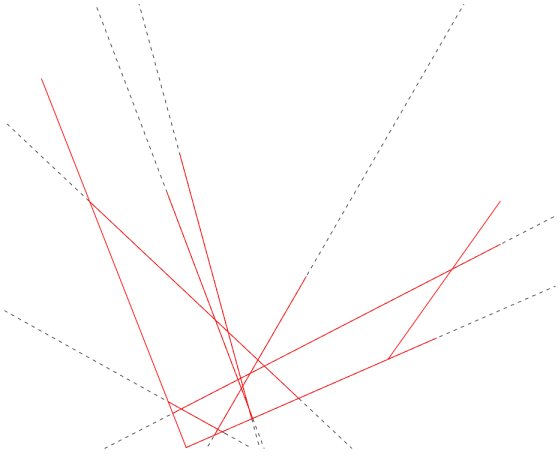


Figure 4.7: An example of a semi-bounded arrangement.

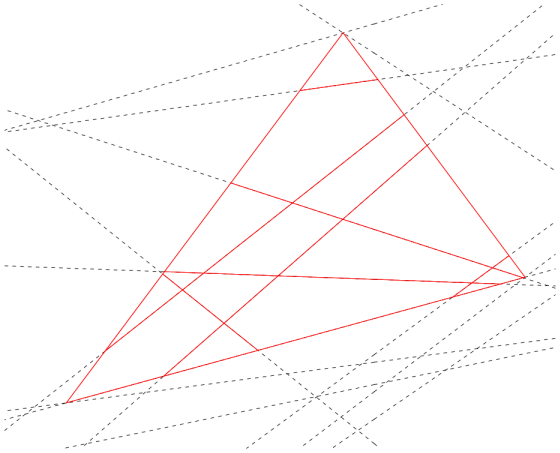


Figure 4.8: An example of a bounded arrangement.

4.6 Testing and robustness

Since the proof of correctness is purely theoretical it makes sense to test the implementation of the arrangement algorithm for errors. The implementation is not necessarily complex, but is very tedious because there are many pointers that needs to be set correctly and it is very easy to overlook small errors. Unlike the query depth algorithms there is no naive algorithm to compare up against, which means that all tests are performed either manually, by drawing the arrangement and looking at the edges and vertices, or automatically by checking that all pointers are set correctly. The automatic test repeatedly builds an arrangement on randomly generated input and verifies that the structure is correct. This is done by testing all local properties of the doubly-connected edge list. An example of such a property is that every edge should be the twin of its own twin. Though this is a good indication that the algorithm is correct, it is still difficult to truly verify that the edge list actually represents the arrangement induced by the set of lines.

In addition to the more complicated testing situation, numerical instability has an even worse effect on the arrangement algorithm. Rounding errors during construction can cause non-parallel lines to become parallel or make lines intersect vertices that they are just very close to. An example of this is already described in Section 4.2. Such degeneracies can in the best case result in a faulty arrangement, but in more critical situations the errors can cause the algorithm to enter a face that the line does not intersect and loop indefinitely. A solution to this problem is to detect infinite loops by keeping track of the visited records, but this is not desirable for an already memory craving data-structure. Alternatively, the algorithm may return prematurely if the line at hand makes more than k intersections where k is the number of lines inserted into the arrangement. This does not prevent the algorithm from looping inside a face that the line does not intersect, but other thresholds can be applied to escape such situations. Another approach that can be of use in applications

where precision is more important than speed is to use an arithmetic library with very high to infinite precision. Such libraries differ in performance, but to get an impression of the slowdown we tested the algorithm using the multiple precision arithmetic library `GMP`. The results can be seen in Table 4.1.

n	doubles	rationals
32	0.129	7.734
256	12.695	567.910
2048	1316.510	36 277.400

Table 4.1: The arrangement building time in ms for select values of n using double and rational number precision.

The table shows some building times using double and infinite precision using rational numbers and reveals that the slowdown is significant. For 2048 lines the algorithm suffers a factor 26 slowdown which is a lot taken into consideration that the construction takes more than 1 second when compiled with double precision. In order to improve this, one could analyze the code and determine which parts demand higher precision and which can manage with less and thereby utilize rational numbers when needed. In conclusion, the algorithm is compiled using double precision for improved efficiency.

4.7 Verification of the space complexity

Having described the arrangement algorithm, the focus of this section is to see how well the algorithm behaves compared to the theoretical space complexity. From Theorem 2.1 we know the exact number of edges and vertices used in an arrangement induced by a set of n lines. Table 4.2 presents these theoretical numbers compared to the number of vertices and edges created when building an arrangement of n random lines. Note that hybrid edges are counted as regular edges.

	Theoretical		Practice			
n	vertices	edges	vertices	outer vertices	edges	outer edges
32	496	1024	496	64	2048	128
256	32640	65536	32640	512	131072	1024
2048	2096128	4194304	2096125	4096	8388548	8192

Table 4.2: The theoretical bound on vertices and edges compared to the actual values from running the algorithm.

The most obvious deviation from the theoretical bound is the number of edges, but the reason for this is simple. Recall that the data-structure is a

doubly-connected edge list, thus every edge has a twin pointing in the opposite direction resulting in a factor of 2 more edges. Outer edges and outer vertices are special in the sense that they do not have a direct connection to the theoretical bound. Their only purpose is to allow the algorithm to pick the correct starting edge to walk along in order to insert the line. Since they do not correspond to actual intersection points between lines, they are not counted in Theorem 2.1. Their number is however only linear in the input size since every line can only enter and exit the arrangement once and a constant number of edges is needed to connect a constant number of vertices in a circular list. Thus we can conclude that the total number of vertices and edges is very close to the numbers from Theorem 2.1 when build on random input.

There are several ways to deviate from the theoretical bound. First off, parallel lines can never intersect and will therefore result in less intersection points and less vertices. Furthermore, lines that coincidence will only insert edges and vertices once. If more than two lines intersect in the same point the total number of vertices will also decrease. Finally, numerical precision can cause any of the above situations to happen for arbitrary lines. Looking at Table 4.2, we see that for 2048 random lines both the edge and vertex count is lower than what is expected from the theoretical bound, i.e. there are missing 3 vertices and 30 edges. Which of the situations described above that causes this is hard to tell, but parallel lines should be really rare because only 2048 lines are sampled. By the same argument coinciding lines are also very unlikely. The most probable cause is numeric instability and gives us an indication on the amount of errors that we can expect in the resulting arrangement. Clearly, the number of errors is minimal and the total number of edges and vertices does not increase beyond the theoretical bound as expected.

4.8 Verification of the building time

Having established that the space complexity is within the theoretical bound, the focus of this section is to verify the theoretical building time of $O(n^2)$. Naturally, the first experiment consists of building an arrangement of n lines for an increasing input size and measuring the building time. Worth noting is that the space consumed by the algorithm is bounded by $O(n^2)$, thus large values of n becomes infeasible to handle because the operating systems starts swapping to slow disk memory. This is in itself an interesting effect, but drastically slows down the algorithm which produces dominating data points and makes it difficult to reason about input sizes that can fit in memory. The results from the experiment is visible in Figure 4.9.

As is visible in the graph, the building time is divided by n^2 since a running time of $O(n^2)$ will result in a curve converging towards a constant factor. Unfortunately, it is not clear that the algorithm has this property when looking at Figure 4.9. For n less than 2^5 the graph appears to be almost constant but there is a noticeable increase from 2^5 to 2^7 and an even more drastic increase from 2^7 to 2^{11}. This does not bode well for verifying that the building time of the arrangement algorithm is $O(n^2)$. However, the analysis may still prove

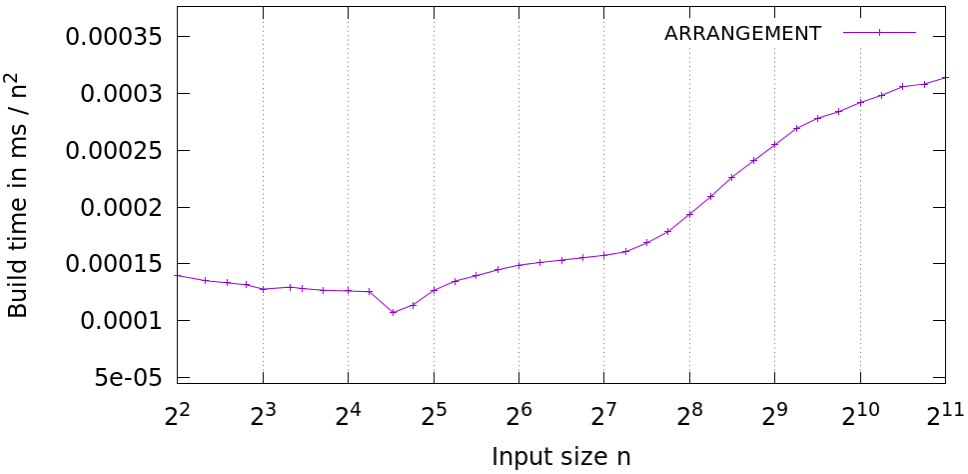


Figure 4.9: The build time in ms for constructing an arrangement of lines.

to be sound because the building time is most likely dominated by external factors, e.g. CPU cache misses. At some point the arrangement outgrows the size of the CPU caches which causes cache misses followed by a look-up in the next cache in the hierarchy. Such an operation takes more clock cycles and stalls the CPU while the data is fetched from a slower memory location. This phenomenon is not considered in the standard random-access machine model in which the analysis is carried out. However the claim is that the bound still holds and the discrepancies in the Figure 4.9 are a consequence of the machine architecture, i.e. cache misses on different levels of the cache hierarchy.

In order to investigate this behaviour, the next three experiments seek to support this claim. For this purpose we use `PAPI` - a library that counts hardware events such as cache misses and instructions. The number of L2 cache misses is subject to the following experiment and the result is visible in Figure 4.10.

Looking at Figure 4.10 we witness an increase in L2 cache misses that starts somewhere between an input of size 2^5 and 2^6 which aligns with the increase in building time in Figure 4.9. This supports the expectation and is in line with the theory. We know from Table 4.2 that the arrangement consists of $496+64 = 560$ vertices and $2048 + 128 = 2176$ edges in total. An edge uses 40 bytes and a vertex uses 32 bytes, thus the space consumed is at least $2176 \cdot 40 + 560 \cdot 32 \approx 105$ KB. Doing a similar calculation for $n = 2^6$ we get that the arrangement uses at least 406 KB. Comparing these results with the size of the L2 cache (256 KB) it is natural that L2 cache misses takes place because the size of the arrangement exceeds the size of the cache.

Though L2 cache misses causes the increase in building time between inputs of size 2^5 and 2^7, it does not explain the steeper increase for inputs larger than 2^7. We expect that a similar limit is reached at this point, namely the size of the arrangement exceeds the size of the L3 cache. The focus of the third experiment is to provide evidence for this hypothesis. The experiment is performed in the

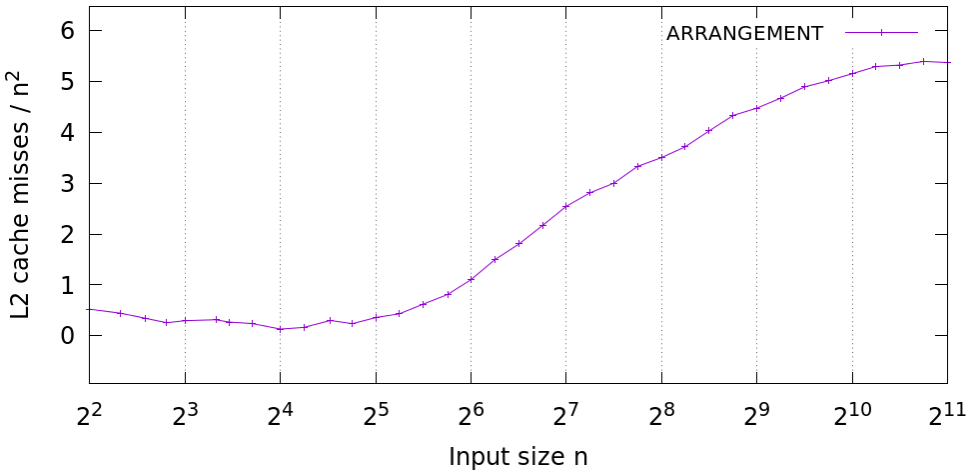


Figure 4.10: The number of L2 caches misses when constructing an arrangement of lines.

same setting as the previous, except that the program counts L3 cache misses instead of L2 cache misses. The result is visible in Figure 4.11.

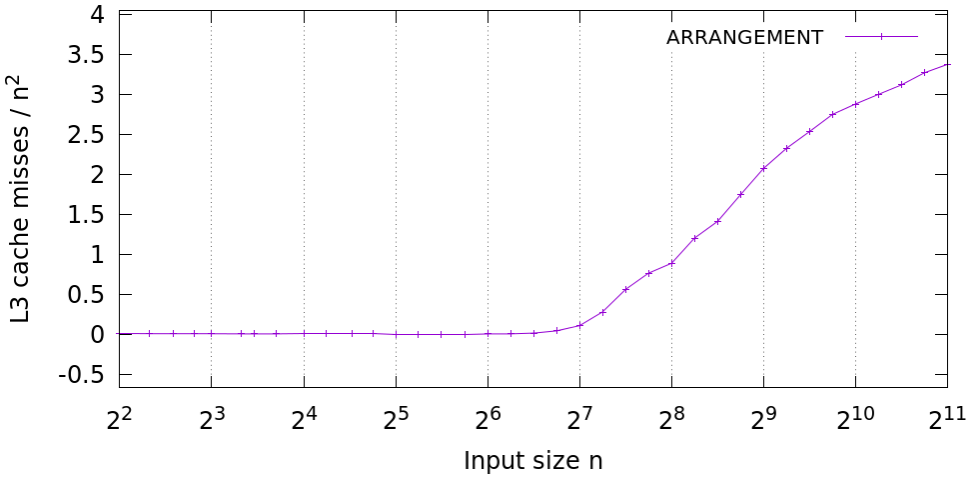


Figure 4.11: The number of L3 caches misses when constructing an arrangement of lines.

Again, the graph supports the expectation as the number of L3 cache misses aligns with the increase in building time in Figure 4.9. As for the previous experiment, one can calculate a lower bound on the memory that the arrangement uses to verify that this behaviour is correct.

