Algorithms for the 2D ham sandwich problem

Graduation thesis
University of São Paulo
Bachelor in Computer Science, 2021

Giovanna Kobus Conrado
Advisor: Cristina G. Fernandes

Abstract

Given two disjoint sets P_1 and P_2 in $\mathbb{R}^2$, a two-dimensional *ham sandwich cut* is a line that bisects both P_1 and P_2 simultaneously. The *ham sandwich problem* consists of finding such a line. A linear-time algorithm to solve this problem was proposed by Chi-Yuan Lo, Jiří Matoušek and William Steiger [7]. We expand on their paper, detailing the steps needed to implement the algorithm, presenting pseudocode for most of the steps, and providing some of the knowledge needed to understand both the algorithm and why it works.

Keywords: Ham sandwich theorem, ham sandwich problem, computational geometry, duality, sorting networks

Contents

Chapter 1

Introduction

The first version of the problem that will be studied in this work was proposed by the Polish mathematician Steinhaus in 1938 in the popular mathematics publication *Mathesis Polska*. The problem inquired: "Is it always possible to bisect three solids, arbitrarily located, with the aid of an appropriate plane?," or, in its informal phrasing: "Can we place a piece of ham under a meat cutter so that the meat, bone, and fat are cut in half?"

The answer turned out to be positive, and from that informal analogy came the name for the more general version of the result: the *ham sandwich theorem*, that states that, given n objects in an n-dimensional Euclidean space, there is an $(n-1)$-dimensional hyperplane that divides them all in half (with respect to their measure) simultaneously.

In computational geometry certain particular cases have been studied. Generally, the goal is to find such hyperplane, and the task of doing so is called the *ham sandwich problem*. The problem has been more extensively studied for the particular case of when the objects are sets of points.

The problem mentioned has application on other computational geometry algorithms, such as finding the regression depth of a hyperplane with respect to a set of points, as described in [2].

In 1984 Meggido devised an algorithm, described in [9], to solve in linear time the two dimensional ham sandwich problem when the two sets of points are separated, i.e., when their convex hulls do not intersect.

A very important result for the problem followed: a paper published in 1994 in the *Discrete and Computational Geometry* journal, by Lo, Matoušek, and Steiger, which solves this discrete version of the problem in $\mathcal{O}(n^{d-1})$ time, where n is the total number of points and d is the number of dimensions of the space those points are in and also the number of sets of points. This paper heavily relies on multiple previous studies in sorting networks and how they can be applied to line arrangements, such as [8] and [4].

The algorithm described by Lo, Matoušek, and Steiger is quite intricate, so this work is mainly focused on explaining their linear algorithm for the two-dimensional case of the discrete version of the problem.

On Chapter 2, we will establish some basic ideas that will be needed to understand the approach used by Lo, Matoušek, and Steiger in [7]. On Chapter 3, we will describe in detail their linear algorithm, using a simpler algorithm to substitute the

one described in [8] to find a suitable line intersection given the problem constraints, resulting in an $\mathcal{O}(n \log n)$ algorithm. On Chapter 4, we will explain sorting networks, a structure that is needed for the linear algorithm. On Chapter 5, we will describe the steps proposed by Matoušek in [8] to make the previous algorithm linear. Finally, on Chapter 6, we will outline the final results and what we could draw from this work.

Chapter 2

The discrete ham sandwich problem

In this section we will establish some definitions needed to fully understand the problem and the algorithms that will be described. We will also formalize some of the notions that were mentioned in the introduction, including the definition of the problem itself.

All given sets of points and lines we consider are finite. We also consider all points and lines to be in general position.

2.1 Ham sandwich cuts

Definition 1. *A line ℓ is said to bisect the set P of points in $\mathbb{R}^2$ if no more than $\frac{|P|}{2}$ points of P lie on either of the open half-planes defined by ℓ (Figure 2.1).*

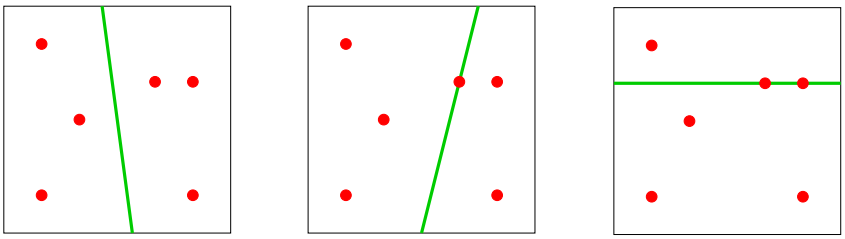


Figure 2.1: In all the images above, the line bisects the set of points.

Let the pair (x, y) represent a point and the equation $y = m \cdot x + b$ represent a line. Note that vertical lines do not have such representation. From now on, the term *line* always refers to non-vertical lines.

We can easily check if such a line bisects a set of points in linear time using Algorithm 1.

Algorithm 1 BISECTS(ℓ, P)

Input: a line ℓ and a set P of points
Output: TRUE if ℓ bisects P and FALSE otherwise

```
below ← 0
above ← 0
for p ∈ P do
    if p.y < ℓ.m · p.x + ℓ.b then below ← below + 1
    if p.y > ℓ.m · p.x + ℓ.b then above ← above + 1
return below ≤ |P|/2 and above ≤ |P|/2
```

Definition 2. *A ham sandwich cut for two sets P_1 and P_2 of points in $\mathbb{R}^2$ is a line that bisects both P_1 and P_2 simultaneously (Figure 2.2).*

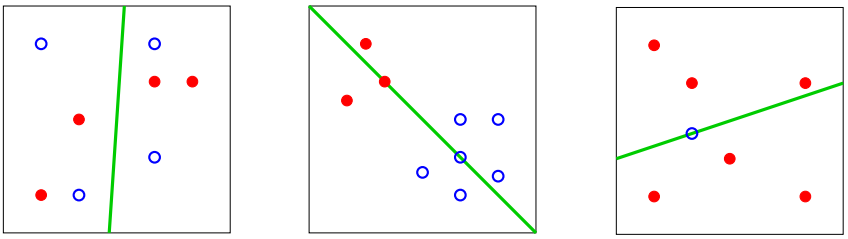


Figure 2.2: The points in red (filled) are in P_1 and the points in blue (not filled) are in P_2. The lines are ham sandwich cuts for P_1 and P_2.

Problem 1 (Two-Dimensional Discrete Ham Sandwich Problem)**.** *Given two sets P_1 and P_2 of points in $\mathbb{R}^2$, find a ham sandwich cut for P_1 and P_2.*

It is not intuitive, but such a cut always exists. In Section 2.3 we will show a constructive proof that uses some of the same ideas of the algorithm this work aims to describe. Although there are simpler proofs of the existence of a ham sandwich cut, they do not help with finding the cut itself and will not be mentioned.

We can make further observations on the problem that will help with its solution.

Proposition 1. *Let P be a set of points in $\mathbb{R}^2$ such that $|P|$ is odd and ℓ be a line that bisects P. There is a point in P that lies on ℓ.*

Proof. By definition, any line that bisects P must leave no more than $\frac{|P|}{2}$ points on either of its sides. Since the number of points that lie strictly on some side of ℓ is always an integer, that line must leave no more than $\left\lfloor \frac{|P|}{2} \right\rfloor = \frac{|P|-1}{2}$ points on either side. So we have a maximum of $|P|-1$ points that lie strictly on any side of ℓ. Thus, at least one point of P has to lie on ℓ. □

Proposition 2. *Let P be a set of points in $\mathbb{R}^2$ such that $|P|$ is even. For every point $x \in P$, any line that bisects $P \setminus \{x\}$ also bisects P.*

Proof. Since $|P|$ is even, $|P \setminus \{x\}|$ is odd. Let ℓ be a line that bisects $P \setminus \{x\}$. From Proposition 1, at least one point from $P \setminus \{x\}$ lies on ℓ. Therefore, at most $\frac{|P\setminus\{x\}|-1}{2} = \frac{|P|-2}{2}$ points lie on either side of ℓ. So, no matter where x lies, at most $\frac{|P|}{2}$ points will lie on either side of ℓ, which, by definition, implies that ℓ bisects P. □

From Proposition 2, we may assume without loss of generality that both P_1 and P_2 have an odd number of points. Otherwise, we can just remove any point from the even sets and a solution to the modified instance will also be a solution to the original instance. Now Proposition 1, along with the fact that a ham sandwich cut always exists, can give us a powerful observation: there always exists a ham sandwich cut that goes through at least one point from P_1 and one point from P_2. That gives us a naive $\mathcal{O}(n^3)$ algorithm (where $n = |P_1| + |P_2|$) briefly described ahead, to solve Problem 1.

Algorithm 2 goes through every point in P_1 and every point in P_2 and checks if the line that passes through both of these points is a ham sandwich cut. This is done by using the function BISECTS, that was defined in Algorithm 1. Since the points are in general position, there are no two points in P_1 and P_2 with the same x-coordinate. So any line defined by two points can be represented by an equation of the type $y = mx + b$.

Algorithm 2 NAIVESOLVE(P_1, P_2)

Input: two nonempty sets P_1 and P_2 of points in general position
Output: a ham sandwich cut for P_1 and P_2

1: **for** $p_1 \in P_1$ **do**
2: **for** $p_2 \in P_2$ **do**
3: $m \leftarrow \frac{p_2.y - p_1.y}{p_1.x - p_2.x}$
4: $b \leftarrow p_1.y - m \cdot p_1.x$
5: $\ell \leftarrow (y = mx + b)$ ▷ line that contains p_1 and p_2
6: **if** BISECTS(ℓ, P_1) **and** BISECTS(ℓ, P_2) **then return** ℓ

An optimization can be done to reduce the running time of Algorithm 2 from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^2 \log n)$. To do so, one can keep three lists: the list of all the pairs of points in $P_1 \cup P_2$, sorted by the slope of the line that contains them, the list of points in P_1 sorted by x-coordinate, and the list of the points in P_2 also sorted by x-coordinate.

The idea is that, every time we go through a potential ham sandwich cut, we will have the points in P_1 and P_2 sorted in the direction of the line perpendicular to that cut, so it is possible to check in constant time if the current cut is a ham sandwich cut or not.

Roughly, we process the pairs of points in order of slope and do the following: if the points in the current pair belong to the same initial set, we swap their positions in the corresponding list of points of that set. Otherwise we have a potential ham sandwich cut and we check if both points are in the middle of their respective lists.

The resulting complexity is that of sorting the list of all pairs of points, that is, $\mathcal{O}(n^2 \log n)$.

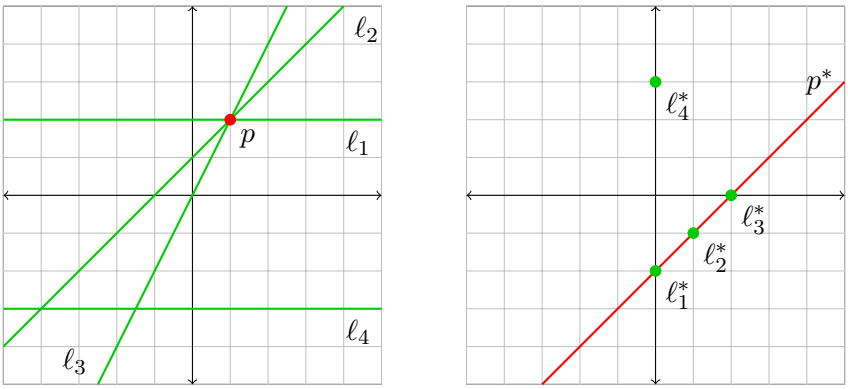
2.2 Duality

When dealing with lines and slopes, things can get cumbersome very quickly. It is sometimes helpful to be able to look at certain problems from a different perspective and the insights might be clearer. To help with that, we will now introduce the idea of plane duality.

A certain symmetry can be seen between points and lines in a plane. Some properties translate really well from points to lines and vice-versa. This is partially due to the fact that both of those structures can be described by two parameters: the points by their coordinates and the lines by their slope and intersection with the y-axis.

We want to find a mapping from points to lines and lines to points that translates well some properties. For instance, three points on a line would become three lines through a point. These mappings are called *duality transforms*. The image of an object under a duality transform is called the *dual* of that object.

The duality transform we will use is the following. Let $p := (p_x, p_y)$ be a point in the plane. The dual of p will be the line $p^* := (y = p_x \cdot x - p_y)$. The dual of a line $\ell := (y = m \cdot x + b)$ will be the point p such that $p^* = \ell$, which is $\ell^* := (m, -b)$ (Figure 2.3).



Primal plane Dual plane

Figure 2.3: An example of plane duality.

We are interested in certain properties that, if held in the primal plane, also hold in the dual plane.

Proposition 3. *A point p lies on a line ℓ if and only if the point ℓ^* lies on the line p^*.*

Proof. Let $p := (p_x, p_y)$ and $\ell := (y = m \cdot x + b)$. Then p lies on ℓ if and only if $p_y = m \cdot p_x + b$. We can rewrite that equation to be $-b = p_x \cdot m - p_y$, which means that $\ell^* := (m, -b)$ lies on $p^* := (y = p_x \cdot x - p_y)$. □

Proposition 4. *A point p lies above a line ℓ if and only if the point ℓ^* lies above the line p^*.*

Proof. Let $p := (p_x, p_y)$ and $\ell := (y = m \cdot x + b)$. Then p lies above ℓ if and only if $p_y > m \cdot p_x + b$. We can rewrite that equation to be $-b > p_x \cdot m - p_y$, which means that $\ell^* := (m, -b)$ lies above $p^* := (y = p_x \cdot x - p_y)$. □

Let us translate some definitions so that we can translate Problem 1 to its dual version. Firstly, we will define what is a point that is in the median zone of a set of lines, which is the dual of a line bisecting a set of points.

Definition 3. *A point p is said to be in the median zone of a set H of lines if there are no more than $\frac{|H|}{2}$ lines above it and no more than $\frac{|H|}{2}$ lines below it.*

Similarly to Algorithm 1, that checks if a line bisects a set of points, we can write a simple linear-time algorithm to check if a point p is in the median zone of a set H of lines.

Algorithm 3 MEDIANZONE(p, H)

Input: a point p and a set H of lines
Output: TRUE if p is in the median zone of H and FALSE otherwise

```
1: below ← 0
2: above ← 0
3: for h ∈ H do
4:     if p.y < h.m · p.x + h.b then below ← below + 1
5:     if p.y > h.m · p.x + h.b then above ← above + 1
6: return below ≤ |H|/2 and above ≤ |H|/2
```

We will also need to define a ham sandwich point, which is the dual of a ham sandwich cut.

Definition 4. *A ham sandwich point for two sets H_1 and H_2 of lines is a point that is in the median zone of H_1 and H_2 simultaneously (Figure 2.4).*

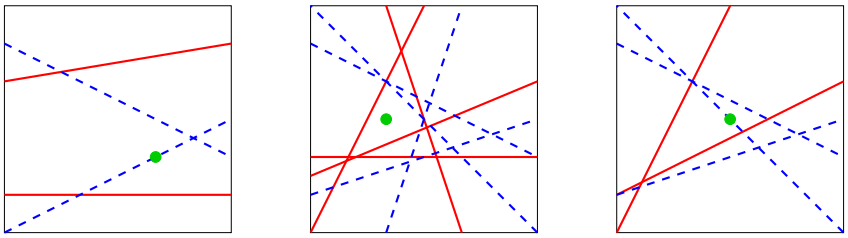


Figure 2.4: The lines in red (solid) represent H_1 and the lines in blue (dashed) represent H_2. The marked points are ham sandwich points for H_1 and H_2.

With that we can define the dual two-dimensional ham sandwich problem, which is the dual of Problem 1.

Problem 2 (Dual Two-Dimensional Ham Sandwich Problem)**.** *Given two sets H_1 and H_2 of lines in $\mathbb{R}^2$, find a ham sandwich point for H_1 and H_2.*

It can be proven that, if the lines are in general position, such a point always exists, so we would like to find it as efficiently as possible. If there are parallel lines, such point might not exist, which is the case for the examples in Figure 2.5. This is due to the dual conversion of the problem. This representation does not allow points that are the dual of vertical lines, and these lines might be the only solution for the primal problem.

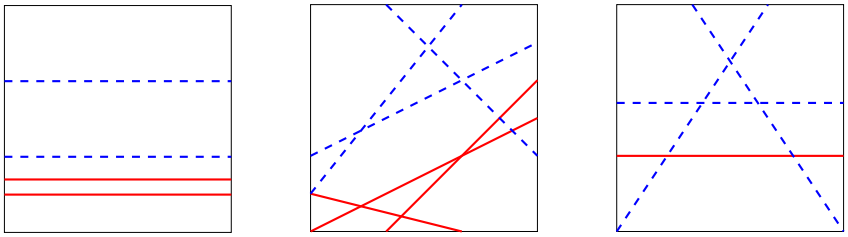


Figure 2.5: The lines in red (solid) represent H_1 and the lines in blue (dashed) represent H_2. There is no ham sandwich point for H_1 and H_2.

Proposition 5. *In general position, Problem 1 and Problem 2 are equivalent. In other words, a line ℓ is a ham sandwich cut for the sets P_1 and P_2 of points in general position if and only if the point ℓ^* is a ham sandwich point for the sets P_1^* and P_2^* of lines (Figure 2.6).*

Proof. By Proposition 4, a line ℓ bisects a set P of points if and only if ℓ^* is in the median zone of P^*. That way, a line ℓ is a ham sandwich cut for the sets P_1 and P_2 of points if and only if ℓ^* is a ham sandwich point for the sets P_1^* and P_2^* of lines. □

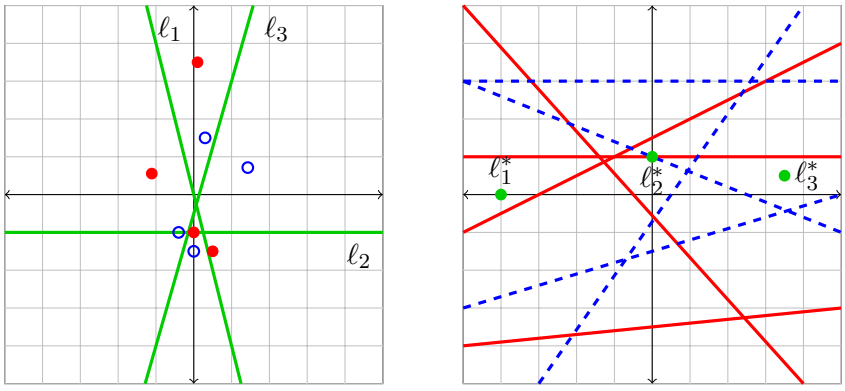


Figure 2.6: The same instance of the ham sandwich problem both in the primal and in the dual plane. For $1 \leq i \leq 3$, line ℓ_i and point ℓ_i^* represent possible solutions in their respective planes.

To solve Problem 2 in general position, similarly to Problem 1, we may assume that the number of lines in both sets is odd. That way, the point that corresponds to a solution will be the intersection of a line in H_1 and a line in H_2.

That gives us a naive solution to Problem 2, presented in Algorithm 4: go through all intersections of two lines and check if each intersection is a ham sandwich point.

Algorithm 4 DUALNAIVESOLVE(H_1, H_2)

Input: two nonempty sets H_1 and H_2 of lines in general position
Output: a ham sandwich point p of H_1 and H_2

```
1: for h1 ∈ H1 do
2:     for h2 ∈ H2 do
3:         x ← (h2.b − h1.b) / (h1.m − h2.m)
4:         y ← h1.m · x + h1.b
5:         p ← (x, y)                          ▷ intersection of h1 and h2
6:         if MEDIANZONE(p, H1) and MEDIANZONE(p, H2) then return p
```

Algorithm 4 just happens to be Algorithm 2 converted to the dual plane. It can also be optimized to run in $\mathcal{O}(n^2 \log n)$ time (where $n = |H_1| + |H_2|$), but, in the dual plane, the optimized version, shown in Algorithm 5, is a sweep line algorithm that is much easier to visualize.

The algorithm sweeps the plane from left to right, keeping the order in which the red and the blue lines intersect the sweep line. Initially, the lines intersect the sweep line in order of their slopes. This order changes every time the sweep line passes by an intersection of two lines of the same set. Intersections of two lines of different sets are candidates to ham sandwich points, so the events of this sweep line algorithm are the intersections of pairs of the given lines. These intersections are processed in order of their x-coordinate.

Intersections of lines of the same set cause the order of the intersection of these lines with the sweep line to invert. Intersections of lines of different sets are tested to see if they are a ham sandwich point. For this test to be made efficiently, the algorithm keeps track of the position of each line in the order of intersection of all lines of its set with the sweep line. To decide whether an intersection of two lines of different sets is a ham sandwich point, it is enough to check whether both lines are in the middle position of their respective color order.

In Algorithm 5, the procedure SLOPESORT(H) sorts the list H of lines by slope in $\mathcal{O}(n \log n)$ time, where $n = |H|$. The procedure INTERSECTIONSORT receives two sets H_1 and H_2 of lines in general position and returns a list of all pairs of lines in $H_1 \cup H_2$, sorted by the x-coordinate of the intersection of the lines. Each line is identified by an integer i in $\{1, 2\}$ and the index of the line in H_i. This procedure runs in $\mathcal{O}(n^2 \log n)$, where $n = |H_1| + |H_2|$.

The complexity of Algorithm 5 is that of INTERSECTIONSORT(H_1, H_2), that is, $\mathcal{O}(n^2 \log n)$. Both this and the primal $\mathcal{O}(n^2 \log n)$ algorithm were described with the intent of showing that visualizing the problem in its dual plane can simplify certain algorithms.

Algorithm 5 OPTIMIZEDDUALNAIVESOLVE(H_1, H_2)

Input: two nonempty sets H_1 and H_2 of lines in general position
Output: a ham sandwich point p for H_1 and H_2

1: **if** $|H_1|$ is even **then** remove an arbitrary line of H_1
2: **if** $|H_2|$ is even **then** remove an arbitrary line of H_2
3: SLOPESORT(H_1) ▷ $H_1[1].m < \cdots < H_1[|H_1|].m$
4: $\pi_1 \leftarrow (1, \ldots, |H_1|)$
5: SLOPESORT(H_2) ▷ $H_2[1].m < \cdots < H_2[|H_2|].m$
6: $\pi_2 \leftarrow (1, \ldots, |H_2|)$
7: ▷ Order of the intersection with the sweep line: $H_i[\pi_i[1]] \prec \cdots \prec H_i[\pi_i[|H_i|]]$ for i=1, 2
8: *Events* $\leftarrow$ INTERSECTIONSORT(H_1, H_2)
9: $t_1 \leftarrow \left\lceil \frac{|H_1|}{2} \right\rceil$
10: $t_2 \leftarrow \left\lceil \frac{|H_2|}{2} \right\rceil$
11: **for** $\{(i, j), (i', j')\} \in$ *Events* **do**
12: **if** $i = i'$ **then** ▷ both lines are in the same H_i?
13: $\pi_i[j] \leftrightarrow \pi_i[j']$ ▷ they exchange places in the order $\prec$
14: **else** ▷ check whether the intersection is a ham sandwich point
15: **if** $\pi_i[j] = t_i$ **and** $\pi_{i'}[j'] = t_{i'}$ **then** ▷ both lines are in the median zone?
16: $x \leftarrow \frac{H_1[t_1].b - H_2[t_2].b}{H_2[t_2].m - H_1[t_1].m}$
17: $y \leftarrow H_1[t_1].m \cdot x + H_1[t_1].b$ ▷ their intersection
18: **return** (x, y)

2.3 Existence of a ham sandwich cut

The duality transformation also makes it easier to prove that there is always a solution to the ham sandwich problem.

Firstly, we will assume that the points in P_1 and P_2 are in general position, which implies that every pair of lines in P_1^* and P_2^* intersects at exactly one point and there are no vertical lines. We will also assume that both $|P_1|$ and $|P_2|$ are odd.

To prove the existence of a ham sandwich cut, we will need to make some extra observations.

Proposition 6. *The median zone of an odd set of lines is a polygonal chain (Figure 2.7).*

Proof. Let H be an odd set of lines. Since $|H|$ is odd, a point in the median zone of H must lay on a line of H. Furthermore, given the x-coordinate of the point, there is only one candidate for its y-coordinate: the one determined by a line that is in the middle when we sort H by evaluation at such x-coordinate.

Notice that the line that represents the median zone only changes when that line intersects with the next median line; that other line will now be the one to represent the median zone. That means that the median zone is a connected series of line segments, i.e., a polygonal chain. □

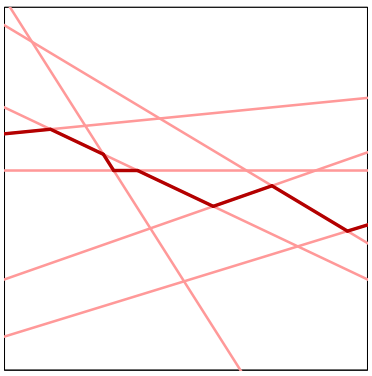


Figure 2.7: The darker polygonal chain is the median zone of the set of lines displayed.

Definition 5. *The i-th level of a set H of lines, denoted as $L_i(H)$, is the polygonal chain such that there are at most $i-1$ lines of H strictly below $L_i(H)$ and at most $|H|-i$ lines of H strictly above $L_i(H)$.*

Note that the median level of an odd set H of lines, $L_{\left\lceil \frac{|H|}{2} \right\rceil}(H)$, is the median zone of H, as it was defined previously.

Now let P_1 and P_2 be the sets of points in general position for which we want to find a ham sandwich cut. We are looking for a line that bisects both P_1 and P_2, which is the same as a point that is in the median zone of both P_1^* and P_2^*. If P_1 and P_2 are odd, that means we want to find an intersection of the median levels of both. If that intersection exists, then it is the dual of a ham sandwich cut.

Lemma 1. *Let H_1 and H_2 be two odd sets of lines in general position. Their median levels intersect at an odd number of points (Figure 2.8).*

Proof. The left unbounded ray and the right unbounded ray of the median level of any odd set of lines in general position lie on the same line: the line that is the median in the slope order. Let us say such line is h_1 for H_1 and h_2 for H_2. Because the lines are in general position, one of them must have a smaller slope. Let us assume without loss of generality that this is h_1. That means that, for small enough x, $h_1(x) > h_2(x)$ and the median level of H_1 is above the median level of H_2 and, for large enough x, $h_1(x) < h_2(x)$ and the median level of H_1 is below the median level of H_2. By continuity, the median levels intersect an odd number of times. □

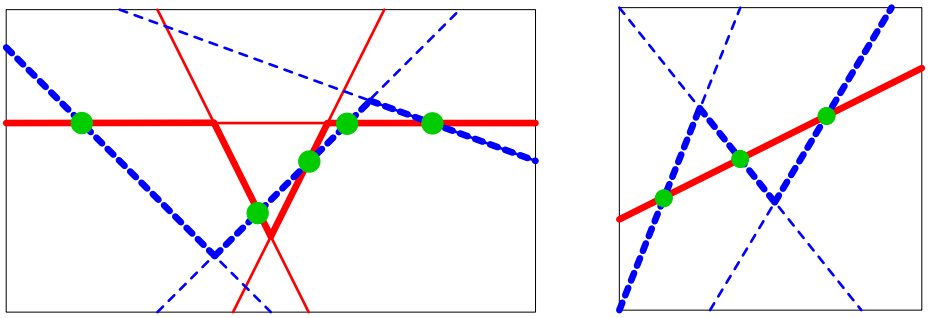


Figure 2.8: The red (solid) lines represent H_1, the blue (dashed) lines represent H_2. The thicker lines represent the median level of each set and the points represent where those median levels intersect.

If the set of lines is not in general position, the lines in which the median levels lay on might be parallel. In those cases, the median level of the two sets of lines do not necessarily intersect and there might be no solution for the problem (Figure 2.9).