Having established that the building time is influenced by cache misses, the question is whether the theoretic bound of $O(n^2)$ still holds in practice? It may be that the implementation is erroneous and simply computes more instructions than anticipated. To reason about this, the final experiment runs the same ex-

periment as before and counts the number of completed instructions performed by the CPU. As mentioned before, the analysis of the building time is carried out in the random-access machine model meaning that basic arithmetic takes constant time and a building time of $O(n^2)$ corresponds to $O(n^2)$ instructions being executed by the CPU. Hence, we expect the number of instructions performed by the algorithm to converge towards a constant value when divided by n^2. The result of the final experiment is visible in Figure 4.12.

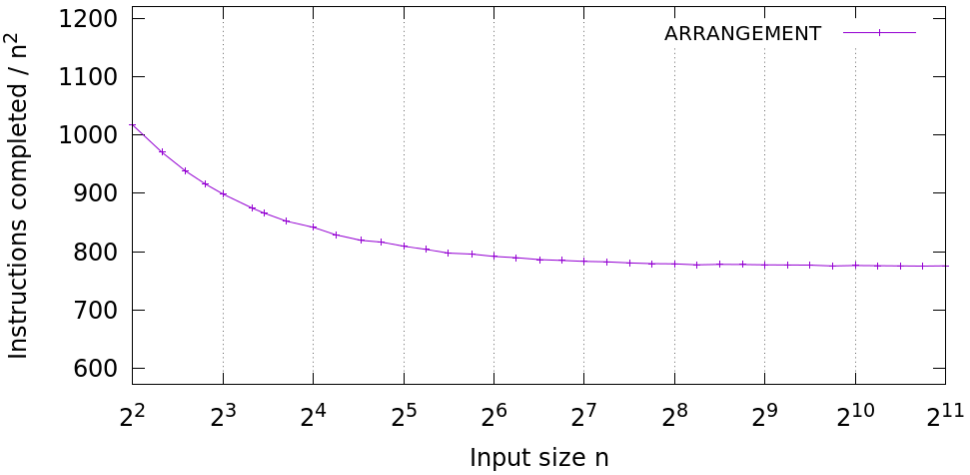


Figure 4.12: The number of completed instructions when constructing an arrangement of lines.

The graph does not give rise to further discussions and clearly shows that the number of instructions is bounded by $O(n^2)$. Based on this fact, it seems fair to conclude that the building time of the arrangement algorithm is indeed bounded by $O(n^2)$ when assuming the random-access machine model, but in practice the time spend is influenced by the clock cycles it takes to handle cache misses which depends on the underlying machine architecture.

4.9 Conclusion

The goal of this chapter was to present and experiment with a versatile, efficient and robust arrangement algorithm that can be used as a tool for other procedures. The resulting algorithm can construct unbounded, semi-bounded and bounded arrangements of lines in $\mathbb{R}^2$, including coinciding and parallel lines. Even though the building time is influenced by cache misses, the building time is bounded by $O(n^2)$ and the space complexity is $O(n^2)$, which is in line with the theory on the subject. In terms of robustness, the algorithm may produce a faulty arrangement or even loop indefinitely because of numerical instability. Though this may be solved using rational numbers, the slowdown makes the algorithm useless in practice. In conclusion, the trade-off between speed and precision promotes the use of doubles for doing arithmetic operations - it is simply the better alternative.

Chapter 5

Cuttings

In this chapter we describe, implement and analyze two algorithms that creates a $\frac{1}{r}$-cutting in $\mathbb{R}^2$ and compare them in order to determine which is faster. The motivation for this study is closely connected with Chapter 4 as we are seeking an algorithm that efficiently partitions the plane into as few triangles as possible. In Section 5.1 and Section 5.2 we present a naive cutting algorithm and describe the details encountered during implementation. A more advanced cutting algorithm is described in a similar fashion in Section 5.3 and Section 5.4. In Section 5.5 we present a brief description of the tests performed and some considerations about robustness. The experiments presented in Section 5.6 and Section 5.7 seek to determine the performance and quality of the algorithms in order to determine a superior algorithm. The chapter is concluded in Section 5.8.

5.1 The naive cutting

By Definition 2.5 a $\frac{1}{r}$-cutting of n lines L in $\mathbb{R}^2$ partitions the plane into triangles where each triangle is cut by at most $\frac{n}{r}$ lines. A trivial solution builds the entire arrangement of L and triangulates the faces. Triangulating the faces are easy because they are convex, i.e. an edge is simply added from one vertex v to all other vertices that are not directly connected to v inside the face. This produces a cutting of size $O(n^2)$, because there are $O(n^2)$ vertices, where no lines intersect any triangle and has a space and time complexity of $O(n^2)$. This is useless seeing that the goal is to use the cutting as a subroutine in a divide and conquer algorithm, but the sub-optimal upper bound proven in Lemma 2.5 allows hope for improvements.

The naive cutting algorithm denoted `CUTTING_NAIVE` is an implementation of the algorithm sketched in the proof of Lemma 2.5 and creates a cutting given r by sampling a subset $Y \subseteq L$ of size $c \cdot r \log r$. The sampling is uniform and picks lines from L with replacement. The algorithm builds an unbounded arrangement of Y using `ARRANGEMENT` and triangulates the faces. The details of the triangulation is explained in Section 5.2. Each triangle Δ is checked against the lines in L to see if the crossing number Δ_c exceeds the threshold $\frac{n}{r}$. Note that triangles are open, thus avoiding the lines on the boundary to contribute to

the crossing number. If a triangle exceeds the threshold, the algorithm stops and starts over. The size and complexity of this cutting algorithm improves greatly over the trivial solution and is proven below. We also present an example of a $\frac{1}{3}$-cutting of 9 lines in Figure 5.1, where no more than 3 lines may intersect any triangle.

Theorem 5.1. Given a set L of n lines in $\mathbb{R}^2$, `CUTTING_NAIVE` creates a $\frac{1}{r}$-cutting of size $O(r^2 \log^2 r)$ in $O(nr^2 \log^2 r)$ time and uses $O(r^2 \log^2 r)$ space.

Proof. By Theorem 2.5 we know that there exists a cutting of size $O(r^2 \log^2 r)$ that results in a $\frac{1}{r}$-cutting of size $O(r^2 \log^2 r)$. What we need to argue is that the algorithm creates an $\frac{1}{2r}$-net of size $O(r \log r)$. By Lemma 2.3 we can do this with probability $1 - \delta$ by sampling m lines from L, where m is defined in the lemma. Setting $\delta = \frac{1}{r}$ we get a constant probability of this happening and the sample size becomes $m = O(r \log r)$. Thus we can expect the algorithm to find an $\frac{1}{2r}$-net within a constant number of trials. Note that for $r = 1$ the probability of success becomes 0, but in that case the input is a $\frac{1}{r}$-cutting because n lines are allowed to cross each triangle.

In terms of running time the algorithm builds an arrangement on $O(r \log r)$ lines which takes $O(r^2 \log^2 r)$ time. The triangulation is a traversal of the arrangement and can be handled within the same bound. Verifying a triangle takes $O(n)$ time, hence the total running time becomes $O(nr^2 \log^2 r)$. Note that the algorithm is expected to repeat this a constant number of times which does not change the asymptotic bound. The space complexity is $O(r^2 \log^2 r)$ due to the construction of the arrangement. □

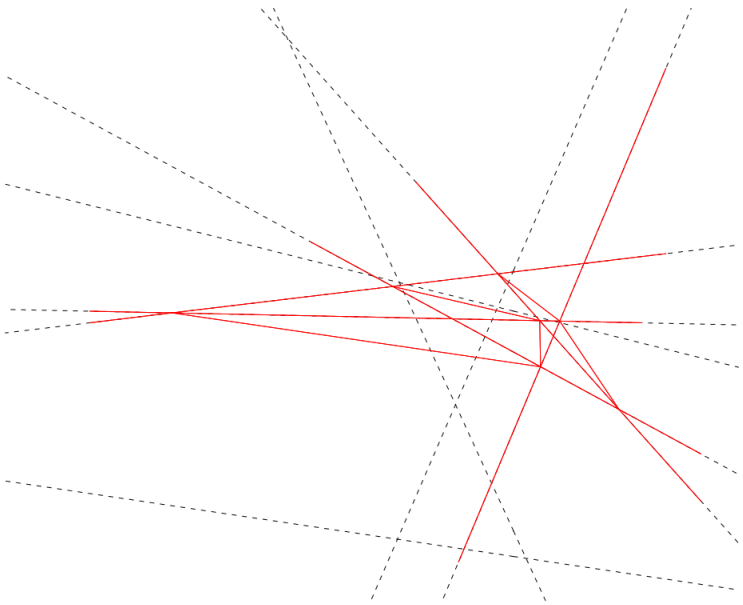
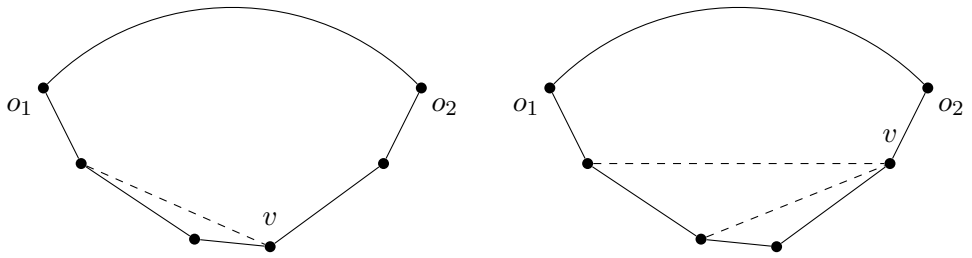


Figure 5.1: An example of a $\frac{1}{3}$-cutting of 9 lines.

5.2 Details of the naive cutting

The naive cutting algorithm is somewhat straightforward to implement when having access to an unbounded arrangement algorithm, because it creates a natural subdivision of the unbounded face. Nevertheless, there are some details that are worth mentioning, like the triangulation procedure. First of all, the arrangement must be traversed such that created triangles are verified exactly once. The is done using a breadth-first search as described in Section 4.2. When a vertex v is removed from the queue all of its incident faces are traversed. In each face the algorithm determines if v is the bottom vertex inside the face, i.e. the vertex with the lowest y-coordinate, and whether the face is unbounded or not. The triangulation procedure handles bounded and unbounded faces differently, so we will explain them separately.

If the face is bounded and v is the bottom vertex, the algorithm traverses the face and triangulates it. Let w denote a vertex visited during such a traversal. If w is a direct neighbor to v there is already an edge connecting them and no crossing edge is created. Otherwise, a crossing edge is created between v and w and the created triangle Δ is checked against all lines in L to verify that Δ_c does not exceed the threshold $\frac{n}{r}$. When the traversal is done, the last triangle is verified in the same manner. Thus, a bounded face is triangulated by a unique bottom vertex and the created triangles are verified exactly once because the breadth-first search visits all vertices exactly once.



(a) An incorrect triangulation. (b) A correct triangulation.

Figure 5.2: Triangulating unbounded faces.

When the face is unbounded, the algorithm traverses the face and triangulates it even though v is not the bottom vertex. This is necessary because the bottom vertex can create an unbounded triangle with 5 borders as depicted in Figure 5.2a. Note that crossing edges are not created when either v or w is an outer vertex. In order to avoid creating such triangles, the algorithm starts by traversing the outer vertices of the arrangement, marks them visited and adds them to the queue. Thus, unbounded faces will always be triangulated by a vertex v with an outer vertex as its neighbor as depicted in Figure 5.2b. In other words, the arrangement is triangulated from the outside in. Verification of a created triangle is only done if v is the actual bottom vertex in that triangle. This potentially leaves triangles unverified when triangulating unbounded faces. That is of course not the case because each of these triangles will have a bottom vertex that has not been visited by the breadth-first search, which

follows from the fact that the arrangement is triangulated from the outside in.

Another interesting implementation detail is evident when calculating the crossing number for unbounded triangles. Since triangles are open, a line l from L cannot intersect a triangle in the vertices bounding it, i.e. the line has to cross the interior of the triangle. This immediately suggest an intersect procedure that checks whether l intersects the interior of at least one of the edges bounding the triangle. This works for bounded triangles, but is not sufficient for unbounded triangles as shown in Figure 5.3 where the line clearly intersects the triangle without intersection the interior of the hybrid edges.

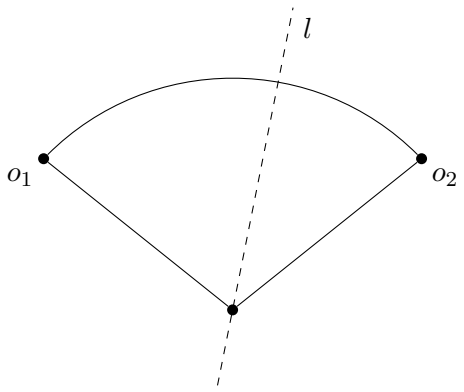


Figure 5.3: A line that intersect an unbounded triangle without intersecting the interior of the edges bounding the triangle.

We handle this by checking the parameters of the line against the parameters stored in the outer vertices o_1 and o_2. If the slope and intersection of l is within these parameters, the line must intersect the interior of the unbounded triangle. This check boils down to a case analysis depending on the outer vertices, i.e. if they are in the starting or ending region as depicted in Figure 4.5. In case 4.5a the line intersects the unbounded triangle if it has a smaller slope than the leftmost hybrid edge and a larger slope than the rightmost hybrid edge. The intersection with the y-axis is checked on equality. Handling the remaining three cases is equally simple and is left out.

As a final remark, the algorithm increments the number of sampled lines when it starts over with a fresh sample. In theory, this is unnecessary, but the algorithm may loop forever for small inputs, thus this fix ensures termination. This will potentially create a larger cutting but should not be a serious problem in practice.