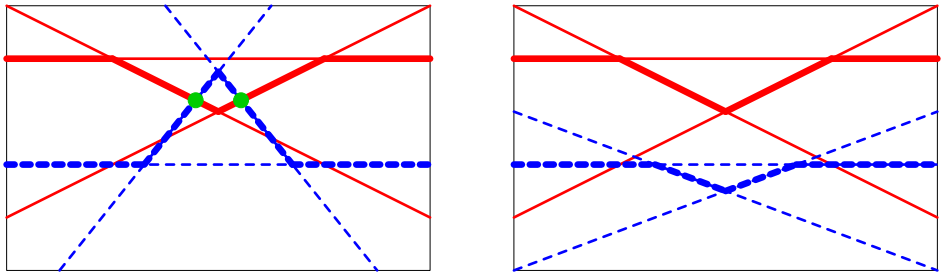


Figure 2.9: When the median levels of the sets of lines are parallel, their median levels may or may not intersect.

Now we can finally prove the result.

Theorem 1. *Given two sets P_1 and P_2 of points in $\mathbb{R}^2$ in general position, there always exists a line that bisects both sets P_1 and P_2 simultaneously.*

Proof. By Proposition 2 we may assume both sets of points are odd, as otherwise we can just remove any point from the even sets and the solution found will still be applicable.

By Proposition 5, it is enough to find a ham sandwich point p for $H_1 := P_1^*$ and $H_2 := P_2^*$, if one such point exists.

From Lemma 1, the median levels of H_1 and H_2 intersect an odd number of times, which is necessarily a positive number of times. That means there is a point p that is in both the median level of H_1 and in the median level of H_2, so p is a ham sandwich point. The dual of p is a line that bisects both sets. □

Chapter 3

Lo, Matoušek, and Steiger algorithm

This chapter presents an algorithm, proposed by Lo, Matoušek, and Steiger [7], that solves Problem 2. Their algorithm has a linear-time implementation. By Proposition 5, it can be easily converted to an algorithm with the same running time for Problem 1.

In this chapter, we will describe an $\mathcal{O}(n \log n)$ implementation for their algorithm, where n is given number of lines, that is, $n = |H_1| + |H_2|$. We will postpone the most intricate part of the linear-time implementation to Chapter 5.

We reinforce that we are assuming that the lines are in general position. This means the algorithm will always find a solution.

3.1 Outline of the algorithm

The algorithm will search for an intersection between the median zones of our two sets of lines, H_1 and H_2. In our description, we will use the following definitions.

Definition 6. *Let T be an open interval on the x-axis. A T-intersection is an intersection between two lines whose x-coordinate lies in T.*

Definition 7. *Let T be an open interval on the x-axis. A T-trapezoid is a trapezoid that has two sides parallel to the y-axis, the first of which is a segment contained in the vertical line defined by the beginning of T and the second of which is a segment contained in the vertical line defined by the end of T.*

Definition 8. *Let T be an open interval on the x-axis, G_1 and G_2 be sets of lines, and p_1 and p_2 be integers such that $1 \leq p_i \leq |G_i|$. Interval T has the odd intersection property in relation to $L_{p_1}(G_1)$ and $L_{p_2}(G_2)$ if $L_{p_1}(G_1)$ and $L_{p_2}(G_2)$ intersect an odd number of times in T.*

The basic idea of the algorithm is to find subsets $G_1 \subseteq H_1$ and $G_2 \subseteq H_2$ in such a way that there is at least one intersection of the median levels of H_1 and H_2 that is also an intersection of a line in G_1 and a line in G_2. We will remove lines from G_1 and G_2 in phases while maintaining this property, and once $|G_1| + |G_2|$ is small enough, we will find a solution using some naive algorithm.

The algorithm will work in phases. At the beginning of each phase, the algorithm will have the following data:

- current sets G_1 and G_2 of lines, with $G_i \subseteq H_i$;
- integers p_1 and p_2, with $1 \leq p_i \leq |G_i|$, and
- an open interval T on the x-axis that has the odd intersection property in relation to $L_{p_1}(G_1)$ and $L_{p_2}(G_2)$.

These are such that all T-intersections between $L_{p_1}(G_1)$ and $L_{p_2}(G_2)$ are valid ham sandwich points for H_1 and H_2. The algorithm will also use a constant α, whose value we will define later.

At the end of each phase, lines are discarded so we have new sets $G_i' \subseteq G_i$, integers $p_i' \leq p_i$ such that $1 \leq p_i' \leq |G_i'|$, and a new interval $T' \subseteq T$ on which the invariant holds for the new data. To maintain the time complexity of the final algorithm equal to the time complexity of one of its phases, the larger of G_i will have its size cut by some fraction that is determined by the constant α.

To start the algorithm, T is the whole x-axis, $G_i = H_i$ and $p_i = \left\lceil \frac{|H_i|}{2} \right\rceil$. To perform one of the phases, we will execute the following steps, always ensuring beforehand that $|G_1| \geq |G_2|$ (otherwise we swap G_1 and G_2).

1. Find an interval $T' \subseteq T$ so that there are no more than $\alpha\binom{|G_1|}{2}$ T'-intersections among the lines in G_1 and T' has the odd intersection property in relation to $L_{p_1}(G_1)$ and $L_{p_2}(G_2)$. Let T' be the new T.

   This step will be represented by the function NewInterval(G_1, G_2, p_1, p_2, T) (Section 3.2), which returns the new interval T' given the parameters as stated above.

2. Construct a T-trapezoid τ that contains the entirety of $L_{p_1}(G_1)$ in T and intersects only a certain predetermined fraction of lines in G_1. This fraction is determined by α and its value is chosen so that such a τ always exists.

   This step will be represented by the function FindTrapezoid(G_1, p_1, T) (Section 3.3), which returns the T-trapezoid τ.

3. Discard all the lines in G_1 that do not intersect τ and update p_1 accordingly.

   This step will be represented by the function DiscardLines(G_1, p_1, τ) (Section 3.4), which returns G_1' and p_1' according to what was described.

If we use $\alpha = \frac{1}{32}$ in Step 2, at the end of the phase, G_1 will have its size cut by at least half [7, Lemma 3.5]. Once the size of G_1 is smaller than some constant, we can use some naive algorithm to find a T-intersection of $L_{p_1}(G_1)$ and $L_{p_2}(G_2)$. This algorithm will be BruteForce(G_1, G_2, p_1, p_2, T) (Section 3.5).

Since $|G_1| \geq |G_2|$ and $|G_1|$ decreases by half by the end of each phase, $|G_1| + |G_2|$ decreases to at most $\frac{3}{4}$ of its previous value by the end of each phase. As long as each phase takes time $\mathcal{O}(|G_1| + |G_2|)$, the total time for this procedure will be a geometric progression $(|H_1| + |H_2|) + \frac{3}{4}(|H_1| + |H_2|) + (\frac{3}{4})^2(|H_1| + |H_2|) + \ldots$, which is $\mathcal{O}(|H_1| + |H_2|) = \mathcal{O}(n)$.

The algorithm that solves the actual problem is presented in Algorithm 6 and the figures that follow represent what one of its phases might look like (lines 9 to 12). In the figures we use an α greater than $\frac{1}{32}$ to make it easier to visualize the lines. As a consequence, less than half of the lines will be discarded in Step 3.

Algorithm 6 HAMSANDWICHPOINT(G_1, G_2)

Input: two nonempty sets G_1 and G_2 of lines in general position
Output: a point that is in the intersection of the median zones of G_1 and G_2

1: $\alpha \leftarrow \frac{1}{32}$
2: **if** $|G_1|$ is even **then** remove any line from G_1
3: **if** $|G_2|$ is even **then** remove any line from G_2
4: **if** $|G_2| > |G_1|$ **then** $G_1 \leftrightarrow G_2$ ▷ ensuring $|G_1| \geq |G_2|$
5: $p_1 \leftarrow \left\lceil \frac{|G_1|}{2} \right\rceil$
6: $p_2 \leftarrow \left\lceil \frac{|G_2|}{2} \right\rceil$
7: $T \leftarrow (-\infty, \infty)$
8: **while** $\binom{|G_1|}{2} > \frac{1}{\alpha}$ **do**
9: $\quad T \leftarrow$ NEWINTERVAL(G_1, G_2, p_1, p_2, T)
10: $\quad \tau \leftarrow$FINDTRAPEZOID(G_1, p_1, T)
11: $\quad (G_1, p_1) \leftarrow$DISCARDLINES$(G_1, p_1, \tau)$
12: $\quad$ **if** $|G_2| > |G_1|$ **then** $(G_1, p_1) \leftrightarrow (G_2, p_2)$ ▷ ensuring $|G_1| \geq |G_2|$
13: **return** BRUTEFORCE(G_1, G_2, p_1, p_2, T) ▷ $|G_1|$ and $|G_2|$ are sufficiently small

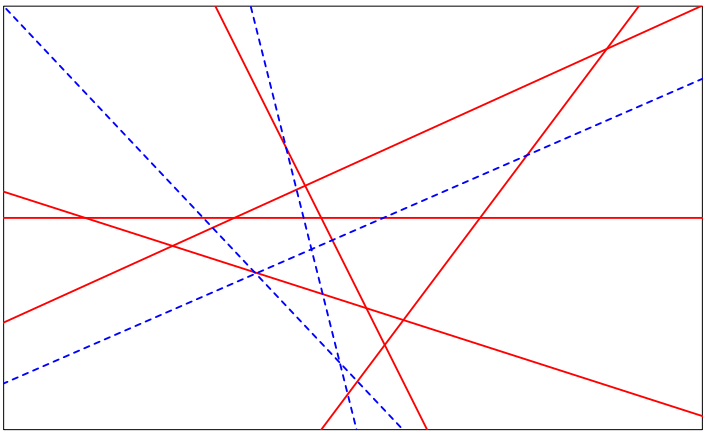


Figure 3.1: An instance of the Dual 2D Ham Sandwich Problem:
the lines in red (solid) represent H_1 and the lines in blue (dashed) represent H_2.

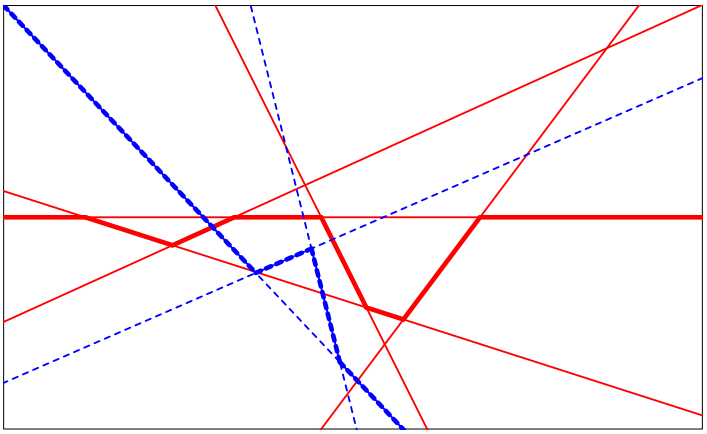


Figure 3.2: The parameters at the beginning of the algorithm: T is the whole x-axis, $G_1 = H_1$, $G_2 = H_2$, $p_1 = 3$ and $p_2 = 2$. The thicker lines show the levels $L_{p_1}(G_1)$ and $L_{p_2}(G_2)$.

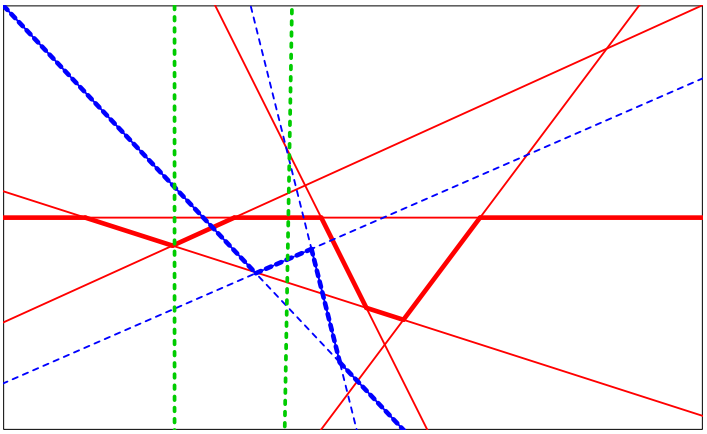


Figure 3.3: The two green (dotted) lines indicate interval T', found in Step 1, containing only a fraction of the intersections from G_1 and an odd number of intersections between $L_3(G_1)$ and $L_2(G_2)$.

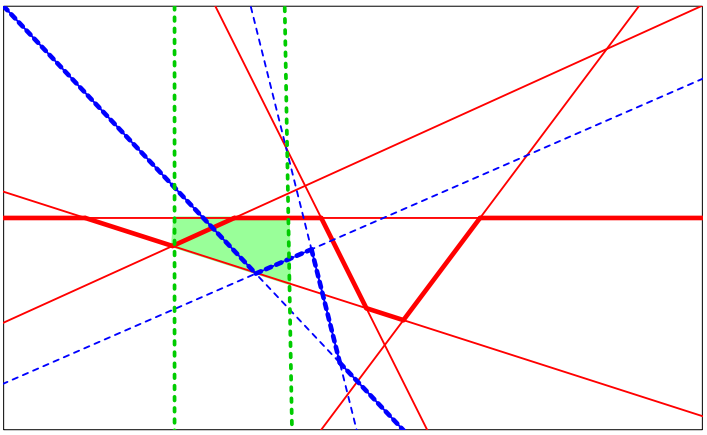


Figure 3.4: In Step 2, we construct the T-trapezoid τ that contains the entirety of $L_{p_1}(G_1)$ in T.

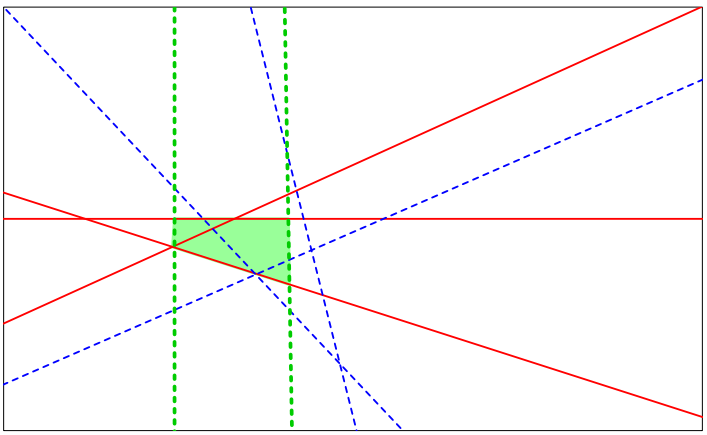


Figure 3.5: In Step 3, we eliminate the red (solid) lines that do not intersect τ.

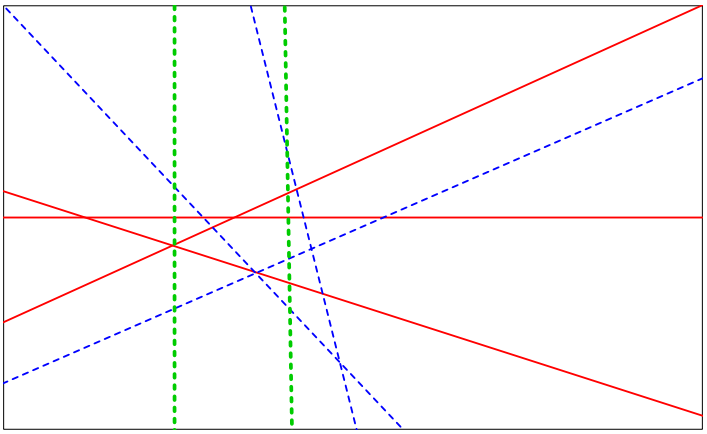


Figure 3.6: Now we are ready to the next phase. Our new G_1 is the current set of lines in red (solid) and our new G_2 is the current set of lines in blue (dashed). The new interval T is the range delimited by the green (dotted) lines. We now want to find a T-intersection between the levels $L_{p_1=2}(G_1)$ and $L_{p_2=2}(G_2)$. We do not need to swap the sets because $|G_1| \geq |G_2|$.

In the following sections, the subroutines used in Algorithm 6 will be explained in details.

3.2 New interval

For didactic purposes, this section will discuss an $\mathcal{O}((|G_1| + |G_2|) \log(|G_1| + |G_2|))$ expected time implementation of the function NewInterval. This complexity is insufficient for the linear-time implementation of Lo, Matoušek, and Steiger algorithm. A worst-case linear implementation will be described in Chapter 5 and uses some of the ideas that will be introduced here.

The function NewInterval receives two sets G_1 and G_2 of lines, integers p_1 and p_2 with $1 \leq p_i \leq |G_i|$, and an interval T that has the odd intersection property in relation to $L_{p_1}(G_1)$ and $L_{p_2}(G_2)$. The function returns an interval $T' \subset T$ that has

the odd intersection property in relation to $L_{p_1}(G_1)$ and $L_{p_2}(G_2)$ and such that the number of T'-intersections among lines in G_1 is no more than $\alpha\binom{|G_1|}{2}$ (for $\alpha = \frac{1}{32}$).

We will find the new interval T' by performing a binary search in the T-intersections among lines in G_1.

The algorithm will execute the following steps:

1.1 Choose a T-intersection among lines in G_1 uniformly at random. The x-coordinate of that intersection divides T into two intervals: T_1, consisting of the part of T that is to the left of this coordinate, and T_2, consisting of the part of T that is to the right of this coordinate.

1.2 Update T to be the one between T_1 and T_2 in which $L_{p_1}(G_1)$ and $L_{p_2}(G_2)$ intersect an odd number of times.

1.3 If the number of T-intersections among lines in G_1 is greater than $\alpha\binom{|G_1|}{2}$, go back to Step 1.1.

Both Step 1.1 and Step 1.3 will be done using the $\mathcal{O}(|G|\log|G|)$ function IntersectionsAndRandomIntersection(G, T), which returns the number of T-intersections among lines in G and a random T-intersection, if one exists. Step 1.2 will be done using the function OddIntersectionProperty(G_1, G_2, p_1, p_2, T), that checks whether T has the odd intersection property in relation to $L_{p_1}(G_1)$ and $L_{p_2}(G_2)$ in $\mathcal{O}((|G_1| + |G_2|)\log(|G_1| + |G_2|))$.

Once we have the functions IntersectionsAndRandomIntersection and OddIntersectionProperty, NewInterval can be written as in Algorithm 7.

Algorithm 7 NewInterval(G_1, G_2, p_1, p_2, T)

Input: two sets G_1 and G_2 of lines in general position, two integers p_1 and p_2 with $1 \leq p_i \leq |G_i|$, and an interval T that has the odd intersection property in relation to $L_{p_1}(G_1)$ and $L_{p_2}(G_2)$

Output: an interval $T' \subset T$ such that the number of T'-intersections among the lines in G_1 is no more than $\alpha\binom{|G_1|}{2}$ (for $\alpha = \frac{1}{32}$) and T' has the odd intersection property in relation to $L_{p_1}(G_1)$ and $L_{p_2}(G_2)$

```
α ← 1/32
(n, p) ←IntersectionsAndRandomIntersection(G1,T)
while n > α(|G1| choose 2) do
    (a, b) ← T
    T1 ← (a, p.x)
    T2 ← (p.x, b)
    if OddIntersectionProperty(G1,G2,p1,p2,T1) then T ← T1
    else T ← T2
    (n, p) ←IntersectionsAndRandomIntersection(G1,T)
return T
```

To calculate the running time, we need to determine how many iterations the while in line 3 will execute. By selecting a random T-intersection within G_1 in lines 2 and 9, one can prove that the expected number of T-intersections within G_1 decreases by some fraction in each iteration. That means that we need on average a constant number of iterations until the number of T-intersections within G_1 is small enough, so the algorithm takes on average $\mathcal{O}((|G_1| + |G_2|) \log(|G_1| + |G_2|))$ time.

3.2.1 Finding intersections

We will find T-intersections in a set G of lines using the concept of inversions. For a permutation π, a pair of indices i and j with $1 \leq i \leq j \leq |\pi|$ is an *inversion* if $\pi[i] > \pi[j]$.

If we have two lines g_1 and g_2 in G such that, at the beginning of the interval T, the evaluation of g_1 is greater than the evaluation of g_2 and, at the end of T, the evaluation of g_1 is smaller than the evaluation of g_2, then g_1 and g_2 must intersect inside T.

So let us say we label the lines in G from 1 to n in the order of their evaluation at the beginning of T. We will define π as the permutation that represents the lines in the order of their evaluation at the end of T. Because of what was described above, each inversion in π will be a T-intersection of two lines in G (Figure 3.7).

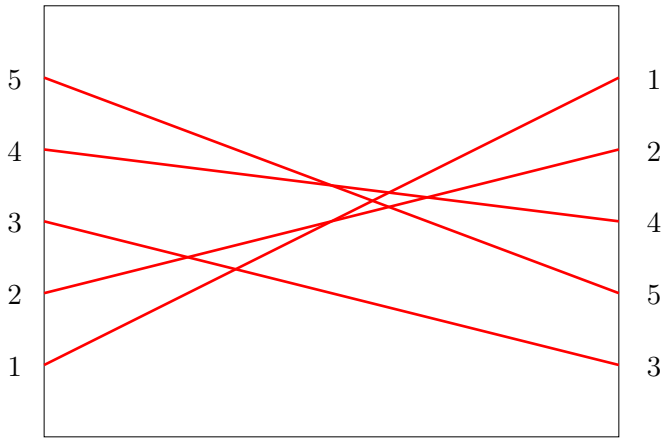


Figure 3.7: Each inversion in the permutation displayed at the right represents an intersection of two lines inside the interval.

That way, we can use variants of inversion counting algorithms to implement IntersectionsAndRandomIntersection. This will be done using Algorithm 9, which transforms the lines into a permutation and then uses Algorithm 8, which counts the inversions in a permutation and returns one chosen uniformly at random.

Algorithm 8 runs in $\mathcal{O}(n \log n)$ time, where $n = j - i + 1$, as it is an adaptation of the well-known Mergesort algorithm for counting inversions. It uses the procedure rand(i, j), that returns an integer in the interval $[i, j)$ uniformly chosen at random.

Algorithm 9 uses as subroutines the procedure SortEval(G, z), that sorts the set G of lines by increasing value of $g(z)$ for each line $g \in G$ and the procedure IndSortEval(G, z), that returns the permutation that sorts G by increasing value of $g(z)$ for each line $g \in G$. In case of ties, the line with larger slope is considered

greater. That is as if the lines have already intersected, so it is as if we were actually sorting by increasing value of $g(z+\epsilon)$ for a tiny $\epsilon > 0$. For $z = -\infty$, the sorting is by decreasing slope and, for $z = \infty$, by increasing slope of the lines in G. Both procedures run in $\mathcal{O}(|G| \log |G|)$ time.

Algorithm 8 InversionsAndRandomInversion(π, i, j)

Input: a permutation π and integers i and j with $1 \le i \le j \le |\pi|$
Output: the number of inversions in $\pi[i\,..\,j]$ and a random pair of integers inverted in $\pi[i\,..\,j]$, if one exists. As a side effect, the algorithm sorts $\pi[i\,..\,j]$.

1: **if** $i \ge j$ **then return** $(0, \text{none}, \text{none})$
2: $k \leftarrow \left\lfloor \frac{i+j}{2} \right\rfloor$
3: $(c_1, u_1, v_1) \leftarrow$ InversionsAndRandomInversion(π, i, k)
4: $(c_2, u_2, v_2) \leftarrow$ InversionsAndRandomInversion$(\pi, k+1, j)$
5: $c_3 \leftarrow 0$ ▷ inversions between the two now sorted halves of π
6: $t_1 \leftarrow i, \quad t_2 \leftarrow k+1$
7: **for** $t \in [i, j]$ **do** ▷ similar to the Merge subroutine of Mergesort
8: **if** $(t_1 \le k$ **and** $t_2 \le j$ **and** $\pi[t_1] < \pi[t_2])$ **or** $t_2 > j$ **then**
9: $\pi'[t] \leftarrow \pi[t_1]$
10: $t_1 \leftarrow t_1 + 1$
11: **else**
12: $c_3 \leftarrow c_3 + k - t_1 + 1$
13: **if** $c_3 > 0$ **then**
14: $r \leftarrow$ rand$(0, c_3)$
15: **if** $r < k - t_1 + 1$ **then** $u_3, v_3 \leftarrow \pi[t_2], \pi[t_1 + r]$
16: $\pi'[t] \leftarrow \pi[t_2]$
17: $t_2 \leftarrow t_2 + 1$
18: **for** $t \in [i, j]$ **do** $\pi[t] \leftarrow \pi'[t]$
19: $c \leftarrow c_1 + c_2 + c_3$
20: **if** $c = 0$ **then return** $(0, \text{none}, \text{none})$
21: $r \leftarrow$ rand$(0, c)$
22: **if** $r < c_1$ **then return** (c, u_1, v_1)
23: **if** $r < c_1 + c_2$ **then return** (c, u_2, v_2)
24: **return** (c, u_3, v_3)

Algorithm 9 IntersectionsAndRandomIntersection(G, T)

Input: a set G of lines in general position and an interval T
Output: the number of T-intersections of lines in G and a random T-intersection of two lines in G

1: $(a, b) \leftarrow T$
2: SortEval(G, a) ▷ $G[1] \prec_a \cdots \prec_a G[|G|]$
3: $\pi \leftarrow$ IndSortEval(G, b) ▷ $G[\pi[1]] \prec_b \cdots \prec_b G[\pi[|G|]]$
4: $(c, u, v) \leftarrow$ InversionsAndRandomInversion$(\pi, 1, |\pi|)$
5: **if** $c = 0$ **then return** $(c, \text{none}, \text{none})$
6: *inter* $\leftarrow$ intersection$(G[u], G[v])$
7: **return** (c, inter)

3.2.2 Checking the odd intersection property

To decide if an interval has the odd intersection property in relation to $L_{p_1}(G_1)$ and $L_{p_2}(G_2)$, we only have to check whether $L_{p_1}(G_1)$ and $L_{p_2}(G_2)$ changed order in that interval (e.g., $L_{p_1}(G_1)$ was below $L_{p_2}(G_2)$ at the beginning of the interval but above it at the end of the interval: by continuity, they must have intersected an odd number of times), which is what we do in Algorithm 10.

We will determine what is the p-th line of a set of n lines when it is sorted by increasing evaluation at a given x using an $\mathcal{O}(n)$ function QUICKSELECT(G, p, x). The linear-time version of QUICKSELECT uses the median of medians of five algorithm proposed by Blum et al. [3]. With that, ODDINTERSECTIONPROPERTY has a final complexity of $\mathcal{O}(|G_1| + |G_2|)$, which is important because we will be using it later in the linear-time implementation of the NEWINTERVAL function.

Algorithm 10 ODDINTERSECTIONPROPERTY(G_1, G_2, p_1, p_2, T)

Input: sets G_1 and G_2 of lines in general position, integers p_1 and p_2 with $1 \leq p_i \leq |G_i|$ and an interval T
Output: TRUE if $L_{p_1}(G_1)$ and $L_{p_2}(G_2)$ intersect an odd number of times in T and FALSE otherwise

1: **if** T is empty **then**
2: **return** FALSE
3: $(a, b) \leftarrow T$
4: $g_1 \leftarrow$ QUICKSELECT(G_1, p_1, a) ▷ $L_{p_1}(G_1)$ at $x = a$ is on line g_1
5: $g_2 \leftarrow$ QUICKSELECT(G_2, p_2, a) ▷ $L_{p_2}(G_2)$ at $x = a$ it on line g_2
6: $order_a \leftarrow g_1(a) < g_2(a)$ ▷ TRUE iff $L_{p_1}(G_1)$ is below $L_{p_2}(G_2)$ at $x = a$
7: $g_1 \leftarrow$ QUICKSELECT(G_1, p_1, b) ▷ $L_{p_1}(G_1)$ at $x = b$ is on line g_1
8: $g_2 \leftarrow$ QUICKSELECT(G_2, p_2, b) ▷ $L_{p_2}(G_2)$ at $x = b$ is on line g_1
9: $order_b \leftarrow g_1(b) < g_2(b)$ ▷ TRUE iff $L_{p_1}(G_1)$ is below $L_{p_2}(G_2)$ at $x = b$
10: **return** $order_a \neq order_b$

Here we will present a version of QUICKSELECT that has *expected* linear time but is better in practice. It consists of a recursive algorithm presented in Algorithm 12. that works by selecting a random pivot in the given range and then determining whether the number we are looking for is located before or after the pivot, and recursively solving the problem for the desired side of the pivot. For the recursion, two parameters i and j are added, to indicate that the set of lines for the current call is $G[i\,..\,j]$. Then QUICKSELECT(G, p, x), shown in Algorithm 11, simply returns the result of the call QUICKSELECTREC$(G, 1, |G|, p, x)$, and has expected linear-time complexity, but its worst case is $\mathcal{O}(|G|^2)$.