5.3 The fixing cutting

The naive cutting algorithm proves the existence of a $\frac{1}{r}$-cutting of size $O(r^2 \log^2)$, but an even better bound may exist due the lower bound of size $\Omega(r^2)$ stated in Lemma 2.4. In fact, as hinted in Section 2.5, the upper bound can be proven to be $O(r^2)$ as well. The cutting algorithm described in this section is denoted `CUTTING_FIX` and proves the existence of such a cutting. The algorithm is a

two-level random sampling algorithm and creates an initial cutting given r by sampling a subset $Y \subseteq L$ of size $c_1 \cdot r$. The sampling is uniform and picks lines from L with replacement. The algorithm builds an unbounded arrangement of Y using `ARRANGEMENT` and triangulates the faces using a bottom vertex triangulation. Each triangle Δ is checked against the lines in L to see if the crossing number Δ_c exceeds the threshold $\frac{n}{r}$. If a triangle exceeds the threshold, the algorithm computes the excess $\Delta_e = \Delta_c \cdot \frac{r}{n}$ and creates a $\frac{1}{\Delta_e}$-cutting of the triangle using a sample of size $O(\Delta_e^{c_2})$. If the triangle still exceeds the threshold after being refined once, the algorithm stops and starts over. The size and complexity of this cutting is stated in the theorem below and Figure 5.4 shows an example of a $\frac{1}{3}$-cutting of 9 lines created by the algorithm.

Theorem 5.2. Given a set L of n lines in $\mathbb{R}^2$, `CUTTING_FIX` creates a $\frac{1}{r}$-cutting of size $O(r^2)$ in $O(nr^2)$ time and uses $O(r^2)$ space.

Proof. We refer to [A9] where it is proven that the algorithm produces a $\frac{1}{r}$-cutting of size $O(r^2)$.

The arguments for the space and time complexity is identical to the ones presented in the proof of Theorem 5.1, thus the time complexity is $O(nr^2)$ and the space complexity is $O(r^2)$. □

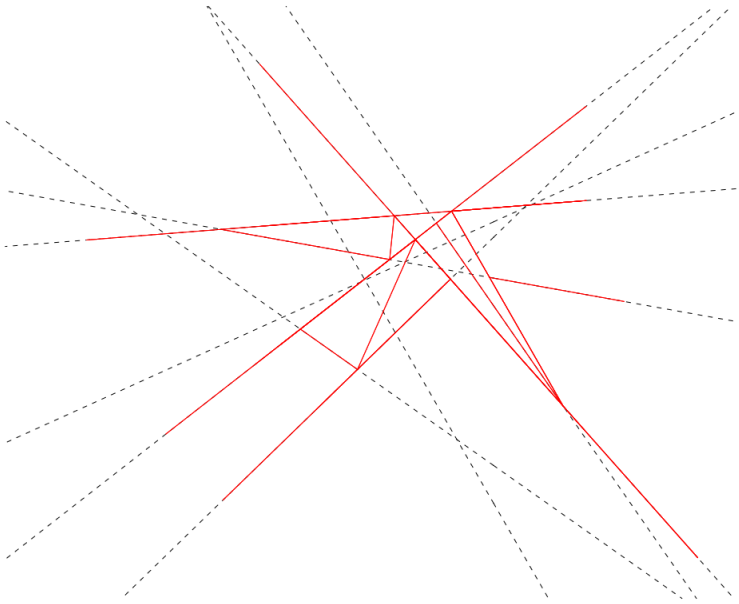


Figure 5.4: An example of a $\frac{1}{3}$-cutting of 9 lines.

5.4 Details of the fixing cutting

For the proof of Theorem 5.2 to hold, the implementation of `CUTTING_FIX` requires a bottom vertex triangulation, because it meets the requirements of

three certain properties: Firstly, all triangles of the triangulation must have a defining set of constant size. Secondly, for a triangle to appear the defining set must be sampled. Lastly, a triangle is created if the former two properties hold and no lines crossing the triangle are sampled. These properties are all true for a bottom vertex triangulation and is proven in [A9]. Unfortunately, the triangulation implemented is identical to the one described in Section 5.2, which is a bottom vertex triangulation in all but the unbounded faces. We argue that there are at most $O(r)$ such faces, so even though the bound does not hold it should not be that bad in practice. We will take a closer look at the performance and quality of the cutting in Section 5.6.

Another noticeable detail is that the algorithm described in Section 5.3 stops and starts over after one level of refinement. This is not the case in the source code, because the algorithm keeps refining triangles until a $\frac{1}{r}$-cutting is obtained. Again, this may degrade the quality of the cutting but will improve on the running time seeing that the algorithm does not have to start over.

5.5 Testing and robustness

The cutting algorithms presented in this chapter are closely related to the arrangement algorithm and share most of the hassles the arrangement introduces. Basically, if anything goes wrong during the construction of the arrangement, be it numerical issues or some other degeneration, the resulting cutting may be faulty or loop forever. Unfortunately, the cutting algorithms can do nothing to remedy any of these problems and it therefore remains an occupational hazard.

To add insult to injury, the additional algorithmic layer of the cutting algorithms add even more numerical instability. When triangles are verified the intersection procedure can cause numerical errors and result in an erroneous crossing number. This may cause either a total rebuild or an additional refinement step depending on the algorithm, which is not desirable. Unlike the arrangement algorithm the additional algorithmic layer is much less prone to infinite loops. The naive algorithm will simply try again if it fails but with a fresh sampled set, avoiding problems caused by the previous sample. The fixing algorithm will refine and eventually build an arrangement of relatively small size. Thus, if the construction of the arrangement does not loop forever, both cutting algorithms will most likely never loop indefinitely. A possible solution to these issues is to use rational numbers but that still comes with a heavy reduction in running time.

The cutting algorithms gives rise to an additional headache. The problem is that the arrangement cannot handle lines that coincide with the edges of the bounding triangle or lines the coincide with the hybrid edges in a semi-bounded arrangement. The algorithm is simply not able to determine a unique starting point for such lines since they will intersect anything on the boundary of the arrangement. With infinite precision, these lines will never be added to the arrangement, but numerical errors makes it possible. This is especially problematic since many of the triangles from the triangulation will have edges that originates from lines inserted into the arrangement.

Luckily there is an obvious solution to prevent these lines from being added, namely by equipping all lines with an unique id. When the arrangement is built, the edges created by a given line stores its id for future reference. The triangle intersection procedures can then use these id's in order to avoid inserting a line into an arrangement bounded by an edge originating from this line. This does not eliminate the problem entirely because input lines may still coincide with some of the edges created by the triangulation.

As a final remark, the testing situation for the cutting algorithms are sub-optimal, just as it was for the arrangement. We implemented an automatic test that verifies the properties of the doubly-connected edge list produced by the cutting. Unfortunately, performing more sophisticated automatic tests is more or less impossible since no local feature of the edge list can prove that the corresponding cutting is correct. The remaining tests are performed manually by drawing the cutting and counting the number of triangles created by the algorithm.

5.6 Fine-tuning the cutting algorithms

In general, a cutting algorithm has two main features that are worth experimenting with. First of all is the building time because it directly affects any application depending on the algorithm. Secondly, the size of the cutting, also referred to as the quality, is also relevant since the goal is to divide the input into as few sub-problems as possible. Trivially, the best of both worlds is ideal but one property might exclude the other because there is no obvious relation between them. Knowing this relation, and how it applies to the cutting algorithms presented in this chapter, is therefore of high interest and we will in this section experiment with the running time and the quality for both cutting algorithms in order to reason about it and consequently announce a winner.

5.6.1 The naive cutting

Both cutting algorithms make use of unknown constants that determine their sampling rate, therefore it makes sense to experiment with these constants before comparing them. The naive algorithm only has a single constant c scaling the number $r \log r$ of lines used to created the cutting. Hence, the objective of the first experiment is to identify the relation between c and the construction time of the naive cutting algorithm. The experiment consists of running `CUTTING_NAIVE` with varying c to compute a $\frac{1}{10}$-cutting of 1000 lines while measuring the building time. Multiple runs are performed and the average is plotted. Note that this setup is similar for the remaining experiments of this chapter unless otherwise specified. The result of the experiment is visible in Figure 5.5.

Before we analyze the results of the experiment it makes sense to discuss the expected outcome. Sampling few lines will result in a small arrangement being build and triangulated. Clearly, this is desirable since the arrangement algorithm takes order $O(n^2)$ to build and a small n will result in a fast building time. The problem with a small sampling rate is that it restrict the number of

triangles, thus it may require many attempts before a suitable cutting is found. On the other hand, sampling to many lines will potentially waste time building a much to big arrangement. Based on this it seems reasonable to assume there will be a sweet spot when the arrangement is of adequate size to produce a valid cutting.

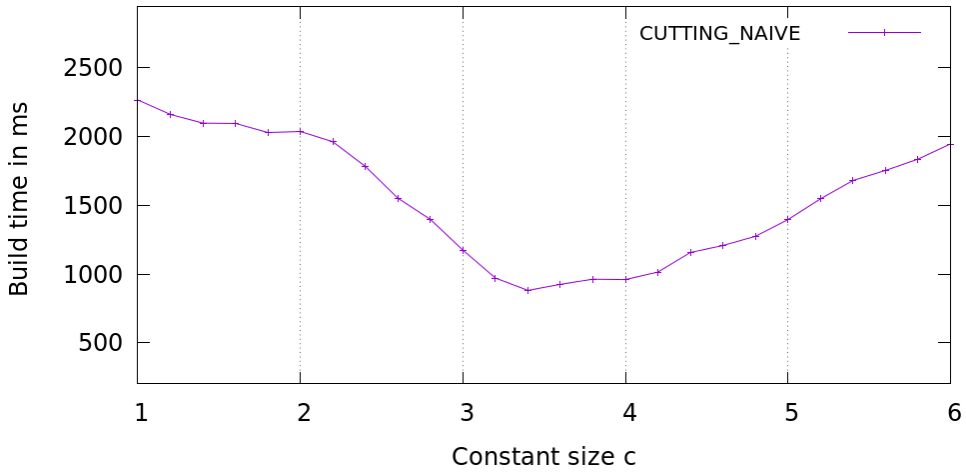


Figure 5.5: The build time in ms when constructing a $\frac{1}{10}$-cutting using the naive cutting.

Looking at Figure 5.5 such a sweet spot is evident. The building time prior to the sweet spot is dominated by the fact that it requires many trials with a relatively small sample before a valid cutting is found. The decline in building towards the sweet spot also makes sense because the growing arrangement will have a higher probability of generating a valid cutting. After the sweet spot, the building time worsens again because the time it takes to construct the arrangement is dominating the build time. In other words, the algorithm saves the cost of rebuilding, but pays the price due to the arrangement construction. One might suspect that the sweet spot takes place when the algorithm succeeds by creating a single arrangement on average. This is somewhat true, but by rerunning the experiment and counting the number of rebuilds performed by the algorithm, the sweet spot actually takes place when the algorithm creates between 1 and 2 arrangements on average. This is to be expected because the complexity of building an arrangement is quadratic, hence it is faster to make a few trials with a slightly smaller input rather than succeeding in the first try every time.

Having analysed the building time of the naive cutting algorithm, the next experiment measures the influence c has on the quality of the cutting. The result of the second experiment is presented in Figure 5.6 where the average number of triangles are plotted. Again it makes sense to discuss the expected result before moving on to the analysis. Unlike the building time, the quality of the naive algorithm should have an obvious quality progression. The algorithm does nothing fancy with the generated arrangement and the number of triangles

is simply bounded by the total number of vertices in the arrangement. Because of this, the number of generated triangles should therefore be minimal when the number of sampled lines is minimal.

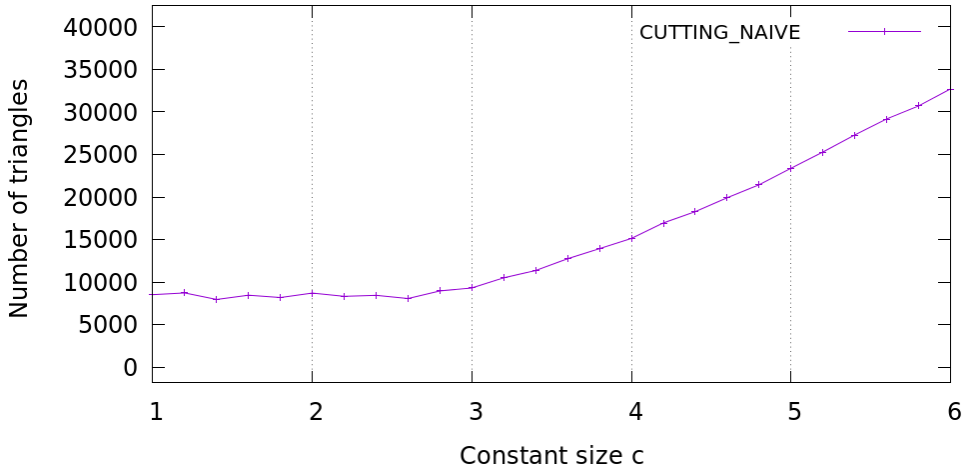


Figure 5.6: The number of triangles when constructing a $\frac{1}{10}$-cutting using the naive cutting.

This trend is clearly visible in Figure 5.6 as smaller values of c results in better quality. Interestingly, the quality appears to be constant up until $c = 3$. This is most likely an artifact caused by the algorithm incrementing the number of sampled lines when a trial fails, i.e. it converges towards a fixed number of sampled lines that successfully creates an $\frac{1}{10}$-cutting. This also supports the drop in running time in Figure 5.5 between $c = 1$ and $c = 2$ since the algorithm will reach this number within fewer iterations. Rerunning the experiment with a focus on the number of sampled lines that causes a successful cutting to be created, verifies that this claim is correct. These two experiments also supports the fact that some trade-off between building time and quality exists. Even though the algorithm seems to be relatively fast and produce a relatively small sized cutting at $c = 3$, the best of both worlds cannot be obtained as a cutting with high quality will incur a building time penalty, and vice verse.

5.6.2 The fixing cutting

Similar experiments must be conducted for `CUTTING_FIX` which is the subject of this section. In this case there are two constants affecting the size and building time of the algorithm, i.e. the constant c_1 that scales the initial sampling size r and c_2 that scales the excess sampling size. Since there is no apparent connection between these two constants we simply start by fixing c_1 and perform the first experiment by varying c_2. Thus, the first experiment performed for `CUTTING_FIX` is the same as the first experiment for `CUTTING_NAIVE` except that the effect of c_2 is measured. The results of this experiment is visible in Figure 5.7.