Algorithm 11 QUICKSELECT(G, p, x)

Input: a set G of lines, an index p such that $1 \leq p \leq i - j + 1$, and a coordinate x
Output: the p-th line of G when sorted by increasing evaluation at x

1: **return** QUICKSELECTREC$(G, 1, |G|, p, x)$

Algorithm 12 QUICKSELECTREC(G, i, j, p, x)

Input: a set G of lines, indices i and j such that $1 \leq i \leq j \leq |G|$, an index p such that $1 \leq p \leq i - j + 1$, and a coordinate x
Output: the p-th line of $G[i\,..\,j]$ when sorted by increasing evaluation at x

```
1:  r ← RAND(i, j + 1)
2:  id ← i
3:  G[r] ↔ G[j]
4:  for k ∈ (i, j) do
5:      if G[k](x) < G[j](x) then
6:          G[k] ↔ G[id]
7:          id ← id + 1
8:  G[j] ↔ G[id]
9:  if id − i + 1 > p then                    ▷ the answer is to the left
10:     return QUICKSELECTREC(G, i, id − 1, p, x)
11: if id − i + 1 < p then                    ▷ the answer is to the right
12:     return QUICKSELECTREC(G, id + 1, j, p − (id − i + 1), x)
13: return G[id]
```

From now on, to simplify expressions in the pseudocode, we will sometimes write $g(x)$ to mean $g.m \cdot x + g.b$ for an arbitrary line g. We did this, for instance, in line 5 of Algorithm 12.

3.3 Find trapezoid

Now we will present a linear-time implementation of FINDTRAPEZOID(G, p, T), based on the description found in [7] just before Lemma 3.5 and on the parameters given by the same Lemma 3.5.

This function receives a set G of lines, an index p with $1 \leq p \leq |G|$, and an interval T, and returns a T-trapezoid τ that contains the entirety of $L_p(G)$ in T and intersects only a certain predetermined fraction of lines in G.

The left side of τ will be bounded by the $(p - \epsilon|G|)$-th and the $(p + \epsilon|G|)$-th lines of G when these lines are sorted by evaluation at the left end of T for a certain $\epsilon > 0$. Analogously, the right side of τ will be bounded by the $(p - \epsilon|G|)$-th and the $(p + \epsilon|G|)$-th lines of G when these lines are sorted by evaluation at the right end of T. According to [7, Lemma 3.5], for the value $\epsilon = \frac{1}{8}$, less than half of the lines in G intersect the obtained T-trapezoid τ and such τ contains the entirety of $L_p(G)$ in T.

That means that we need to be able to determine what is the p-th line of a set G of lines when it is sorted by increasing evaluation at a given x. That can be done in linear time using the QUICKSELECT function discussed in Section 3.2.2.

With that, our final algorithm to find the T-trapezoid is as shown in Algorithm 13.

Algorithm 13 FINDTRAPEZOID(G, p, T)

Input: a set G of lines in general position, an integer p such that $1 \leq p \leq |G|$, and an interval T

Output: a T-trapezoid that intersects less than half of the lines in G and that contains the entirety of $L_p(G)$

1: $(a, b) \leftarrow T$
2: $\mathit{offset} \leftarrow \left\lfloor \frac{|G|}{8} \right\rfloor$
3: $\mathit{left}_{up} \leftarrow$QUICKSELECT$(G, p + \mathit{offset}, a)(a)$ ▷ $L_{p+\mathit{offset}}(G)$ at a
4: $\mathit{left}_{down} \leftarrow$QUICKSELECT$(G, p - \mathit{offset}, a)(a)$ ▷ $L_{p-\mathit{offset}}(G)$ at a
5: $\mathit{right}_{up} \leftarrow$QUICKSELECT$(G, p + \mathit{offset}, b)(b)$ ▷ $L_{p+\mathit{offset}}(G)$ at b
6: $\mathit{right}_{down} \leftarrow$QUICKSELECT$(G, p - \mathit{offset}, b)(b)$ ▷ $L_{p-\mathit{offset}}(G)$ at b
7: $\tau \leftarrow (\mathit{left}_{up}, \mathit{left}_{down}, \mathit{right}_{up}, \mathit{right}_{down})$
8: **return** τ

3.4 Discard lines

Lastly, let us describe the procedure that performs Step 3 in the algorithm outlined in Section 3.1. That is, we will describe the function DISCARDLINES(G, p, τ), that receives a set G of lines in general position, an integer p such that $1 \leq p \leq |G|$, and a T-trapezoid τ. The function returns a new set G' of lines that is obtained by removing all lines in G that do not intersect τ and a new index p' such that $L_{p'}(G') = L_p(G)$ within T.

This can be done by simply going through all lines in G and checking if they intersect τ. Additionally, for any line in G strictly below τ, the value of p must be decreased by one. We can do that in linear time, as shown in Algorithm 14.

In this algorithm we use two subroutines: INTERSECTS(τ, g), that returns TRUE if the line g intersects the trapezoid τ, and ABOVE(τ, g), that returns TRUE if τ is strictly above g.

Algorithm 14 DISCARDLINES(G, p, τ)

Input: a set G of lines, an index p such that $1 \leq p \leq |G|$ and a trapezoid τ

Output: a subset G' of G that contains only lines of G that intersect τ, and p' so that $L_{p'}(G') = L_p(G)$ within T

1: $G' \leftarrow \{\}$
2: $p' \leftarrow p$
3: **for** $g \in G$ **do**
4: **if** INTERSECTS(τ, g) **then** $G' \leftarrow G' \cup \{g\}$
5: **if** ABOVE(τ, g) **then** $p' \leftarrow p' - 1$
6: **return** (G', p')

3.5 Brute force

The following brute-force procedure, presented in Algorithm 15, can be used at the base of Lo, Matoušek, and Steiger algorithm, when the number of lines in the given sets is small enough. It is a variant of Algorithm 4, the DualNaiveSolve that finds an intersection of the median levels of two sets of lines. In Algorithm 15, we find the intersection of two arbitrary levels. It uses as a subroutine a procedure that decides whether a point q is in $L_p(G)$, presented in Algorithm 16, which is a variant of Algorithm 3.

Algorithm 15 BruteForce(G_1, G_2, p_1, p_2, T)

Input: two nonempty sets G_1 and G_2 of lines, indices p_1 and p_2 with $1 \leq p_i \leq |G|$, and an interval T
Output: a T-intersection between $L_{p_1}(G_1)$ and $L_{p_2}(G_2)$

1: **for** $g_1 \in G_1$ **do**
2: **for** $g_2 \in G_2$ **do**
3: $x \leftarrow \frac{g_2.b - g_1.b}{g_1.m - g_2.m}$
4: $y \leftarrow g_1.m \cdot x + g_1.b$
5: $q \leftarrow (x, y)$ ▷ intersection of g_1 and g_2
6: **if** $x \in T$ **and** IsPthLevel(G_1, p_1, q) **and** IsPthLevel(G_2, p_2, q) **then**
7: **return** q

Algorithm 16 IsPthLevel(G, p, q)

Input: a set G of lines, an index p with $1 \leq p \leq |G|$, and a point q
Output: TRUE if q is in the p-th level of G and FALSE otherwise

1: $below \leftarrow 0$
2: **for** $g \in G$ **do**
3: **if** $q.y < g.m \cdot q.x + g.b$ **then** $below \leftarrow below + 1$
4: **return** $below + 1 = p$

Chapter 4

Sorting networks and variants

The linear-time implementation of the function NewInterval uses a generalization of a so called *sorting network*. These sorting devices were first studied in 1954 by Armstrong, Nelson, and O'Connor, who later patented the idea [10]. They quickly became a subject of interest in the Computer Science community for their theoretical properties and real-world applications, and are now commonly referenced in renowned books in Computer Science [5, 6].

4.1 Sorting networks

A sorting network is a device that can be used to efficiently sort an array when multiple disjoint pairs of elements can be compared at the same time. The sorting algorithm based on a sorting network will work in steps. In each step some disjoint pairs of positions of the array will have their contents swapped if they are in the wrong order. Since these pairs are disjoint, those swaps can be done in parallel. In subsequent steps, other disjoint pairs of positions will be considered and, once all the steps are done, the initial array is ordered (Figure 4.1).

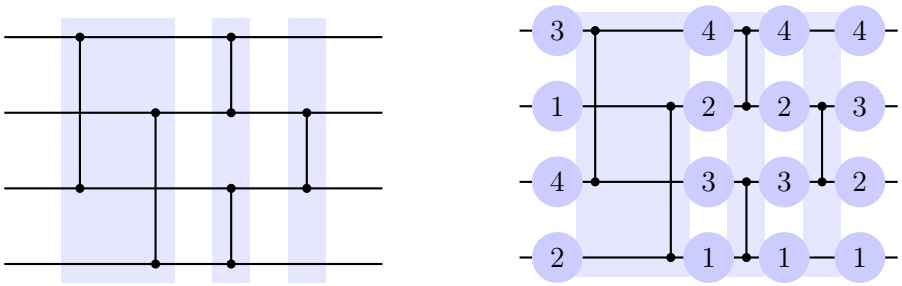


Figure 4.1: The device can be more easily understood through a drawing that resembles a network, hence the name. Each horizontal line corresponds to a position in the array and each vertical line is a potential swap. The rectangles represent the steps of the network. The second figure shows a simulation of what an execution of a sorting network might look like. The steps are processed from left to right and all swaps in the same step are done at once. The permutation that sorts the input vector defines the movements done in the network, in this case: $(3, 1, 4, 2)$.

What follows is one of the possible formalizations of this structure, as written by Matoušek in [8]. It also defines the notation we will be using.

> A *parallel step* P of width n is a set $P = \{(p_1, q_1), \ldots, (p_s, q_s)\}$ of ordered pairs, such that $p_1, q_1, p_2, q_2, \ldots, p_s, q_s$ are distinct members of $[n]$ (thus $s \leq n/2$). The ordered pair (p_i, q_i) is called a *comparator*. A network H of width n and depth D is the ordered D-tuple $H = (P_1, P_2, \ldots, P_D)$ where the P_i are parallel steps of width n.
>
> The computation of a network $H = (P_1, P_2, \ldots, P_D)$ on an input vector $(i_1, \ldots, i_n)$ proceeds as follows: In the first step, i_p is compared with i_q for every $(p, q) \in P_1$, and whenever $i_p > i_q$, these entries in the input vector are exchanged. The input vector modified in this way is then processed by the second parallel step, etc.
>
> The computation of the network obviously depends only on the permutation π which orders the input vector (i.e., such that $i_{\pi(1)} < \cdots < i_{\pi(n)}$). Let $H(\pi)$ denote the permutation performed on the input vector by the network when the input vector is ordered by π. Thus, H is a sorting network in the usual sense if and only if $H(\pi) = \pi$ for every π.

Algorithmically, an interesting problem is how to build a network with few parallel steps that sorts an array of some given size. Most of the networks of width n that are used in practice have $\mathcal{O}(\log^2 n)$ depth, and therefore $\mathcal{O}(n \log^2 n)$ size. An important discovery was the AKS network [1], by Ajtai, Komlós, and Szemedéri, that has $\mathcal{O}(\log n)$ depth, and therefore $\mathcal{O}(n \log n)$ size. This network is extensively used in theoretical constructions, as will be the case for this work, but is unpractical for real applications due to big constants hidden behind its seemingly efficient time complexity. In particular, it involves expander graphs.

4.2 Tolerant ϵ-sorting networks

Using the notation above, π is the permutation that sorts the input vector of a network and $H(\pi)$ is the permutation performed by the network H when that same vector is given as an input.

If H is a sorting network then, for every input vector, $H(\pi) = \pi$. In other words, $H(\pi) \circ \pi^{-1}$ must be the identity permutation, and thus has zero inversions.

Sometimes we do not need to completely sort an array. It might be convenient to have a network H such that, for every input vector, $H(\pi)$ and π are similar enough, although not necessarily equal.

Formally, for an $\epsilon > 0$, we call a network H of width n an *ϵ-sorting network* if $H(\pi) \circ \pi^{-1}$ has a number of inversions no greater than $\epsilon\binom{n}{2}$.

Additionally, for an integer t, a network H is a *t-tolerant* ϵ-sorting network if any network generated by deleting at most t comparators from H is an ϵ-sorting network.

We will be further investigating properties of the AKS network previously mentioned. There are two important properties, stated by Matoušek [8, Lemmas 3.5 and 3.6], that we will be using. These properties are stated in the next lemmas.

Lemma 2. *There exists a constant C such that, for each n and $0 < \epsilon < 1$, the first $C \log \frac{1}{\epsilon}$ steps of the AKS sorting network of width n form an ϵ-sorting network.*

Lemma 3. *Let H be an ϵ-sorting network of depth D and let $t < \frac{\epsilon n}{2^{D+1}}$. Then H is a t-tolerant 2ϵ-sorting.*

From these two lemmas, we derive the following by detailing what is asymptotically stated by Matoušek [8, Corollary 3.7].

Corollary 1. *There exists a constant C such that, for $n \gg 1$ and $0 < \epsilon < 1$, we can construct a t-tolerant ϵ-sorting network $H = H(n, \epsilon)$ of width n and depth at most $C \log(\frac{1}{\epsilon})$, for $t < \epsilon^{C+1} n/4$.*

This means that, for constant values of ϵ, we can have a t-tolerant ϵ-sorting network of constant depth for values of t that depend linearly on n.