Since c_2 is used to determine the sampling size for triangles that violate

the crossing number, it will affect the number of refinements performed by the algorithm. If the constant is low, it might cause additional refinement steps to be performed resulting in more arrangements being build and consequently a slower algorithm. Likewise, a large constant could cause the construction of unnecessarily large arrangements resulting in the same effect. This gives rise to a trade-off between the number of refinement steps and the size of the refinement arrangements, which is reminiscent of the situation for the naive algorithm. Based on this it is expected that c_2 should also have a sweet spot with respect to the building time.

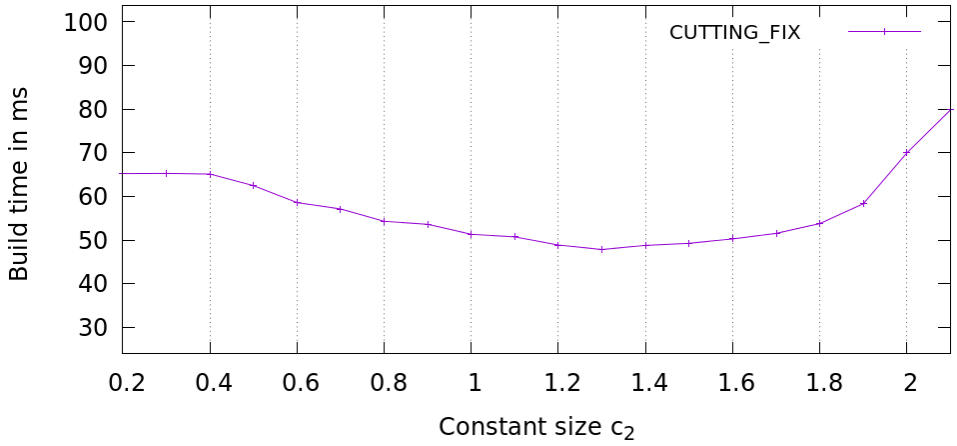
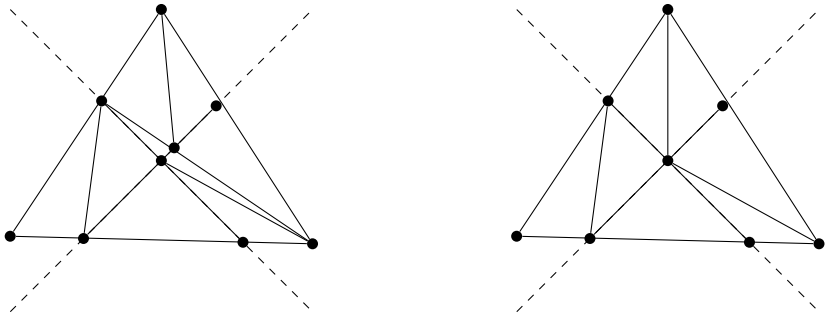


Figure 5.7: The build time in ms when constructing a $\frac{1}{10}$-cutting using the fixing cutting.

Looking at Figure 5.7 this expectation is clearly fulfilled. Counting the number of refinements performed for the different values of c_2 during the experiment confirms, that prior to the sweet spot the algorithm performs a lot of refinements by creating small arrangements. This number decreases as c_2 gets bigger and becomes optimal in terms of building time for $c_2 = 1.3$. Performing the experiment beyond what is visible in Figure 5.7, we also experience that the number of refinements performed converges towards the number of triangles in the initial cutting with two many lines crossing it. This makes sense because the constant c_2 becomes so large that a single refinement is sufficient. After this point, increasing the constant will only spend more time on building unnecessary large arrangements resulting in much worse building times. Unlike the constant c for the naive algorithm, the price paid for using a small value is more subtle. It is more evident that using a too large value of c_2 will cause a much more significant slowdown.

Having seen the effects of c_2 in terms of building time, we turn our attention to the quality. The setup is identical to the former, except that we measure the number of triangles created. The result is depicted in Figure 5.9. For this experiment, it is rather difficult to predict how smaller values of c_2 will affect the quality. In the extreme case, a single line is chosen when sampling

the excess, i.e. triangles are simply cut into two parts and triangulated as shown in Figure 5.8a where two lines are sampled using four refinement steps resulting in 9 triangles. Sampling the same two lines in a single refinement step yields a better cutting with 7 triangles as illustrated in Figure 5.8b and provides evidence that sampling fewer lines is not always optimal. So, if we are to expect anything it must be that there will be be sweet spot where the quality is optimal, since we trivially expect an upper bound on c_2 where too many lines are sampled causing the quality to degrade.



(a) Sampling in four refinement steps. (b) Sampling in one refinement steps.

Figure 5.8: Sampling two lines crossing a triangle in four and one refinement steps respectively.

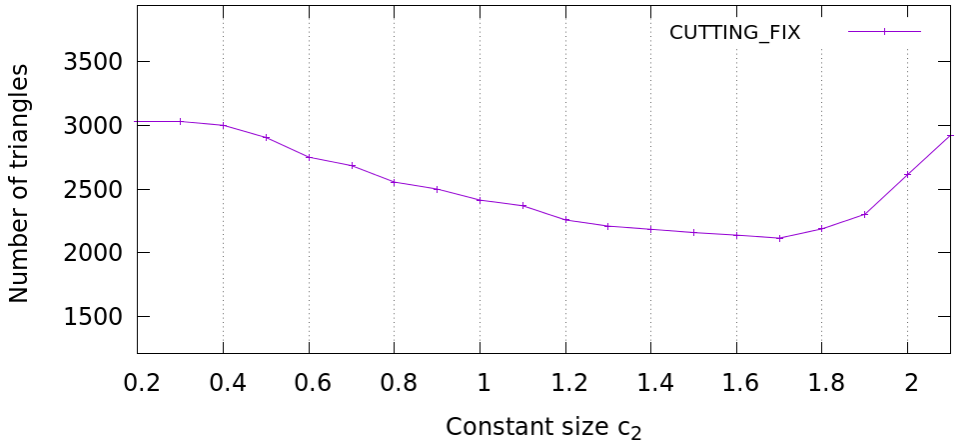


Figure 5.9: The number of triangles when constructing a $\frac{1}{10}$-cutting using the fixing cutting.

The result of Figure 5.9 shows that such a sweet spot exists at around $c_2 = 1.7$. Measuring the number of refinement steps during the experiment, we see that the number of refinements gradually decreases from around 1500 to 250 as c_2 gets bigger. This is in line with our expectation and makes sense because a low sampling rate will have a lower probability of partitioning invalid triangles into valid triangles, thus additional steps are required which can produce a

lower quality cutting as depicted in Figure 5.8. In the other extreme, too many lines are sampled and the cutting created by sampling the excess becomes unnecessarily large. In contrast to the naive cutting algorithm, it seems that there is a smaller trade-off between building time an quality. Picking c_2 between 1.3 to 1.8 produces a cutting of high quality without increasing the building time significantly, which suggests that this algorithm has the upper hand.

In addition to the results in Figure 5.9, we also implemented a version that stops whenever the cutting is not valid after one level of refinement, i.e. the algorithm does not keep refining triangles that exceeds the threshold. This implementation is analogous to the one explained in Section 5.3. The motivation for this experiment is that our implementation might produce a larger cutting of lower quality. Even though we tweaked the parameters for this version, we were not able to produce a $\frac{1}{10}$-cutting of 1000 lines with less than 6000 triangles. Comparing this with 2200 triangles, we can conclude that even though our implementation diverges from the theory, it seems to perform well in practice. We excluded the graph with the results since it did not provide further insight.

What remains to cover is the effect of the initial sampling constant c_1. For this purpose the building time and number of triangles are measured with c_2 set to an optimal value based on the former experiments. In terms of expectations, there is nothing new under the sun because too small values should cause many refinements which degrades the quality. Too large values should build an excessive arrangement causing both the quality and building time to degrade. The results are depicted in Figure 5.10 and Figure 5.11.

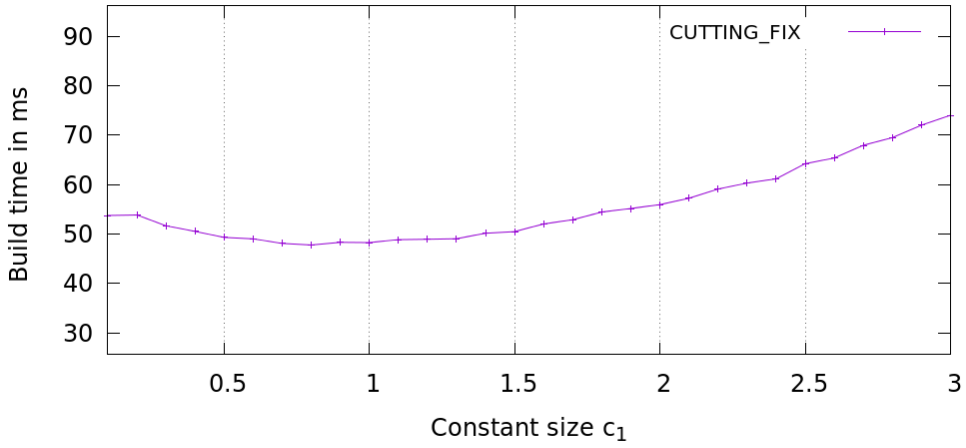


Figure 5.10: The build time in ms when constructing a $\frac{1}{10}$-cutting using the fixing cutting.

The results from the graphs simply emphasizes what we already discovered from the previous experiments: There exists a sweet spot in terms of building time and quality. Fortunately, these spots seems to overlap in a large range. In conclusion, setting $c_1 = 1$ seems appropriate and in line with the theory.

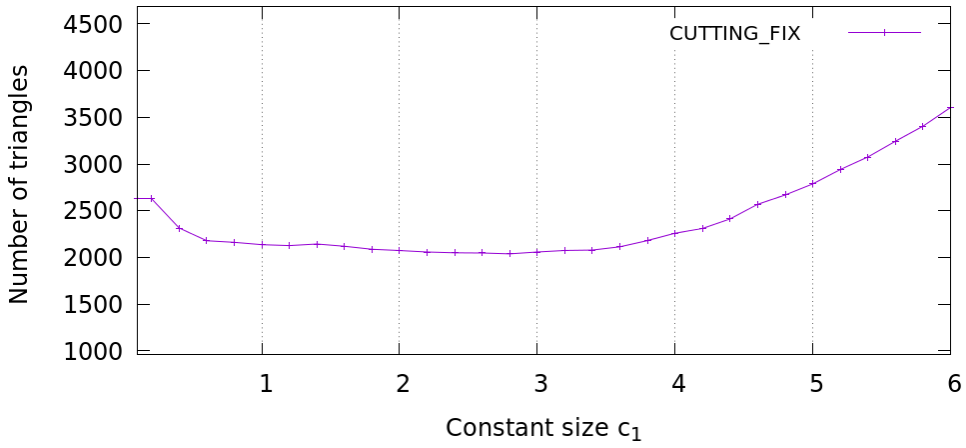


Figure 5.11: The number of triangles when constructing a $\frac{1}{10}$-cutting using the fixing cutting.

5.7 Comparing the cutting algorithms

Having established optimal constants for both algorithms in terms of building time and quality, we are now able to compare them against each other and verify the theory on the subject. Based on the discoveries from Section 5.6.1 and Section 5.6.2 we let $c = 3.0$, $c_1 = 1.0$ and $c_2 = 1.6$ in the following experiments. Naturally, the first experiment measures the building time for the two cutting algorithms on input of varying size. In terms of the outcome, we expect that the naive algorithm will be inferior due to the bounds stated in Theorem 5.1 and Theorem 5.2. The result is shown in Figure 5.12. Note that the running time is divided by n, thus both algorithms should converge towards a constant value seeing that the sampling factors does not depend on n.

The resulting figure clearly shows that the naive algorithm is indeed inferior. The algorithm actually dominates the building time in such a degree that it is left out in Figure 5.13 in order to better reason about the fixing algorithm. Looking at the measured data we can see that the fixing algorithm is a factor 26 faster than the naive algorithm for $n = 2^{14}$, which is a significant slowdown for a relatively small input.

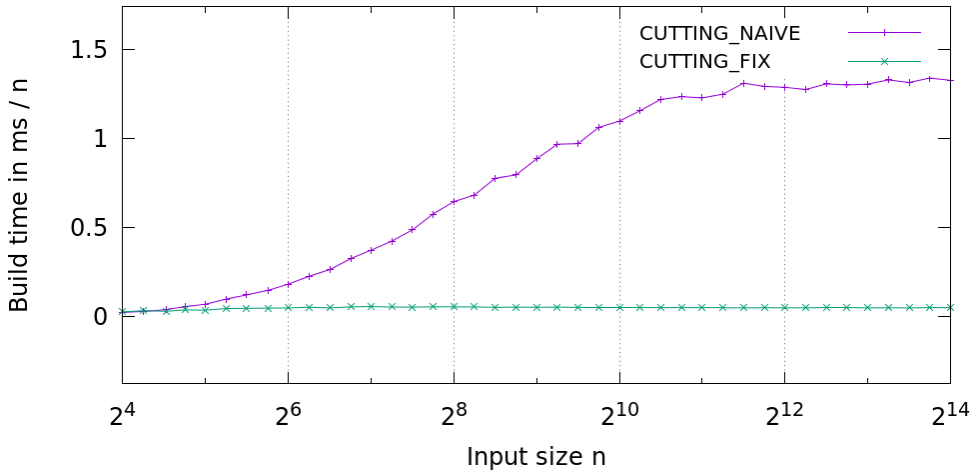


Figure 5.12: The build time in ms when constructing a $\frac{1}{10}$-cutting using both algorithms.

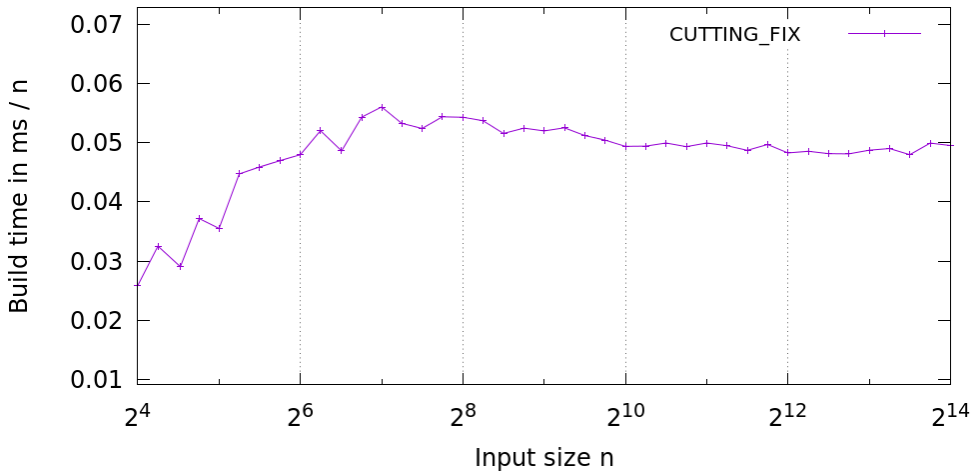


Figure 5.13: The build time in ms when constructing a $\frac{1}{10}$-cutting using the fixing cutting.