4.3 Construction and simulation of sorting networks

By Lemmas 2 and 3, a network as in Corollary 1 is a truncate version of an AKS sorting network. This truncated version of an AKS sorting network of width n has a linear number of comparators on n. Unfortunately, there is no explicit statement in [1, 8] that the construction of such a network takes time proportional to its size. In fact, even the time to construct an expander graph needed for the AKS sorting network is not explicitly stated. But, to assure the linear complexity of the algorithm of Lo, Matoušek, and Steiger, there must be a construction of such a network that takes time linear on its size. So we will assume that such a construction exists, and refer to such constructions in the next chapter. Concretely, we will use in Chapter 5 the two following functions: SortingNetwork(n), that returns a sorting network of width n in time $\mathcal{O}(n \log n)$, and TolerantSortingNetwork(n, ϵ), that returns a t-tolerant ϵ-sorting network of width n and depth at most $C \log(\frac{1}{\epsilon})$, for C given by Corollary 1, for $t = \epsilon^{C+1} n/5$, that runs in time $\mathcal{O}(n \log(\frac{1}{\epsilon}))$.

Now we will present an example of how we might run a sorting network. This also establishes the pseudocode notation that will be used in later algorithms.

For now, the pseudocode will ignore the parallel properties of the network, and the pairs in each step will be processed sequentially. This will be changed later to exploit properties of the structure of the problem at hand. The algorithm also requires the existence of a function IsGreater(S, i, j), that returns TRUE if $S[i] > S[j]$ and FALSE otherwise.

Additionally, it is worth noting that Algorithm 17 applies to both regular sorting networks and tolerant ϵ-sorting networks. It performs a call to IsGreater per comparator in the network.

Algorithm 17 RunNetwork(S, H)

Input: a vector S and a network H of width $|S|$
Output: $H(\pi)$, where π is the permutation that sorts S

```
1: π ← (1, ..., |S|)
2: for P ∈ H do
3:     for (p, q) ∈ P do
4:         if IsGreater(S, π[p], π[q]) then π[p] ↔ π[q]
5: return π
```

Chapter 5

Linear-time implementation

Algorithm 6, presented in Chapter 3, outlines an $\mathcal{O}(n \log n)$ solution to the ham sandwich problem. The bottleneck for the complexity of the solution is Algorithm 7: the procedure NewInterval, so in this section we will go into more detail on how to make it linear.

5.1 Search for a good intersection

The function NewInterval receives two sets G_1 and G_2 of lines, integers p_1 and p_2 with $1 \leq p_i \leq |G_i|$, and an interval T that has the odd intersection property in relation to $L_{p_1}(G_1)$ and $L_{p_2}(G_2)$. It returns an interval $T' \subset T$ that also has the odd intersection property in relation to $L_{p_1}(G_1)$ and $L_{p_2}(G_2)$ and such that the number of T'-intersections among lines in G_1 is no more than $\alpha\binom{|G_1|}{2}$ (for $\alpha = \frac{1}{32}$).

The linear implementation of this function that we will outline was originally described by Lo, Matoušek, and Steiger [7].

The main idea of the algorithm behind this function is to divide the x-axis in a constant number of disjoint open intervals $T_1, \ldots, T_C$, each of them containing no more than a fraction $\alpha\binom{|G_1|}{2}$ of the intersections among the lines in G_1.

If every intersection of $L_{p_1}(G_1)$ and $L_{p_2}(G_2)$ is in exactly one interval, there is an interval T_i such that $T_i \cap T$ has the odd intersection property. With that, one can check for every $i \in [C]$ if $T \cap T_i$ has such property in linear time, using Algorithm 10, and return the interval that does.

To describe the linear function NewInterval, firstly we will assume we have a function FindApproximateIntersection that receives a set G of lines in general position, an integer k with $1 \leq k \leq \binom{|G|}{2}$, and a real number δ with $0 < \delta < 1$, and returns an x-coordinate a such that the number s of intersections of lines in G to the left of the vertical line $x = a$ satisfies $|k - s| \leq \delta\binom{|G|}{2}$. In other words, it finds the x-coordinate of an intersection of lines in G that deviates from the k-th one by at most $\delta\binom{|G|}{2}$ intersections. The original outline of this function was presented by Matoušek [8] and runs in $\mathcal{O}(|G| \log^3 \frac{1}{\delta})$ time.

We will firstly show that such function is enough for us to design a linear version of NewInterval. Then we will expand on how to implement the function with the desired time complexity.

The linear NewInterval will use auxiliary functions IntervalIntersection, that takes two intervals T and T' and returns their intersection in constant time, and OddIntersectionProperty that was previously described as Algorithm 10. The linear implementation of NewInterval is shown in Algorithm 18 so we can analyse its time complexity.

Algorithm 18 NewInterval(G_1, G_2, p_1, p_2, T)

Input: two sets G_1 and G_2 of lines in general position, two integers p_1 and p_2 with $1 \leq p_i \leq |G_i|$, and an interval T that has the odd intersection property in relation to $L_{p_1}(G_1)$ and $L_{p_2}(G_2)$

Output: an interval $T' \subset T$ such that the number of T'-intersections among the lines in G_1 is no more than $\alpha\binom{|G_1|}{2}$ (for $\alpha = \frac{1}{32}$) and T' has the odd intersection property in relation to $L_{p_1}(G_1)$ and $L_{p_2}(G_2)$

1: $\alpha \leftarrow \frac{1}{32}$
2: $\delta \leftarrow \frac{\alpha}{4}$
3: $u_0 \leftarrow -\infty$
4: **for** $i \in [0, \lfloor \frac{2}{\alpha} \rfloor)$ **do**
5: $\quad u_i \leftarrow$ FindApproximateIntersection$(G_1, \left\lceil i\frac{\alpha}{2}\binom{|G_1|}{2}\right\rceil, \delta)$
6: $\quad T_i \leftarrow (u_{i-1}, u_i)$
7: $\quad T' \leftarrow$ IntervalIntersection(T, T_i)
8: $\quad$ **if** OddIntersectionProperty(G_1, G_2, p_1, p_2, T') **then return** T'
9: $T_{last} \leftarrow (u_{\lfloor \frac{2}{\alpha} \rfloor}, +\infty)$
10: $T' \leftarrow$ IntervalIntersection(T, T_{last})
11: **return** T'

As α is a constant, the number of iterations of the **for** loop in line 4 is constant. So Algorithm 18 takes time asymptotically equivalent to the sum of the running times of FindApproximateIntersection and OddIntersectionProperty. The time complexity of NewInterval will then be $\mathcal{O}(n \log^3 \frac{1}{\delta} + n)$, which is $\mathcal{O}(n \log^3 \frac{1}{\delta})$, where $n = |G_1| + |G_2|$. Because δ is also a constant, the function runs in $\mathcal{O}(n)$ time. Now what is left to prove is that this function produces a correct result.

Take $N = \binom{|G_1|}{2}$. Let $t_1 < \cdots < t_N$ be the x-coordinates of all the intersections of lines in G_1. Consider $t_j = -\infty$ if $j < 1$ and $t_j = \infty$ if $j > N$. Given a certain integer k with $0 \leq k \leq N$ and $\delta > 0$, FindApproximateIntersection can be used to find the x-coordinate of an intersection between lines in G_1 in the interval $[t_{\lceil k-\delta N \rceil}, t_{\lceil k+\delta N \rceil})$. We want to use that to determine the intervals $T_1, \ldots, T_C$.

There are two things that need to be proven: firstly that the intersection of T with one of the intervals considered will have the odd intersection property in relation to $L_{p_1}(G_1)$ and $L_{p_2}(G_2)$; secondly that all intervals $T_1, \ldots, T_C$ have at most αN intersections among lines in G_1.

Now let us calculate the maximum possible amount of x-coordinates of intersections between lines in G_1 within any interval T_i. For $0 \leq i < \lfloor \frac{2}{\alpha} \rfloor$, let u_i denote the output of FindApproximateIntersection with $G = G_1$, $k = \left\lceil i\frac{\alpha}{2}N \right\rceil$, and $\delta = \frac{\alpha}{4}$. From the description of the algorithm, we know that

$$u_{i-1} \geq t_{\left\lceil \left\lceil (i-1)\frac{\alpha}{2}N \right\rceil - \frac{\alpha}{4}N \right\rceil} \quad \text{and} \quad u_i \leq t_{\left\lceil \left\lceil i\frac{\alpha}{2}N \right\rceil + \frac{\alpha}{4}N \right\rceil - 1}.$$

Since $T_i = (u_i, u_{i+1})$, the number of intersections among lines in G contained in T_i is at most:

$$\begin{aligned}
&\left\lceil \left\lceil i\frac{\alpha}{2}N \right\rceil + \frac{\alpha}{4}N - 1 \right\rceil - \left\lceil \left\lceil (i-1)\frac{\alpha}{2}N \right\rceil - \frac{\alpha}{4}N \right\rceil - 1 \\
&\qquad \leq \left\lceil \left(i\frac{\alpha}{2}N + 1\right) + \frac{\alpha}{4}N \right\rceil - \left\lceil \left((i-1)\frac{\alpha}{2}N\right) - \frac{\alpha}{4}N \right\rceil - 2 \\
&\qquad \leq \left(\left(i\frac{\alpha}{2}N + 1\right) + \frac{\alpha}{4}N + 1\right) - \left(\left((i-1)\frac{\alpha}{2}N\right) - \frac{\alpha}{4}N\right) - 2 \\
&\qquad = \alpha N.
\end{aligned}$$

Figures 5.1, 5.2, and 5.3 show an example of what that might look like.

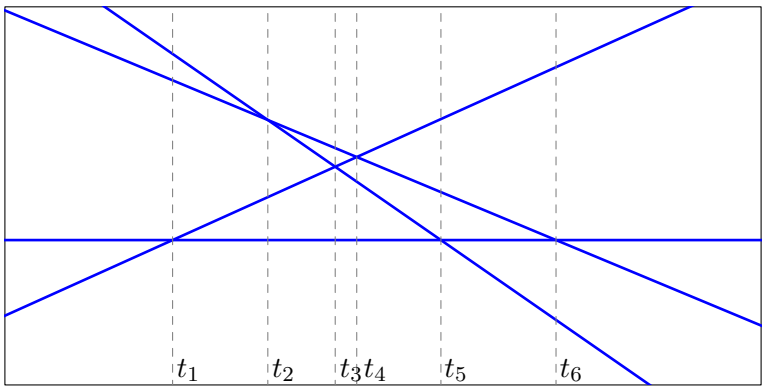


Figure 5.1: The labels of the x-coordinates of the intersections between lines in a set G.

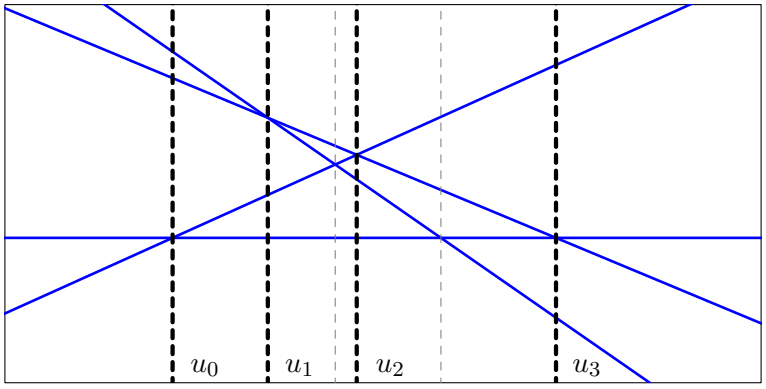


Figure 5.2: For $\alpha = \frac{1}{2}$ (to enable the visualization of a small example) and, since $|G| = 4$ and $N = 6$, the intervals in which each u can lie will be: $u_0 \in [t_0, t_2)$, $u_1 \in [t_2, t_4)$, $u_2 = [t_3, t_5)$ and $u_3 \in [t_5, t_7)$. In the image above, each u lies in its interval.

Also, one can see that the intervals $T_1, \ldots, T_C$ cover the entire span of T except for a subset of $T \cap \{t_1, \ldots, t_N\}$. Since we are assuming general position, the x-coordinate of no intersection of $L_{p_1}(G_1)$ and $L_{p_2}(G_2)$ coincides with the x-coordinate of an intersection of two lines in G_1. So, because T contains an odd number of intersections between $L_{p_1}(G_1)$ and $L_{p_2}(G_2)$ and each one of them must happen in

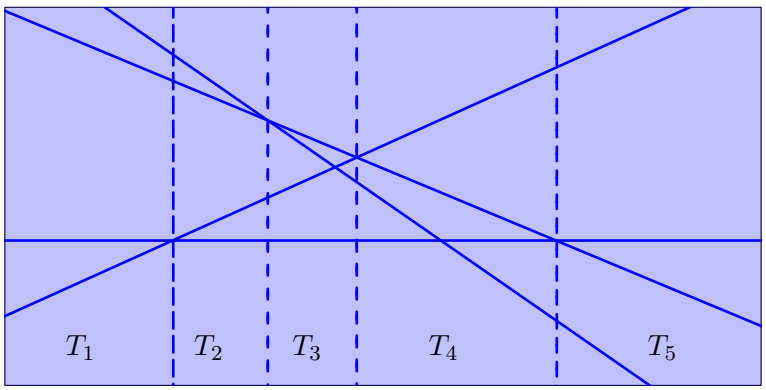


Figure 5.3: Now we can establish our open intervals T_i, in which no interval contains more than $\alpha N = 3$ intersections among the lines in G. Notice that T_1 and T_5 extend to minus infinity and infinity, respectively.

exactly one interval T_i, there must be an interval $T \cap T_i$ containing an odd number of intersections between $L_{p_1}(G_1)$ and $L_{p_2}(G_2)$.

Additionally, we only need to find u_i for $0 \leq i < \lfloor \frac{2}{\alpha} \rfloor$. Since α is a constant, we will be making at most $\lfloor \frac{2}{\alpha} \rfloor$ calls to FindApproximateIntersection, as shown in Algorithm 18.

Our goal is to describe an implementation of FindApproximateIntersection that runs in $\mathcal{O}(|G| \log^3 \frac{1}{\delta})$ time. We will start by describing a slower algorithm to find the k-th intersection among lines in G and will incrementally add modifications to turn this into the desired implementation of FindApproximateIntersection.

5.2 Permutations and intersections

Consider the x-coordinates of all the intersections among the lines in G sorted increasingly. Call this sorted sequence $t_1, t_2, \ldots, t_N$. Our goal is to determine t_k.

Although this is not a sweep line algorithm, we will use similar terminology, associating the x-axis with the notion of time. For example, we can say that t_{i+1} happens later than t_i.

As in a sweep line algorithm, imagine a vertical line that sweeps the plane from left to right. The order in which the lines in G intersect this sweep line changes exactly when the sweep line is passing by one of the coordinates t_l. Keep track of the order in which they intersect the sweep line during this sweep. Specifically, call π_l the permutation that sorts the lines in G in the order in which they intersect the sweep line when it just passed t_l. In particular, π_0 is the permutation that sorts the lines in G by slope, in decreasing order. Note that π_{l-1} and π_l differ by one swap between two consecutive lines in the order given by π_l. The two lines that have been swapped intersect exactly at t_l.

This swap between two adjacent elements that were previously in the same order as in π_0 makes the number of inversions in π_l be one more than the number of inversions in π_{l-1} with respect to π_0. So π_l has exactly l inversions with respect to π_0, as exemplified in Figure 5.4.

We will show that, given π_k for some k, one can find the coordinate t_k in linear

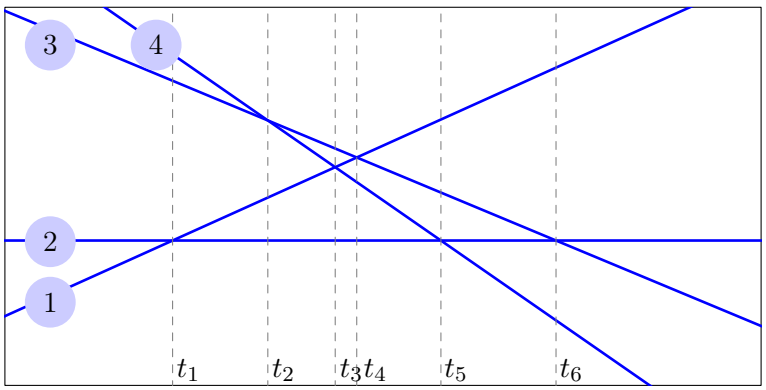


Figure 5.4: To simplify, in the example, the numbering of the lines coincides with π_0, that is, π_0 is the identity. That way, $\pi_0 = (1, 2, 3, 4)$, $\pi_1 = (2, 1, 3, 4)$, $\pi_2 = (2, 1, 4, 3)$, $\pi_3 = (2, 4, 1, 3)$, $\pi_4 = (2, 4, 3, 1)$, $\pi_5 = (4, 2, 3, 1)$, and $\pi_6 = (4, 3, 2, 1)$. Notice that, for every l, π_l has l inversions in relation to π_0.

time. That way, finding the permutation π_k is enough to solve the problem.

Consider all the pairs of lines that have intersected at an x-coordinate no greater than t_k. By the definition of t_k, one of them must have intersected at exactly t_k. The pair of lines that have intersected at coordinates no greater than t_k are the pairs that have changed in order from π_0 to π_k, that would be the pairs labeled (i, j) such that $\pi_0^{-1}(i) > \pi_0^{-1}(j)$ and $\pi_k^{-1}(i) < \pi_k^{-1}(j)$ or such that $\pi_0^{-1}(i) < \pi_0^{-1}(j)$ and $\pi_k^{-1}(i) > \pi_k^{-1}(j)$.

Since the only difference between π_k and π_{k-1} is that two lines that are adjacent in π_k have swapped places, we know that the pair of lines whose intersection has x-coordinate exactly t_k will be adjacent in π_k. Using that, we can simply iterate through the adjacent lines in π_k, check in constant time if they have already intersected, and find such pair whose intersection has the largest x-coordinate, which will be t_k.

We do so in the linear-time algorithm GetCoordinate, Algorithm 19, that receives a set G of lines and a permutation π, and returns the corresponding t_k if $\pi = \pi_k$ for some k with $0 \leq k \leq N$.

Algorithm 19 GetCoordinate(G, π)

Input: a set G of lines and a permutation π
Output: t_k if $\pi = \pi_k$ for some k

```
1: t ← −∞
2: for i ∈ [1, |G| − 1] do
3:     if (G[π[i]].m < G[π[i + 1]].m) then
4:         t ← max{t, Intersection(G[π[i]], G[π[i + 1]])}
5: return t
```

To exemplify the simulation of the algorithm, let us consider the set of lines in our input is the same as in Figure 5.4 and $\pi = (2, 4, 3, 1)$. Initially $t = -\infty$. We will iterate through the adjacent elements on π. Elements 2 and 4 are such that the slope of line 2 is greater than the one of line 4, meaning that line 2 appears before line 4 in π_0, so their intersection has not passed yet. Thus t remains unchanged.

Elements 4 and 3 are such that the slope of line 4 is smaller than the one of line 3, so their intersection, which happens to be t_2, already passed, hence $t = \max(t, t_2) = t_2$. Elements 3 and 1 are such that the slope of line 3 is smaller than the one of line 1, so their intersection, which happens to be t_4, already passed, hence $t = \max(t, t_4) = t_4$. There are no more adjacent pairs so that is the final value for t. Since $\pi = \pi_4$, t_4 is indeed the answer we were looking for.

5.3 Determining the permutation

To efficiently determine π_k, we will apply some comparison-based sorting algorithm to order G according to the intersection with the line $x = t_k$.

A comparison-based sorting algorithm works by asking queries with the following format: is element i greater than element j? Such query will be answered by some comparison operation. There are algorithms that, to sort an array of size n, only need to answer $\mathcal{O}(n \log n)$ such queries. It can be proven that no algorithm can sort an array with $o(n \log n)$ questions.

The comparison operation has the same form as the function IsGreater discussed in Section 4.3, but it refers to the order in which the lines in G intersect the line $x = t_k$. That is, to determine π_k, the queries will be: given a set G of lines, two integers i and j ($1 \leq i, j \leq |G|$), and an integer k ($0 \leq k \leq \binom{n}{2}$), is $\pi_k^{-1}(i) > \pi_k^{-1}(j)$? If such query can be answered by some function IsGreater(G, i, j, k), we can construct an algorithm that executes a sorting network and obtains π_k, just as Algorithm 17, but with the extra parameter k, as shown in Algorithm 20.

Algorithm 20 RunNetwork(G, H, k)

Input: a set G of lines, a network H of width $|G|$, and an integer k ($0 \leq k \leq \binom{n}{2}$)
Output: the permutation π_k

```
1: π ← (1, ..., |G|)
2: for P ∈ H do
3:     for (p, q) ∈ P do
4:         if IsGreater(G, π[p], π[q], k) then π[p] ↔ π[q]
5: return π
```

The final algorithm to find t_k is shown in Algorithm 21, and uses the function SortingNetwork(n) that returns a sorting network of width n. It also uses the functions RunNetwork, described above, and GetCoordinate, that was described in Section 5.2.

Algorithm 21 FindIntersection(G, k)

Input: a set G of lines and an integer k
Output: the value of t_k

```
1: H ← SortingNetwork(|G|)
2: π ← RunNetwork(G, H, k)
3: return GetCoordinate(G, π)
```

Now let us discuss how to implement IsGreater(G, i, j, k). Let $u_{i,j}$ be the x-coordinate of the intersection of $G[i]$ and $G[j]$. We know that the lines $G[i]$ and $G[j]$

intersect at t_l for some l. We can find the value of l by finding π_l using IndSortEval, described in Section 3.2.1, to sort the lines at $x = u_{i,j}$, and counting its number of inversions of π_l in relation to π_0. One of the following cases will happen:

If $l > k$, then $t_l > t_k$, so at $x = t_k$ the intersection between $G[i]$ and $G[j]$ has not happened yet. That way, the two lines $G[i]$ and $G[j]$ will have kept their initial slope order, given by π_0. Analogously, if $l \leq k$, the two lines will have already intersected at $x = t_k$, so their order will be the inverse of what their initial slope order π_0 was. This is exemplified in Figure 5.5.

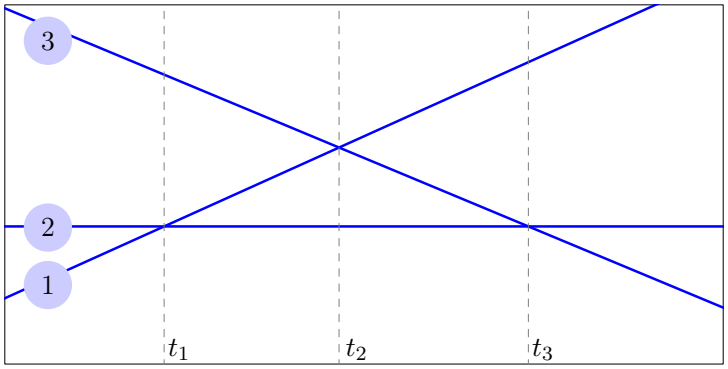


Figure 5.5: In this example, $\pi_0 = (1, 2, 3)$, $\pi_1 = (2, 1, 3)$, $\pi_2 = (2, 3, 1)$, and $\pi_3 = (3, 2, 1)$. Take lines 1 and 3, that intersect at t_2. Notice how 1 and 3 are in the same relative order as in π_0 for all π_k such that $k < 2$. On the other hand, for all π_k such that $k \geq 2$, 1 and 3 are inverted in relation to π_0

This leads to the implementation of IsGreater shown in Algorithm 22, and concludes the description of an algorithm to find t_k. Algorithm 22 uses the function Inversions, that takes the set G of lines and a permutation π as parameters, and returns the number of inversions in π in relation to the permutation π_0 that sorts G by decreasing slope. This function, similarly to Algorithm 8, can be implemented to run in time $\mathcal{O}(|G| \log |G|)$.

Algorithm 22 IsGreater(G, i, j, k)

Input: a set G of lines, indices i and j with $1 \leq i, j \leq |G|$, and an integer k
Output: TRUE if $\pi_k^{-1}(i) > \pi_k^{-1}(j)$ and FALSE otherwise

```
1: t ← Intersection(G[i], G[j]).x
2: π ← IndSortEval(G, t)
3: l ← Inversions(G, π)                    ▷ inversions with respect to π_0
4: if l ≤ k then                           ▷ t = t_l ≤ t_k?
5:     return G[i].m > G[j].m              ▷ π_0^{-1}(i) < π_0^{-1}(j)
6: else
7:     return G[i].m < G[j].m              ▷ π_0^{-1}(i) > π_0^{-1}(j)
```

Let us see what a call like IsGreater$(G, 1, 3, 3)$ would look like, where G is the set of lines in Figure 5.5. Firstly we have that the intersection of lines 1 and 3 is t_2, so $t = t_2$. The permutation π is the one that sorts G at t_2, that is $\pi = \pi_2 = (2, 3, 1)$. The number of inversions of π_2 with respect to π_0 is 2, so $l = 2$. Since $l \leq k = 3$, we enter line 5, and return whether the slope of line 1 is greater than the slope of line 3,

which is the case. That way, our function concludes that $\pi_3^{-1}(1) > \pi_3^{-1}(3)$, which is the case since $\pi_3^{-1}(1) = 3$ and $\pi_3^{-1}(3) = 1$, as can be seen in the figure.

The final time complexity of Algorithm 21 is the time complexity of IsGreater, which is $\mathcal{O}(|G| \log |G|)$, multiplied by the number of queries done by the sorting network, also $\mathcal{O}(|G| \log |G|)$. This leads to a final time complexity of $\mathcal{O}(|G|^2 \log^2 |G|)$.

As you can see, this is worse than the naive solution, which would be to simply find all intersections and sort them by x-coordinate, which can be done in $\mathcal{O}(|G|^2 \log |G|)$. However, we can now start reducing the time complexity by making certain parts of the algorithm more efficient. The first aspect to be changed is to replace the sorting algorithm by a sorting network.

5.4 Efficiently answering queries on a sorting network

To make FindIntersection more efficient, we will modify some aspects of the procedure RunNetwork so we can benefit from some properties of the ham sandwich point problem.

Consider a sorting network H. Each parallel step of H gives us a set P of queries. Let $u_{i,j}$ be the x-coordinate of the intersection of the lines $G[i]$ and $G[j]$. Let the median comparator in P be its comparator (a, b) that has the median value of $u_{a,b}$.

We will perform a binary search of sorts to answer half of the queries in P at once. Consider the median comparator (a, b) of P. By using the algorithms IndSortEval and Inversions, described in Section 5.3, we can compute the number of inversions of the permutation that sorts the lines in G according to their intersection with the vertical line at $x = u_{a,b}$. That permutation is π_l for some l. We will use the fact that $u_{a,b}$ is the median among the x-coordinates corresponding to the queries in P to obtain information on other queries of P.

For example, suppose that $l > k$. Then we know that $u_{a,b} = t_l > t_k$ and $G[a]$ and $G[b]$ are in the same order in π_k as they are in π_0. So if $u_{a,b} > t_k$, then for every $(p, q) \in P$ such that $u_{p,q} > u_{a,b}$, we also have $u_{p,q} > t_k$. That way, p and q are also in the same order in π_k as they are in π_0. Since (a, b) is the median comparator of P, we can answer half of the queries in P in this case.

If $l < k$, we can make an analogous reasoning to conclude that we can also answer half of the queries in P, this time for every $(p, q) \in P$ such that $u_{p,q} < u_{a,b}$. If $l = k$, we have found our answer, that is, $t_k = u_{a,b}$, and can exit the algorithm completely.

With that, our algorithm will be as shown in Algorithm 23. It uses the functions SortingNetwork, IsGreater, and GetCoordinate, that were previously defined and the function Partition that receives a set G of lines, a permutation π, a parallel step P of a network of width $|G|$, and two indices i and j such that $1 \leq i \leq j \leq |P|$. This function reorders the comparators of $P[i\,..\,j]$ in such a way that the median comparator (a, b) of $P[i\,..\,j]$ ends up exactly in $P[\left\lceil \frac{i+j}{2} \right\rceil]$, all comparators $(p, q) \in P$ such that $u_{p,q} < u_{a,b}$ end up in $P[i\,..\,(\left\lceil \frac{i+j}{2} \right\rceil - 1)]$, and all comparators $(p, q) \in P$ such that $u_{p,q} > u_{a,b}$ end up in $P[(\left\lceil \frac{i+j}{2} \right\rceil + 1)\,..\,j]$. To do so, it finds the median comparator using a function similar to QuickSelect, described in Section 3.2.2. The permutation π is passed as a parameter to Partition so that it accesses the lines in G in the order in which they are just before step P.