The figures also reveal that the building time of both algorithms converges as expected. One peculiarity is that the curves does not converge until the input has a substantial size. We believe that this is an effect of the initial sampling factor. In order to clarify, we have that $r = 10$ hence the naive algorithm samples $3 \cdot 10 \cdot \log 10 \approx 100$ lines. This will cause the algorithm to make a relatively large sample compared to small values of n, and explains why the building time of the naive algorithm evidently starts converging around $n = 2^{10} = 1024$ in Figure 5.12. This also explains why the fixing algorithm converges faster than the naive algorithm because the initial sampling factor is merely 10 which is relatively low compared to any n.

The final experiment of this chapter compares the quality using the stated

optimal constants. The expectation is unchanged because the fixing algorithm should be superior due to the bounds stated in Theorem 5.1 and Theorem 5.2. The result of running both algorithms is depicted in Figure 5.14 whereas the naive algorithm is left out in Figure 5.15.

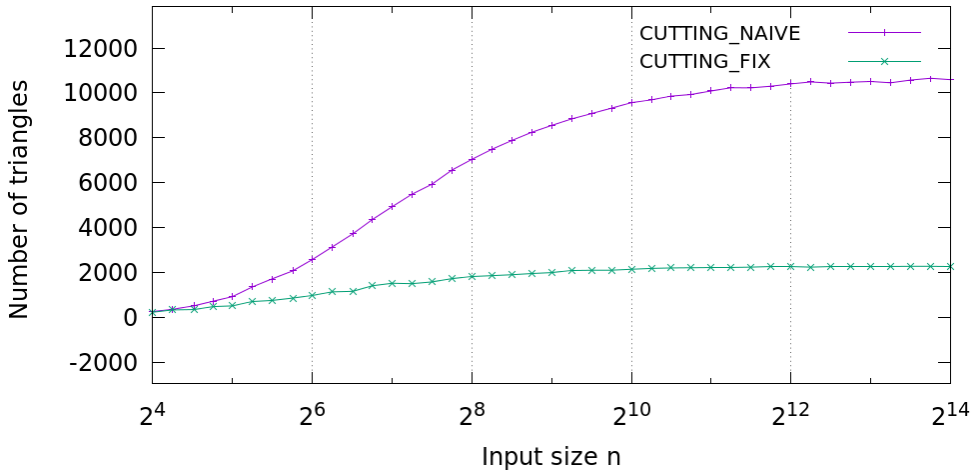


Figure 5.14: The number of triangles when constructing a $\frac{1}{10}$-cutting using both algorithms.

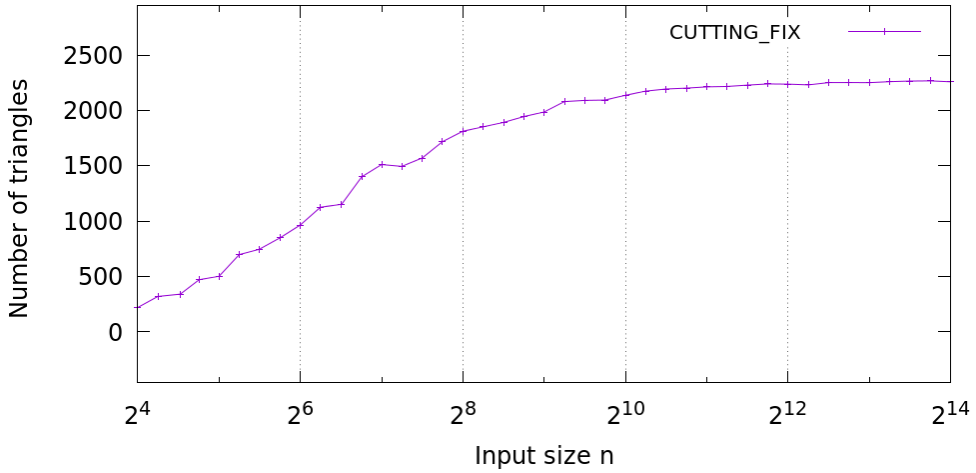


Figure 5.15: The number of triangles when constructing a $\frac{1}{10}$-cutting using the fixing cutting.

Looking at Figure 5.14 it is evident that the naive algorithm is inferior in terms of quality as well. For $n = 2^{14}$ the naive algorithm creates 10586 triangles on average. Comparing this with the 2259 triangles that the fixing algorithm creates we get a factor 4.6 difference in quality, which is substantial given that the quality directly affects the number of recursions in a divide and conquer context. In terms of the theoretic bound, the naive algorithm is expected to

create in the order of $10^2 \cdot \log^2 10 \approx 1103$ triangles meaning that it is off by an approximate factor of 10. The fixing algorithm is off by an approximate factor of 22 by an analogous argument. These constants are hidden in the asymptotic notation and shows that the bounds on the size of the cutting are relatively loose.

The peculiarity mentioned for the building time also applies for the quality experiment. The naive algorithm starts converging at $n = 2^{10}$ and up until this point, the number of sampled lines gradually increases. As mentioned, 100 lines are sampled with replacement, thus the created arrangement grows steadily until 100 distinct lines are sampled each time. Furthermore, this causes the $\frac{1}{10}$-cutting to fail more often and the algorithm retries with an incremented sample size resulting in more triangles. Evidently, the number of trials becomes constant on average at $n = 2^{10}$ where the curve starts converging. A similar phenomenon affects the fixing cutting in Figure 5.15. Measuring the number of refinements that the algorithm issues during the experiment, we observe that they becomes constant at around $n = 2^{10}$ where the number of triangles starts converging. In conclusion, the experiments conducted is in line with the theory and the fixing algorithm is superior in terms of building time and quality.

5.8 Conclusion

In this chapter we presented and compared two algorithms for creating a $\frac{1}{r}$-cutting in $\mathbb{R}^2$. The first algorithm `CUTTING_NAIVE` runs in $O(nr^2 \log^2 r)$ time and creates a cutting of size $O(r^2 \log^2 r)$ whereas the more advanced algorithm `CUTTING_FIX` runs in $O(nr^2)$ time and creates a cutting of size $O(r^2)$. Both algorithms depends on random sampling and must be fine-tuned in order to perform optimally. For the fixing cutting it appears that $c_1 = 1$ and $c_2 = 1.6$ is optimal, but that still produces a cutting that is 22 times larger than what is expected. From a theoretical point of view, the latter algorithm is superior in terms of building time and quality, which we have shown also holds in practice. As a final remark, both algorithms inherit the robustness issues from the arrangement algorithm, thus the same trade-off between running time and precision is still relevant in this context.

Chapter 6

Halfspace median

In this chapter we study the problem of finding the halfspace median in $\mathbb{R}^2$. In Section 6.1 we present a simple naive algorithm with an impractical running time. In Section 6.2 and Section 6.3 we describe a more sophisticated divide and conquer algorithm using the concepts of arrangements of lines and cuttings as presented in Chapter 4 and Chapter 5. For this purpose we assume that the input is in general position, i.e. there exists no coinciding points and there are no more than two points on a line. In Section 6.4 we present a brief description of the tests performed and some considerations about robustness. The experiments presented in Section 6.5 are aimed at fine-tuning the improved algorithm and in Section 6.6 we seek to verify the theoretical running time and test how well the algorithm performs in a practical setting. We also illustrate the robustness properties of the halfspace median in Section 6.7. Finally, the chapter is concluded in Section 6.8.

6.1 The naive algorithm

The halfspace median is simply the point with maximum halfspace depth as formalized in Definition 1.3. This definition immediately proposes a naive algorithm that computes the depth of any interesting point and maximizes the depth. The notion of interesting points is clarified below and ensures that it suffices to check a finite number of points in the plane.

The algorithm is denoted `MAX_HS_NAIVE` and computes a point at maximum depth by forming all possible lines from any two points from the input P using two nested loops. With these lines, the algorithm computes all possible intersection points q, using two additional nested loops, and determines $\mathrm{D}_{hs}(P, q)$ by calling `QUERY_HS_NAIVE`. The point q is an interesting point and the resulting median is the one with maximum depth. This is a very simple algorithm but as Theorem 6.1 shows, it is extremely slow which makes it useless in practice.

Theorem 6.1. Given a set $P \subset \mathbb{R}^2$ of size n, `MAX_HS_NAIVE` computes $M_{hs}(P)$ in $O(n^6)$ time and uses $O(n)$ space.

Proof. First we argue that the algorithm is correct. To see that the interesting points represent the only possible solutions, we consider the arrangement of

lines built from the $O(n^2)$ lines going through any two input points. Clearly, every point inside a face of this arrangement must have equal depth, i.e. the depth can only change by crossing a boundary of a face. Furthermore, any two points on an edge in the arrangement trivially has the same depth as they lie on the same line. Combining these two facts, it suffices to check the depth of the vertices of the arrangement because they correspond to all possible depths. Since the intersections of the $O(n^2)$ pairs of lines corresponds to the $O(n^4)$ vertices in the arrangement, the above algorithm is correct.

In terms of running time, we know that the depth of a point can be calculated in $O(n^2)$ time by Theorem 3.3. Since there are n input points, the algorithm forms $O(n^2)$ lines which results in $O(n^4)$ intersection points. Hence, the running time of the algorithm is $O(n^6)$ in total. Looking at the space consumed, the algorithm iteratively stores a single pair of lines in memory along with the corresponding intersection point. Seeing that this is constant and the input points must be stored at all times, we can conclude that the algorithm uses $O(n)$ space. □

From a practical point of view, the naive algorithm is useless and the only real advantage lies in the simplicity of the implementation. Given algorithms that can compute the depth $\mathrm{D}_{hs}(P, q)$, the algorithm is straight forward and does not introduce any interesting implementation details. An obvious optimization may be applied by using `QUERY_HS_SORT` as a subroutine instead of `QUERY_HS_NAIVE` and reduces the running time to $O(n^5 \log n)$, though this has little effect on its practical use.

6.2 The level algorithm

The main motivation for improving on the naive algorithm is the horrifying running time. For a small input consisting of 100 points, the algorithm executes in the order of $100^6 = 10^{12}$ instructions. This is considered infeasible on most systems, but improvements can be achieved by more sophisticated techniques.

We presented several algorithms that improve on the naive algorithm in Section 1.1 in the introduction. The best proposal is an optimal randomized algorithm by Chan [A22] which runs in $O(n \log n)$ in $\mathbb{R}^2$. Unfortunately, the algorithm is conceptually difficult and Chan suspects that it has very large constants. In the following we describe a slightly simpler algorithm that has a larger asymptotic bound, but still improves over the naive solution and some of the referenced algorithms from Section 1.1.

Consider the slightly simpler problem of determining whether a point of halfspace depth $k + 1$ exists. By dualizing the input P into a set of lines L, this problem is equivalent to determining whether there exists a separating hyperplane between the upper and lower k-level in the arrangement of lines induced by L. This is a classic optimization problem and may be solved by the

following minimization

$$\begin{aligned} \arg\min_{a,b} \quad & ax + b \\ \text{subject to} \quad & ax_i + b \leq y_i \qquad \forall (x_i, y_i) \in U_k \\ & y_i \leq ax_i + b \qquad \forall (x_i, y_i) \in L_k \end{aligned} \tag{6.1}$$

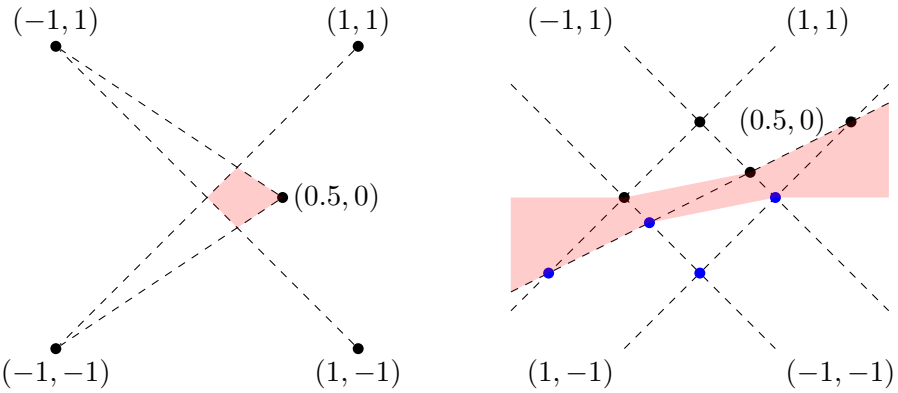
where U_k and L_k denotes the set of vertices on the upper and lower k-level respectively.

The correctness of the dual problem follows from Definition 2.1, which states that the dual transform is incidence and order preserving. Assume that there exists a point $p = (x, y)$ with halfspace depth $k + 1$ in the primal plane. This means that all halfspaces going through p has at least $k + 1$ points on both sides of it. Note that we can safely ignore the vertical halfspace because there always exists a non-vertical halfspace infinitely close to it resulting in the same depth. The point p dualizes to a line $l = (x, -y)$ and the halfspaces dualizes to points on l by the incidence preserving property. By the order preserving property, any point on l must have at least $k + 1$ lines above and below it. Note that l cannot be a vertical line, thus above and below are well defined. Hence, deciding whether there exists a point with halfspace depth $k+1$ or not, is equivalent to asking whether there exists a line in dual space where all points on it have least $k+1$ lines above and below them. By Definition 2.2, all points on a separating line between the upper and lower k-level vertices have $k + 1$ lines above and below them, since the k-level is a contour consisting of lines from the input. Thus, the decision problem reduces to finding such a separating line which the minimization problem 6.1 expresses. An example of the relation between the primal and dual problem can be seen in Figure 6.1.

The problem of finding the halfspace median simply requires a search in $k+1$ and an efficient way of finding k-level vertices. Unfortunately, the number of vertices on the k-level may be super-linear as stated in Lemma 2.1. For simplicity, we will assume that the size of the k-level is $O(n)$ which is not an unfair assumption in practice. We will return to the validity of this assumption in Section 6.6.

The level algorithm denoted `MAX_HS_LEVEL` is inspired by the idea sketched above and computes the halfspace median by a binary search in k which ranges from 0 to $\lceil \frac{n}{2} \rceil$. The algorithm follows a divide and conquer scheme. It takes a parameter r, which defines the size of the cuttings created, and a parameter b, which defines the size of the base case, and builds a cutting tree with fanout $O(r^2)$. Initially, the input P is dualized into a set of lines L using equation 2.1 and for each value of k, the root of the cutting tree is built using a $\frac{1}{r}$-cutting on L using a specialized version of `CUTTING_FIX`. The cutting is special in the sense that triangles are discarded when they do not contain vertices from the upper or lower k-level, i.e. the algorithm prunes children that cannot contain any constraints.

Assume that the cutting algorithm encounters a valid triangle Δ_i that contains vertices from the upper or lower k-level. If the crossing number is larger than the base case size the algorithm performs an inductive step and creates



(a) The primal problem defined on a set of points. The region at maximum halfspace depth is marked as a red convex hull and is bounded by the lines going through any two input points. Any point in this convex hull is a halfspace median with depth 2.

(b) The dual problem defined on a set of lines. The region that separates the vertices on the upper 1-level (black) and the vertices on the lower 1-level (blue) is marked red. Any line in this region dualizes to a halfspace median with depth 2.

Figure 6.1: The primal and dual interpretation of the same halfspace median problem.

a new $\frac{1}{r}$-cutting on the crossing lines. Otherwise, the base case is encountered and the vertices on the upper and lower k-level are recorded. This is done by computing the intersection points of the lines crossing Δ_i, checking whether these intersection points are inside Δ_i and determining their level by counting the number of lines strictly above and below. For this purpose the cutting algorithm records the number of lines strictly above and below Δ_i when computing the crossing number. We denote the lines strictly above Δ_i as a_i and the lines strictly below Δ_i as b_i.

The procedure above provides two sets of constraints, one consisting of the upper k-level vertices and another consisting of the lower k-level vertices. The minimization problem 6.1 is solved using a randomized incremental linear programming algorithm as described in [A13]. If the program is not infeasible, the algorithm retries the entire procedure with a larger value of k defined by the binary search. Otherwise, a lower value of k is chosen. When the search stops the resulting depth $k+1$ is returned together with the point $p = (a, -b)$ at maximum halfspace depth.

Theorem 6.2. Given a set $P \subset \mathbb{R}^2$ of size n, `MAX_HS_LEVEL` computes $M_{hs}(P)$ in $O(n^{1+\epsilon})$ time and uses $O(\max(r^2 \log_r n, n))$ space.

Proof. Clearly, the algorithm is correct by the construction explained above and by the correctness proof of the linear programming algorithm presented in [A13]. What remains is to prove the running time and space consumed.

In terms of running time, the binary search enforces a $O(\log n)$ slowdown. For each k in the search, a pruned cutting tree is built and a linear program is solved, hence the slowest of these two procedures will dominate the total

running time, along with the binary search. The linear program takes expected $O(n)$ time because there are $O(n)$ constraints. The proof is presented in [A13]. Building the cutting tree requires us to solve the recurrence

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ c_1 n r^2 + c_2 r T(\frac{n}{r}) & \text{otherwise} \end{cases}$$

where $b = 1$. In the base case the problem can be solved in constant time. Otherwise, the algorithm builds a $\frac{1}{r}$-cutting in $O(nr^2)$ time by Theorem 5.2 and calls recursively on any triangle that contains vertices from the upper or lower k-level. For this purpose we introduce another assumption that may be absurd in the worst case but is fair in practice, namely that $O(r)$ triangles contain vertices from the k-level. By unrolling the recurrence we get the following sum

$$\begin{aligned}
T(n) &= c_1 n r^2 + c_2 r T\left(\frac{n}{r}\right) \\
&= c_1 n r^2 + c_2 r \left(c_1 n r + c_2 r T\left(\frac{n}{r^2}\right)\right) \\
&= c_1 n r^2 + c_1 c_2 n r^2 + c_2^2 r^2 T\left(\frac{n}{r^2}\right) \\
&= c_1 n r^2 + c_1 c_2 n r^2 + c_2^2 r^2 \left(c_1 n + c_2 r T\left(\frac{n}{r^3}\right)\right) \\
&= c_1 n r^2 + c_1 c_2 n r^2 + c_1 c_2^2 n r^2 + c_2^3 r^3 T\left(\frac{n}{r^3}\right) \\
&= c_1 n r^2 + c_1 c_2 n r^2 + c_1 c_2^2 n r^2 + c_2^3 r^3 \left(c_1 \frac{n}{r} + c_2 r T\left(\frac{n}{r^4}\right)\right) \\
&= c_1 n r^2 + c_1 c_2 n r^2 + c_1 c_2^2 n r^2 + c_1 c_2^3 n r^2 + c_2^4 r^4 T\left(\frac{n}{r^4}\right) \\
&\vdots \\
&= c_1 n r^2 + c_1 c_2 n r^2 + c_1 c_2^2 n r^2 + c_1 c_2^3 n r^2 + \ldots + c_1 c_2^{\log_r n - 1} n r^2 + c_2^{\log_r n} r^{\log_r n}.
\end{aligned}$$

The recurrence bottoms out after $\log_r n$ steps because $\frac{n}{r^{\log_r n}} = \frac{n}{n} = 1$ and the base case takes constant time to solve. Continuing the derivation we can simplify the term by

$$\begin{aligned}
T(n) &= c_1 n r^2 + c_1 c_2 n r^2 + \ldots + c_1 c_2^{\log_r n - 1} n r^2 + c_2^{\log_r n} n \\
&= c_1 c_2^{\log_r n} n r^2 \left(\frac{n}{c_2^{\log_r n} n} + \frac{c_2 n}{c_2^{\log_r n} n} + \ldots + \frac{c_2^{\log_r n - 1} n}{c_2^{\log_r n} n} + \frac{1}{c_1 r^2}\right) \\
&= c_1 c_2^{\log_r n} n r^2 \left(\frac{1}{c_2^{\log_r n}} + \frac{1}{c_2^{\log_r n - 1}} + \ldots + \frac{1}{c_2} + \frac{1}{c_1 r^2}\right) \\
&= O\left(c_2^{\log_r n} n\right).
\end{aligned}$$

Note that all but the last term inside the parenthesis is the geometric series obtained by multiplying by $\frac{1}{c_2}$, which is less than 1, hence it converges towards a constant. The last term is also a constant. Thus the parenthesis is dominated by the $c_1 c_2^{\log_r n} n r^2$ term.

At this point, we know that c_2 is a constant given by the cutting algorithm, but we are free to choose r as we want. Thus, by defining $\epsilon = \frac{1}{\log_{c_2} r} > 0$ we can determine the bound on the running time as

$$T(n) = O\left(c_2^{\frac{\log_{c_2} n}{\log_{c_2} r}} n\right) = O\left(n^{\frac{1}{\log_{c_2} r}} n\right) = O\left(n^{1+\epsilon}\right).$$

This results in a running time of $O(n^{1+\epsilon} \log n)$ because $\epsilon > 0$, i.e. building the pruned cutting tree takes more time than solving the linear program. In total the bound becomes $O(n^{1+\epsilon'})$ because there exists an $\epsilon' > 0$ that removes the $\log n$ factor.

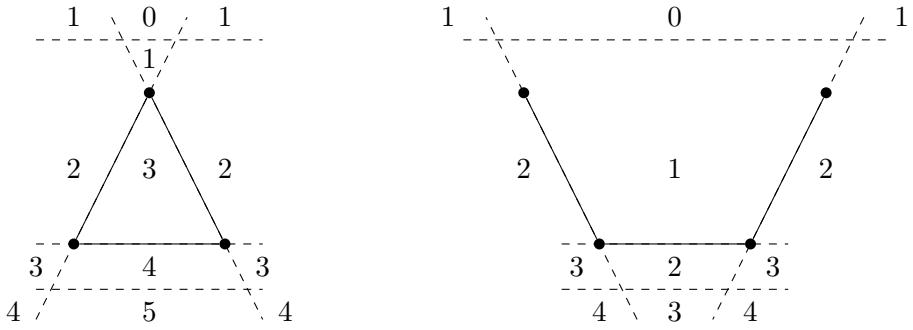
In terms of space, the algorithm needs to dualize the input and keep it in memory which uptakes $O(n)$ space. The linear program does not change the bound as we assume there are $O(n)$ constraints. Finding the constraints using the pruned cutting tree only requires the initial cutting along with one path from the root to a leaf. We already know that the height is $O(\log_r n)$, hence this uptakes $O(r^2 \log_r n)$ space. Depending on how r is chosen, the space complexity becomes $O(\max(r^2 \log_r n, n))$. □

6.3 Details of the level algorithm

The algorithm `MAX_HS_LEVEL` is in theory relatively simple, it is a binary search wherein constraints are found and a linear program is solved, but it relies on several other algorithms that may cause problems. The cutting algorithm for example, produces open triangles that counts the number of lines that intersect the interior of the triangle in order to avoid degeneracies. From the perspective of the level algorithm, this is insufficient because vertices on the upper and lower k-level may be on such border lines, i.e. the algorithm needs closed triangles. To incorporate this in the implementation, the algorithm keeps track of the border lines in a set separate from the lines crossing a triangle. These border lines are not considered when refining triangles but are considered when handling the base case of `MAX_HS_LEVEL`. This potentially adds a given constraint several times since triangles share border lines. We will investigate the practical number of constraints in Section 6.6.

A vital detail that ensures efficiency of the algorithm is the procedure that prunes triangles that does not contain vertices from the upper and lower k-level. This check is relatively straight forward but introduces a subtle special case that must be handled in order to ensure correctness. Ignoring the lower k-level vertices, a triangle Δ_i cannot be pruned if the upper k-level vertices are not above and not below the triangle. The first condition holds when $k \geq a_i$ and the second condition holds when $n - b_i \geq k$ and can easily be verified using the illustration in Figure 6.2a. In this case the upper 0-level vertices are above the triangle, because $0 \geq 1$ does not hold, and the upper 5-level vertices are below the triangle, because $5 - 1 \geq 5$ does not hold. The remaining upper level vertices intersect the triangle and is consistent with the two conditions. The special case is depicted in Figure 6.2b where the triangle Δ_i is a four

sided unbounded triangle with both of its hybrid edges pointing upwards. The structure of this triangles pushes the upper 4-level vertices down below the triangle causing the second condition to fail. In order to remedy the error, this case is handled by an extended condition where the level is incremented $n - b_i \geq k + 1$. A symmetric argument handles the first condition for four sided unbounded triangles with both of its hybrid edges pointing downwards, where the check becomes $k - 1 \geq a_i$. Furthermore, checking whether a triangle contains vertices from the lower k-level is done by exchanging a_i with b_i in the first condition and b_i with a_i in the second condition.



(a) The general case. (b) The special case.

Figure 6.2: Checking whether a triangle may be pruned.

The careful reader may notice that the pruning check does not test whether points with $k - 1$ lines strictly above or below them are contained in a given triangle. By Definition 2.2 these vertices are a part of the upper or lower k-level and must be considered. Such a check is unnecessary due to the construction of the cutting algorithm, because it does not create new vertices. Hence, a vertex with $k-1$ lines strictly above it will be included in a triangle with a vertex with k lines strictly above it. If a line at some point separates the two, it means that their upper and lower level changes which contradicts that their levels are $k-1$ and k respectively. The vertices in the upper and lower k-level are therefore correctly determined seeing that the base case procedure includes vertices with upper and lower level k and $k - 1$.

A rather large implementation detail is the linear programming algorithm. In general, the algorithm is straight forward given that the linear program is not unbounded. Fortunately, that is easy to verify because we are minimizing the slope of a line which is unbounded if and only if the upper and lower k-level can be separated by a vertical line. When that is not the case, the initial solution may be determined by choosing a constraint from both k-levels. Having permuted the constraints, they may be checked one by one against the current solution. In case a constraint c_i causes a violation the algorithm initiates a one-dimensional search on c_i - a general property of linear programming. The result of this search has two outcomes: Either the program is infeasible or a new solution is found. The algorithm determines this by bounding the solution using c_i and the previous constraints $c_1, \ldots, c_{i-1}$. In our case, this boils down to a case analysis depending on three binary values. The first two values depend

on whether c_i and the previous constraint c_j belongs to the upper or lower k-level. The third value depends on which of the two constraints has the largest x-coordinate. In total, this produces $2^3 = 8$ cases where 4 bounds the minimum slope of the solution and the remaining 4 bounds the maximum slope of the solution. For the sake of completeness we present two cases that bounds the minimum and maximum slope of the solution in Figure 6.3.

(a) Two constraints on the upper k-level bounding the maximum slope of the solution.

(b) Two constraints on the lower k-level bounding the minimum slope of the solution.

Figure 6.3: Bounding the solution during a one-dimensional search.

6.4 Testing and robustness

We have proven that the algorithms presented in the previous sections are correct. Even though these proofs ensure theoretical correctness it still makes sense to test the algorithms in order to verify their implementation. Like the arrangement and cutting algorithms, automatically asserting that the returned halfspace median is correct on any input is basically impossible. Instead, we made a select suite of tests that verifies the result on a fixed input along with an automatic test that compares their results on random input. Unfortunately, the naive algorithm is so slow that it restricts the test to relatively small inputs. None the less, it does provide a foundation for arguing that both algorithms are correctly implemented and calculate the halfspace median.

In regards to robustness, the naive algorithm is basically only affected by the numerical issues of the corresponding depth algorithm used to calculate the depth of a point. The level algorithm on the other hand inherits all the problems associated with the cutting and the arrangement algorithms, which occasionally results in infinite loops. The additional layers on top of the cutting algorithm does not introduce any new interesting numerical or robustness issues. At this point, infinite precision libraries are not really an option. The constants are so big that the running time degrades in such a degree that even the smallest inputs are infeasible in practice.

6.5 Fine-tuning the level algorithm

The algorithm `MAX_HS_LEVEL` depends on two hyper-parameters that must be supplied by the caller, namely r and b. The value of r controls the size of the cutting which is built in every inductive step and b defines the base case which

is executed whenever less than b lines cross a triangle. There seems to be no obvious values for these parameters and in this section we will reason about them and attempt to determine their optimal values.

The value of r is the subject of the first experiment, which consists of computing the halfspace median of 4000 random normal distributed points while measuring the running time. The base case size b is kept fixed and multiple runs are performed for each value of r. The result is depicted in Figure 6.4.

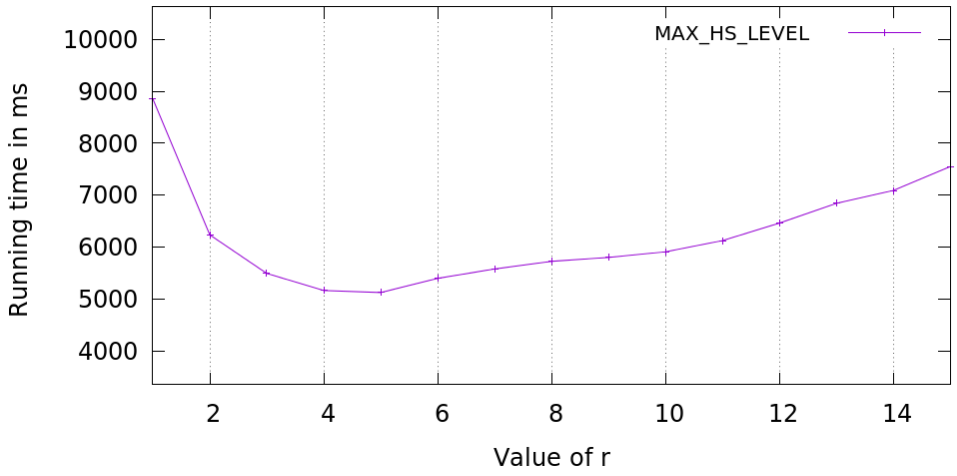


Figure 6.4: The running time in ms when calculating the halfspace median.

Since r directly affects the number of generated triangles in the cutting, and these triangles are used to prune disjoint areas that does not contain any vertices from the k-level, the more triangles that can be pruned the better. Additionally, a sufficiently large r will produce a very shallow cutting tree where almost everything except the k-level vertices can be pruned in the first step and no additional refinements are needed. Following this logic, the value of r must be large enough for the algorithm to efficiently prune large parts of the input space such that the least number of inductive steps are performed. However, this is accompanied by a large penalty because each triangle Δ_i requires $O(n)$ work in order to determine the crossing number, a_i and b_i which will degrade the running time if the cutting creates too many triangles. Based on this, it is most reasonable to assume that a sweet spot for r exists where the number of triangles is relatively low in order to prevent this penalty.

Inspecting Figure 6.4 it is apparent that this sweet spot occurs around $r = 5$. As expected, the spot takes place for a relative low value of r where the corresponding number of triangles being tested for containing k-level vertices is small. After $r = 5$, the penalty caused by calculating the crossing number, a_i and b_i for each triangle dominates the running time and outgrows the performance gained by building a more shallow cutting tree. For small values of r, we also experience that the running time degrades because a deeper cutting tree is built. Again, the bottleneck is caused by iterating trough a subset of lines for each triangle to calculate the crossing number. For very small values of r

the price gained on a single level is simply dominated by the many extra levels produced since a minimal part of the input space can be pruned for each new cutting. Worth noting is that the building time of the cutting is minuscule compared to the worst-case linear cost for each created triangle. Thus, a large fan out of the cutting tree is rarely preferable since the running time will degrade because of the many triangles created on each level. Basically, this experiment tells us that a relatively small value of r is better because that results in the least number of triangles without generating a too large cutting tree.

Having established an understanding of how r affects the algorithm, we move on to experimenting with how the size of the base case affects the running time. The second experiment performed is identical to the first with a single exception, namely r is a fixed value and the base case size b is varied. The result is displayed in Figure 6.4.

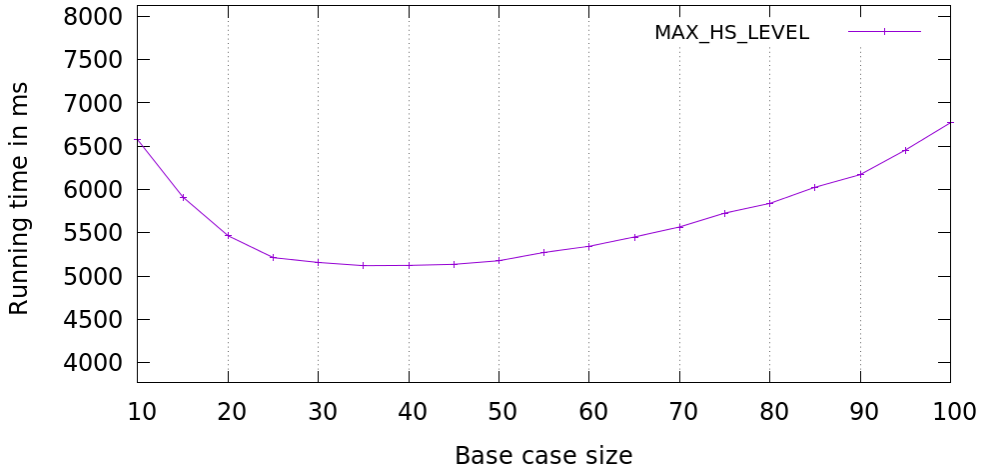


Figure 6.5: The running time in ms when calculating the halfspace median.

The value of b determines when the number of crossing lines Δ_c associated with a triangle becomes constant. Once the base case is reached, an $O(\Delta_c^3)$ algorithm is applied in order to compute the k-level vertices. Since the running time of this naive base case algorithm is cubic, running it for a large set of crossing lines is not desirable. On the contrary, a large base case causes fewer levels in the cutting tree. Based on these observations, it makes sense to expect a sweet spot for b where the size of the cutting tree is relatively low and the base case algorithm is not executed on a too large set of lines.

From Figure 6.4 there is an evident optimal base case size at around $b = 40$. For smaller values of b, the main reason for the slowdown is the fact that additional levels in the cutting tree causes more triangles to be created which we know is expensive based on the previous experiment. Continuously increasing the value of b does reduces the size of the produced cutting tree, but after the sweet spot the time spend on solving the base case starts outweighing the gain from the smaller cutting tree. One might suspect that cache misses influences the optimal value of b, but that does not appear to be the case. A line consumes

24 bytes which means that $24 \cdot 40 = 0.96$ KB's are needed to store the store lines when $b = 40$. Seeing that the L1 data cache is 32 KB and the running time starts degrading at $b = 50$, we suspect that the structure of the cutting tree and number of triangles created dominates the running time.

6.6 Verification of the running time

From a theoretical point of view it is safe to assume that `MAX_HS_LEVEL` is the superior algorithm when compared with `MAX_HS_NAIVE`. The objective of this section is to support this assumption along with verifying that the theoretic running time is sound and determine which value of ϵ applies in practice.

Before investigating the practical running time of the level algorithm, a slight detour is required. The complexity arguments depend on the assumption that the k-level is linear in size. Since this is a conjecture, there is no theoretic guarantee that it holds and the number could dominate the running time resulting in a slower practical bound. A proof of the conjecture is beyond the scope of this thesis, instead we seek to verify the conjecture empirically by measuring the number of generated constraints. The experiment consists of computing the halfspace median for an increasing number of normal distributed points and counting the number of generated constraints. For each possible n, multiple runs are performed and the average number of generated constraints is calculated. The result of this experiment is plotted in Figure 6.6.

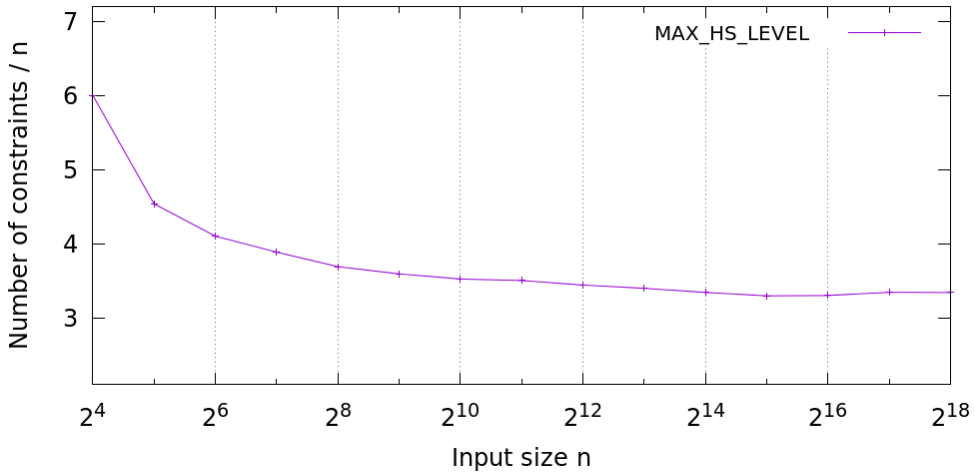


Figure 6.6: The number of constraints gathered when calculating the halfspace median.

Looking at Figure 6.6 we see that the number of constraints converges towards a constant factor when divided by n. If the size of the k-level depended on k, as stated by the best known upper bound in Lemma 2.1, we would expect to see a gradual increase because k gets bigger alongside the input size n. Hence, we conclude that the k-level is linear in this practical setting. Furthermore, the fact that the constant is 3 strongly indicates that the generated

constraints does not contain too many duplicates. We mentioned earlier that this was a potential problem, but seeing that both the lower and upper k-level is accounted for, the total number of constraints generated for both levels is close to linear in practice.

Regarding the second assumption introduced in the proof of Theorem 6.2, where we assume that the k-level vertices are contained in $O(r)$ triangles, we measured the average number of recursive steps on each level of the cutting tree during the above experiment. The number stabilizes as the input reaches a substantial size and the only thing that changes is the number of levels in the cutting tree, which is natural seeing that n increases.

Based on the empirical evidence of the assumptions made so far, we seek to verify the theoretic running time and determine which value of ϵ applies in practice. The setup of this experiment is identical to the former, except that we measure the running time instead of the number of constraints. The result is depicted in Figure 6.7.

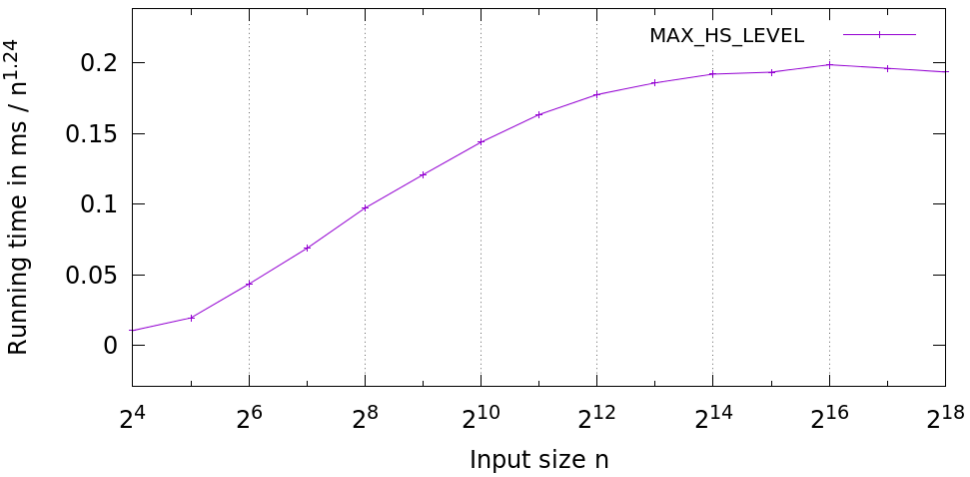


Figure 6.7: The running time in ms when calculating the halfspace median.

Trying different values of ϵ we found that $\epsilon = 0.24$ is close to the smallest value where the running time converges. This concludes that the theoretic bound is valid and relatively tight. Surprisingly, this makes the $O(n \log n)$ optimal solution proposed by Chan not as far fetched as one might suspect. Unfortunately, we were not able to test the algorithm on larger inputs than $n = 2^{18}$ because the algorithm loops indefinitely from time to time due to numeric instability. This is one of the main disadvantages of the algorithm and is a natural problem in relation to its practical use.

In order to make sense of the constants hidden in the asymptotic notation, we present the running times in Table 6.1 along with that of the naive algorithm. Given the relatively of all things, it is impossible to conclude whether this is efficient or not, but the speedup is significant. For an input of size $n = 2^{18} = 262144$ the algorithm finds the halfspace median in around 16-17 minutes, which is not that bad given the severity of the problem.

n	MAX_HS_NAIVE	MAX_HS_LEVEL	n	MAX_HS_LEVEL
16	43.526	0.328	4096	5349
32	3021.380	1.431	8192	13 225
64	193 854.000	7.537	16 384	32 279
128		28.159	32 768	76 795
256		94.099	65 536	186 317
512		276.237	131 072	434 505
1024		776.812	262 144	1 012 390
2048		2084.340		

Table 6.1: The running time in ms when calculating the halfspace median.

In relation to the question of efficiency there are some optimizations that could improve the running time of the algorithm. Seeing that the cutting produces a disjoint partition of the plane and queries this partition, the algorithm is suited for parallelization. The triangles can simply be processed in parallel which should result in a speedup on a multiprocessor systems.

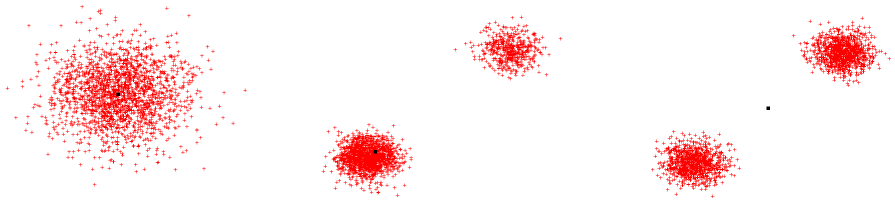
Furthermore, the base case procedure is a cubic algorithm that computes the level of intersections between crossing lines of a triangle. This problem can be solved in quadratic time by constructing the arrangement induced by the lines inside the triangle and process the vertices on each line from left to right. This can be done by finding the leftmost intersection of a line l with the bounding box, compute its level in linear time and traverse the remaining vertices on the line. Depending on the slope of the line l_i creating a vertex on l the next level can be determined in constant time, i.e. if l_i has a larger slope the level increases and if l_i has a smaller slope the level decreases.

Another suggestion is based on sacrificing some space in return of time. The level algorithm has very good space complexity, which is linear in most cases because the pruned cutting tree is rebuilt for each value of k, meaning that the entire tree is never stored explicitly. We can avoid this either by building the entire cutting tree prior to the binary search, or simply extend the tree on demand whenever triangles that contain k-level vertices needs to be handled. This should result in an improvement since a lot fewer cuttings are created, but increases the space complexity to $O(n^{2+\epsilon})$ in the worst case.

Finally a trivial speedup can be achieved by uniformly sampling the input as was done in Chapter 3 for single points. The theory of ϵ-approximations should still apply seeing that the median is the point with maximum halfspace depth, hence we would expect approximations to produce very good results even though a very small sample is used.

6.7 Verification of the breakdown point

In the previous section we verified the theoretic bound of `MAX_HS_LEVEL` and saw that $\epsilon = 0.24$ seems to be realistic in practice. With a working median algorithm we are able to visualize the result of our endeavours and may return to Definition 1.1 and verify the robustness properties of the halfspace median. Recall that the theoretic breakdown point is $\frac{1}{3}$ in $\mathbb{R}^2$, thus we would expect the median to deviate from the true center of the data cloud when at least $\frac{1}{3}$ of the data points are noisy. In the extreme case, the noise is a dense cluster which pulls the median away from its center in a single direction. We present three such examples by running the level algorithm on 2000 normal distributed points with varying levels of noisy points, which is shown in Figure 6.8. These examples contain 0, $\frac{n}{3}$ and $\frac{n}{2}$ noisy data points respectively, and we see that the noise gradually affects the median and consequently causes it to be right between the two clusters. These examples show that the halfspace median is indeed very robust and the algorithm works as expected.



(a) 0 noisy data points. (b) $\frac{n}{3}$ noisy data points. (c) $\frac{n}{2}$ noisy data points.

Figure 6.8: Examples of the halfspace median with 0, $\frac{n}{3}$ and $\frac{n}{2}$ noisy data points respectively.

6.8 Conclusion

In this chapter we have studied the problem of calculating the halfspace median and verified its robustness against outliers. The naive algorithm is very simple and easy to implement but the bound on the running time is a whopping $O(n^6)$ which makes it useless in practice. The level algorithm is an improved algorithm that reduces the running time to $O(n^{1+\epsilon})$ using various techniques such as geometric divide and conquer, arrangements of lines, $\frac{1}{r}$-cuttings, pruning, binary search and linear programming. This algorithm is in theory simple, but takes a lot of effort to implement due to the number of algorithmic tools it utilizes. The algorithm takes two hyper-parameters r and b and experiments illustrate that $r = 5$ and $b = 40$ is optimal and results in a running time of $O(n^{1.24})$. The constants still appear to be rather large seeing that it takes around 16-17 minutes to compute the halfspace median for 2^{18} points. Nevertheless, the improvement over the naive algorithm is substantial and the algorithm has great optimization potential. In conclusion, our solution makes it feasible to compute the halfspace median in $\mathbb{R}^2$, though the algorithm suffers from numeric instability which causes it to loop indefinitely on rare occasions.

Chapter 7

Conclusion

In this thesis we have studied the problem of computing the simplicial and halfspace depth in $\mathbb{R}^2$ and tested how sampling affects the result. We also studied the problem of computing the halfspace median in $\mathbb{R}^2$, using various computational techniques, and tested its efficiency and practical use.

We conclude that the simplicial and halfspace depth of a point can be determined in $O(n \log n)$ as presented in Chapter 3. This can further be improved by uniformly sampling the input within a tolerable error. Such an approximation works best for points deep in the data cloud, whereas the error of outlying points is larger. This does not pose a problem in practice as the error does not change the fact that outlying points has a low normalized depth and points deep inside the data cloud has a large normalized depth. The approximation method works well and can improve the algorithm for ranking a set of n points to a $O(nk \log k)$ algorithm where $k << n$.

In Chapter 4 we described a versatile implementation of an arrangement of lines, which is a necessary data-structure for computing the halfspace median efficiently. We conclude that this structure can be built in $O(n^2)$ time and uses $O(n^2)$ space and may be implemented by introducing the concept of a bounding box at infinity, such that bounded, semi-bounded and unbounded arrangements can be constructed. Unfortunately, the algorithm suffers from numeric instability which we were unable to resolve.

In Chapter 5 we described two algorithms for creating a $\frac{1}{r}$-cutting, which is used for doing geometric divide and conquer when computing the halfspace median. The better implementation creates a $O(r^2)$ size cutting in $O(nr^2)$ time and takes two hyper-parameters c_1 and c_2. Experiments shows that $c_1 = 1$ and $c_2 = 1.6$ is optimal and produces the fastest algorithm. Unfortunately, the cutting algorithm also suffers from the numeric stability issues, some of which are inherited from the arrangement algorithm.

In relation to the halfspace median studied in Chapter 6, we conclude that our algorithm solves the problem in $O(n^{1+\epsilon})$ in dual space using various algorithmic techniques. We found that optimal hyper-parameters are $r = 5$ and $b = 40$ and results in a running time of $O(n^{1.24})$. The implementation is not simple as such, but the ideas behind it are relatively straightforward. In terms of efficiency, the algorithm finds the halfspace median in around 16-17 minutes

on a point set of size 2^{18}, which is a great improvement over the naive implementation. The algorithm may be optimized for improved performance, but suffers from numeric instability caused by the arrangement and cutting algorithms. These instabilities causes the algorithm to loop indefinitely from time to time, which is a disadvantage in relation to its practical use.

7.1 Future work

The numeric instability issues mentioned above provides a basis for future research and solving these will naturally make the algorithm more useful in practice. We suspect that our approach of checking intersections, by reconstructing lines from segments, is numerically unstable and alternative intersection methods may prove to be more reliable. An obvious approach is to save a pointer on each edge to the line that created it and use its parameters in the intersect procedure. Our implementation already saves and id, but a pointer to the line itself may provide more information without increasing the space consumed. There may be other parts of the code that contribute to these errors and determining them may require a great deal of work.

The optimizations suggested for the improved algorithm in Section 6.6 also gives rise to future work. These optimizations and the main ideas behind the algorithm are not restricted to $\mathbb{R}^2$ and it could be interesting to implement a version that handles d-dimensional data in order to test its efficiency in higher dimensions.

A more theoretical line of work may be partaken by removing the assumption that the k-level is linear and that it is contained in $O(r)$ triangles of a cutting partition. This may require more sophisticated theoretic tools and we suspect that the gain is questionable in practice, at least in $\mathbb{R}^2$.

Bibliography

[A1] A. Weber. Uber den Standort der Industrien, Tubingen. English translation by C. Friedrich (1929): Alfred Webers theory of location of industries In *University of Chicago Press*, 1909.

[A2] B. Chazelle. Sampling. In *The Discrepancy Method*, 169–202, 2000.

[A3] D. Donoho and M. Gasko. Breakdown Properties of Location Estimates Based on Halfspace Depth and Projected Outlyingness. in *The Annals of Statistics 20*, 1803–1827, 1992.

[A4] D. Haussler and E. Welzl Epsilon-Nets and Simplex Range Queries. In *Proc. of the second annual symposium on Computational geometry*, 61–71, 1986.

[A5] G. Aloupis, S. Langerman, M. Soss and G. Toussaint. Algorithms for Bivariate Medians and a Fermat-Torricelli Problem for Lines. In *Comput. Geom. Theory Appl.* 26(1), 69–79, 2003.

[A6] H. Hotelling. Stability in Competition. In *Economic Journal 39*, 41–57, 1929.

[A7] J. Hayford. What is the Center of an Area, or the Center of a Population? In *Journal of the American Statistical Association 8*, 47–58, 1902.

[A8] J. Matoušek. Computing the Center of Planar Point Sets. In *Computational Geometry: Papers from the DIMACS special year (J. Goodman, R. Pollack, and W. Steiger, eds), American Mathematical Society*, vol 6, 221–230, 1991.

[A9] J. Matoušek. Number of Faces in Arrangements. In *Lectures on Discrete Geometry*, 2002.

[A10] J. Shewchuk. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. In *Discrete and Computational Geometry*, 18(3), 305–363, 1997.

[A11] J. Tukey. Mathematics and the Picturing of Data. In *Proc. Int. Congress of Mathematicians 2* 523–531, 1975.

[A12] M. Berg, O. Cheong, M. Kreveld and M. Overmars Arrangements and Duality. In *Computational Geometry, Algorithms and Applications*, 173–190, 2008.

[A13] M. Berg, O. Cheong, M. Kreveld and M. Overmars Linear Programming. In *Computational Geometry, Algorithms and Applications*, 63–93, 2008.

[A14] P. Afshani. On Approximate Simplicial Depth Queries, 2008.

[A15] P. Agarwal, B. Aronov, T. Chan and M. Sharir. On Level in Arrangements of Lines, Segments, Planes and Triangles. In *Discrete Comput. Geom.*, 19:315–331, 1998.

[A16] P. Rousseeuw. Multivariate Estimation with high Breakdown Point. In *Mathematical Statistics and Applications (Dordrecht)*, vol. B, 283–297, 1985.

[A17] P. Rousseeuw and H. Lopuhaa. Breakdown Points of Affine Equivariant Estimators of Multivariate Location and Covariance Matrices. In *The Annals of Statistics*, vol. 19, no. 1, 229–248, 1991.

[A18] P. Rousseeuw and I. Ruts Bivariate Location Depth. In *Applied Statistics*, vol. 45, no. 1, 516–526, 1996.

[A19] P. Rousseeuw and I. Ruts. Constructing the Bivariate Tukey Median. In *Statistica Sinica 8*, 828–839, 1998.

[A20] R. Liu On a Notion of Simplicial Depth. In *Proc. Natl. Acad. Sci*, vol. 85, 1732–1734, 1988.

[A21] S. Langerman and W. Steiger. The Complexity of Hyperplane Depth in the Plane. In *Japan Conference on Discrete and Computational Geometry*, November 2000.

[A22] T. Chan. An optimal Randomized Algorithm for Maximum Tukey Depth. In *Proc. 15th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 423–429, 2004.

[A23] T. Dey. Improved Bounds on Planar k-sets and Related Problems. In *Discrete Comput. Geom.*, 19:373–382, 1998.

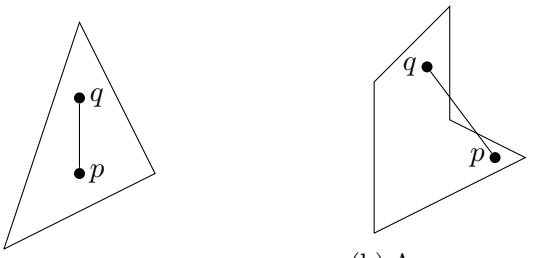
[A24] Z. Chen. Bound for the Breakdown Point of the Simplicial Median. In *Journal of Multivariate Analysis* 55, 1–13, 1995.

Appendices

7.2 Convex sets

In this section we recap the basic definition of convex sets and proof a simple proposition about convex combinations which is used in Theorem 2.4.

A convex set is a set where the line between any two points in the set does not cross the boundary. Examples of a convex and non-convex set are depicted in Figure 7.1a and 7.1b. Formally, convex sets are defined by the set of all convex combinations of any two points in the set. The points in such a convex combination lies exactly on the line segment between the two points, which is in line with the intuition. This notion is formalized in Definition 7.1 and 7.2.



(a) A convex set.

(b) A non-convex set.

Figure 7.1: A convex and non-convex set.

Definition 7.1. Given $p, q \in P$ the set of all convex combinations of p and q is defined by:

$$\{(1-a)p + aq : 0 \leq a \leq 1\}.$$

Definition 7.2. The set P is convex provided that the set of all convex combinations of any two points $p, q \in P$ is a subset of P.

A property about convex sets is that any convex combination of k points from the set remains inside the set. In other words, convex sets are closed under linear combinations of points from the set where all coefficients are non-negative and sum to one. The proof of the statement is by induction and concludes this section.

Proposition 7.1. Let $p_1, \ldots, p_k$ be points in a convex set P and let $a_1, \ldots, a_k \geq 0$ be coefficients such that $\sum_{i=1}^{k} a_i = 1$ then

$$q = \sum_{i=1}^{k} a_i p_i \in P.$$

Proof. The base case $k = 1$ holds because $a_1 = 1$ and $q = p_1$ is already in P. Since P is convex the case $k = 2$ holds by Definition 7.2 and 7.1 because q is convex combination of two points from P. In the inductive step we assume that the statement holds for k and need to prove that the convex combination $q = \sum_{i=1}^{k+1} a_i p_i \in P$. We define the coefficient $a = \sum_{i=1}^{k} a_i$ and observe that

$$1 - a = \sum_{i=1}^{k+1} a_i - \sum_{i=1}^{k} a_i = a_{k+1}$$

which holds because q is convex combination so $\sum_{i=1}^{k+1} a_i = 1$. We now use this coefficient to rewrite the last part of q

$$q = \sum_{i=1}^{k+1} a_i p_i = \sum_{i=1}^{k} a_i p_i + a_{k+1} p_{k+1} = \sum_{i=1}^{k} a_i p_i + (1 - a) p_{k+1}$$

and rewrite the first part by multiplying and dividing by a

$$q = a \sum_{i=1}^{k} \frac{a_i}{a} p_i + (1 - a) p_{k+1}.$$

Observe that the coefficients $\frac{a_i}{a}$ are larger than or equal to zero and $\sum_{i=1}^{k} \frac{a_i}{a} = 1$ because of the way a is defined. By the induction hypothesis $\sum_{i=1}^{k} a_i p_i \in P$, which gives us that q is a convex combination of two points since $p_{k+1} \in P$, the coefficients a and $1 - a$ are larger than or equal to zero and sums to one $a + (1 - a) = 1$. The inductive step then holds by Definition 7.2 and 7.1. $\square$