Algorithm 23 FindIntersection(G, k)

Input: a set G of lines and an integer k
Output: the value of t_k

```
1: H ← SortingNetwork(|G|)
2: π ← (1, ..., |G|)
3: for P ∈ H do
4:     i ← 1,   j ← |P|
5:     while i < j do
6:         Partition(G, π, P, i, j)
7:         m ← ⌈(i+j)/2⌉
8:         (a, b) ← P[m]
9:         if (G[π[a]].m > G[π[b]].m) = IsGreater(G, π[a], π[b], k) then
10:            for (p, q) ∈ P[i..m] do
11:                if G[π[p]].m > G[π[q]].m then π[p] ↔ π[q]
12:            i ← m + 1
13:        else
14:            for (p, q) ∈ P[(m+1)..j] do
15:                if G[π[p]].m < G[π[q]].m then π[p] ↔ π[q]
16:            j ← m − 1
17: return GetCoordinate(G, π)
```

By reducing the number of unanswered queries in a step by half repeatedly, we can solve a parallel step of the network H of width $|G|$ with only $\mathcal{O}(\log|G|)$ calls to IsGreater. Since we have $\Theta(\log|G|)$ parallel steps and the function IsGreater takes $\mathcal{O}(|G|\log|G|)$ time, the current time complexity of FindIntersection as in Algorithm 23 is $\mathcal{O}(|G|\log^3|G|)$.

This is still not linear, but it is already better than the naive algorithm. Now we will need to insert approximation ideas to remove the extra $\mathcal{O}(\log^3|G|)$ multiplier. Specifically, we will introduce three approximation ideas independently, and each one of them will eliminate a $\log|G|$ from the time complexity of the algorithm.

5.5 Approximately finding the t_k

We will modify the algorithm FindIntersection in three steps, each one of them will remove a $\log|G|$ from its time complexity, substituting it with some function of an approximation parameter, but will also make its answer more inaccurate. As long as the approximation parameter is a constant, any function of it will also be so. That way, once we perform all approximations, we will have reached a linear algorithm.

The three approximations will be:

- Substituting the use of IsGreater by a function that approximately counts the number of inversions in a permutation π_l
- Substituting the sorting network with a tolerant ϵ-sorting network;
- Answering half the queries on each network step a constant number of times, instead of doing it until all queries are answered.

Meanwhile, we will also eliminate the need for the usage of the function GetCoordinate, that does not work for approximate versions of π_k.

Now let us go into further details on how to introduce each of the approximations.

5.5.1 Counting intersections in linear time

This section will present a function ApproximateIntersections(G, t, δ) that receives a set G of lines in general position, an x-coordinate t, and a number δ with $0 < \delta < 1$, and returns a number that differs by no more than $\delta\binom{|G|}{2}$ from the number of intersections among lines in G that have x-coordinate no greater than t. The running time of this function will be $\mathcal{O}(|G| \log(\frac{1}{\delta}))$.

One exact way to do this, similar to Algorithm 22, is sorting the lines in G by evaluation at $t_0 = -\infty$ (which is sorting by decreasing slope), calculating the evaluation of each line at $x = t$ (resolving ties using the slopes), and counting the inversions in the resulting array. However, we cannot do the sorting or the counting because each of these alone would cost more than linear time. On the other hand, we only need to determine approximately the number of such inversions.

The approach we will use, presented in Algorithm 24, is very similar to that. The difference is that the sorting and the inversion counting are both going to be approximate.

Algorithm 24 ApproximateIntersections(G, t, δ)

Input: a set G of lines, an x-coordinate t, and a number δ with $0 < \delta < 1$

Output: the number of intersections among lines in G that have x-coordinate not greater than t, wrong by no more than $\delta\binom{|G|}{2}$. As a side effect, the algorithm rearranges G.

1: ApproximateSlopeSort$(G, \delta/2)$
2: $ans \leftarrow$ ApproximateInversions$(G, t, \delta/2)$
3: **return** ans

The following definition will help us to formally describe the output of the two algorithms used in ApproximateIntersections.

Definition 9. *Let A be an array of size n. If A is split into no more than $\frac{2}{\epsilon}$ segments, each with at most $\max\{1, \epsilon n\}$ consecutive elements of A, and every inversion in A with respect to an order is inside some of these segments, we say that A is* ϵ-bucket-sorted *with respect to that order.*

Algorithm ApproximateSlopeSort(G, ϵ) rearranges the set G of lines so that it is ϵ-bucket-sorted according to decreasing slope. It runs in $\mathcal{O}(|G| \log(\frac{1}{\epsilon}))$. Algorithm ApproximateInversions(G, t, ϵ) returns the number of inversions in G, off by no more than $\epsilon\binom{|G|}{2}$, with respect to the order of the evaluation of the lines at $x = t$, with ties sorted by increasing slope of the lines. As a side effect, it rearranges G so that it is ϵ-bucket-sorted according to the evaluation of the lines at $x = t$. It also runs in $\mathcal{O}(|G| \log(\frac{1}{\epsilon}))$.

From this, clearly Algorithm 24 runs in $\mathcal{O}(|G| \log(\frac{1}{\delta}))$. Now let us argue that Algorithm 24 is correct. Let I be the number of intersections among lines in G that have x-coordinate not greater than t. We have to prove that $|ans - I| \leq \delta\binom{|G|}{2}$. Note

that I is exactly the number of inversions that G, sorted by evaluation at $x = t$, has in relation to G sorted by decreasing slope of the lines.

Let G_0 be the state of G after line 1 was executed. Let I_0 be the number of inversions in G_0 with respect to the order of the evaluation of the lines at $x = t$, with ties sorted by increasing slope of the lines. By the description of algorithm ApproximateIntersections, we have that $|ans - I_0| \leq \frac{\delta}{2}\binom{|G|}{2}$, that is,

$$ans - \frac{\delta}{2}\binom{|G|}{2} \;\leq\; I_0 \;\leq\; ans + \frac{\delta}{2}\binom{|G|}{2}.$$

On the other hand, as G_0 is $\frac{\delta}{2}$-bucket sorted with respect to the decreasing slope order by the description of ApproximateSlopeSort, the number I' of inversions that G_0 may have with respect to this order is such that

$$0 \leq I' \leq \frac{4}{\delta}\frac{(\frac{\delta}{2}n)(\frac{\delta}{2}n-1)}{2} = \frac{\delta}{2}n^2 - n < \frac{\delta}{2}\binom{n}{2}.$$

Therefore, $I \leq I_0 + I' \leq ans + \delta\binom{|G|}{2}$ and $I \geq I_0 - I' \geq ans - \delta\binom{|G|}{2}$.

Now we will describe ApproximateSlopeSort and ApproximateInversions in more detail.

Algorithm ApproximateSlopeSort(G, ϵ) rearranges the set G of lines so that it is ϵ-bucket-sorted according to decreasing slope. It does so by calling the recursive function ApproximateSlopeSortRec(G, ϵ, i, j), that rearranges the set $G[i\,..\,j]$ of lines so that it is ϵ-bucket-sorted according to decreasing slope. Algorithm ApproximateSlopeSortRec is a recursive divide-and-conquer algorithm, similar to Quicksort. It uses the function PartitionInHalf to divide the array in two halves in such a way that there are no inversions between lines of different halves with respect to their slope. We will do so in such a way that G is divided into no more than $\frac{1}{\epsilon}$ buckets of size at most $\lceil \epsilon n \rceil$ each, leaving it ϵ-bucket-sorted.

The final procedure ApproximateSlopeSort will be as shown in Algorithm 25. It calls the function ApproximateSlopeSortRec, shown in Algorithm 26.

Algorithm 25 ApproximateSlopeSort(G, ϵ)

Input: a set G of lines and a number ϵ such that $0 < \epsilon < 1$
Output: G is now ϵ-bucket-sorted decreasingly by slope
1: ApproximateSlopeSortRec$(G, 1, |G|, \epsilon)$

Algorithm 26 ApproximateSlopeSortRec(G, i, j, ϵ)

Input: a set G of lines, integers i and j with $1 \leq i, j \leq |G|$, and a number ϵ with $0 < \epsilon < 1$

Output: $G[i\,..\,j]$ is now ϵ-bucket-sorted decreasingly by slope

1: **if** $j - i + 1 > \max\{1, \epsilon|G|\}$ **then**
2: PartitionInHalf(G, i, j)
3: ApproximateSlopeSortRec$(G, i, \left\lfloor \frac{i+j}{2} \right\rfloor, \epsilon)$
4: ApproximateSlopeSortRec$(G, \left\lfloor \frac{i+j}{2} \right\rfloor + 1, j, \epsilon)$

Where PartitionInHalf is described in Algorithm 27. It receives a set G of lines in general position and indices i and j with $1 \leq i \leq j \leq |S|$. It rearranges $G[i\,..\,j]$ so that, for $h = \lfloor (i + j)/2 \rfloor$, all lines in $G[i\,..\,h]$ have slope greater than the lines in $G[h + 1\,..\,j]$. We use $G[i\,..\,h].m < G[h{+}1\,..\,j].m$ to express this condition. Moreover, the order of the lines in $G[i\,..\,h]$ and in $G[h + 1\,..\,j]$ is exactly the order they had in G at beginning of the procedure.

Algorithm 27 PartitionInHalf(G, i, j)

Input: a set G of lines in general position and indices i and j with $1 \leq i \leq j \leq |G|$

Output: rearranges G so that $G[i\,..\,h].m < G[h{+}1\,..\,j].m$ for $h{=}\left\lfloor \frac{i+j}{2} \right\rfloor$

1: **if** $i < j$ **then**
2: $k \leftarrow$ QuickSelectRec$(G, i, j, \left\lfloor \frac{i+j}{2} \right\rfloor)$
3: $pivot \leftarrow G[k].m$ ▷ median slope of lines in $G[i\,..\,j]$
4: $G' \leftarrow G[i\,..\,j]$
5: $t_1 \leftarrow i, \quad t_2 \leftarrow \left\lfloor \frac{i+j}{2} \right\rfloor$
6: **for** $t \in [i, j]$ **do**
7: **if** $G'[t].m \geq pivot$ **then**
8: $G[t_1] \leftarrow G'[t]$
9: $t_1 \leftarrow t_1 + 1$
10: **else**
11: $G[t_2] \leftarrow G'[t]$
12: $t_2 \leftarrow t_2 + 1$

Considering that QuickSelectRec runs in time $\mathcal{O}(j - i)$, as discussed in Section 3.2.2, we can easily conclude that PartitionInHalf(G, i, j) also runs in time $\mathcal{O}(j - i)$. In a similar way in which one can prove the time complexity of the Mergesort algorithm, since ApproximateSlopeSortRec runs in $\mathcal{O}(j-i)$ time, not considering the recursive calls, the complexity of ApproximateSlopeSort will be $\mathcal{O}(nD)$, where $D = \log \frac{1}{\delta}$ is the depth of the recursion of ApproximateSlope-SortRec. Therefore, the complexity of ApproximateSlopeSort is $\mathcal{O}(n \log \frac{1}{\epsilon})$.

Now let us discuss the accuracy of the algorithm.

Notice that there are no inversions (in relation to the slope order of the lines) among lines of different buckets. Since each bucket has size at most $\lceil \epsilon n \rceil$, the number of inversions in each bucket is at most $\frac{(\epsilon n)(\epsilon n - 1)}{2}$. That means that, by the end of

that process, the total number of inversions in the array with respect to π_0 will be:

$$\frac{1}{\epsilon}\frac{(\epsilon n)(\epsilon n-1)}{2} = \frac{\epsilon n^2 - n}{2} \leq \epsilon\binom{n}{2}.$$

Another important aspect to note is that an element position will not be wrong by more than δn.

Now let us discuss ApproximateInversions.

Algorithm ApproximateInversions(G, t, ϵ) returns the number of inversions in G, off by no more than $\epsilon\binom{|G|}{2}$, with respect to the order of the evaluation of the lines at $x = t$, with ties sorted by increasing slope of the lines. As a side effect, it rearranges G so that it is ϵ-bucket-sorted according to the evaluation of the lines at $x = t$. It does so by calling the recursive function ApproximateInversionsRec(G, i, j, t, ϵ), that does the same as ApproximateInversions, but with respect to $G[i\,..\,j]$ instead of G.

The final procedure ApproximateInversions will be as shown in Algorithm 28. It calls the function ApproximateInversionsRec, shown in Algorithm 29.

Algorithm 28 ApproximateInversions(G, t, ϵ)

Input: a set G of lines, an x-coordinate t, and a number r
Output: An approximate number of inversions with respect to the input set G. As a side effect, the algorithm rearranges G.

1: **return** ApproximateInversionsRec$(G, 1, |G|, t, \epsilon)$

Algorithm 29 ApproximateInversionsRec(G, i, j, t, ϵ)

Input: a set G of lines, integers i and j such that $1 \leq i, j \leq |G|$, an x-coordinate t, and a number ϵ
Output: An approximate number of inversions with respect to the input set $G[i\,..\,j]$. As a side effect, the algorithm rearranges $G[i\,..\,j]$.

1: $inv \leftarrow 0$
2: **if** $j - i + 1 > \max\{1, \epsilon|G|\}$ **then**
3: $\quad inv \leftarrow$ SeparateInHalf(G, i, j, t)
4: $\quad +$ApproximateInversionsRec$(G, i, \left\lfloor \frac{i+j}{2} \right\rfloor, t, \epsilon)$
5: $\quad +$ApproximateInversionsRec$(G, \left\lfloor \frac{i+j}{2} \right\rfloor + 1, j, t, \epsilon)$
6: **return** inv

Where SeparateInHalf is described in Algorithm 30. It receives a set G of lines in general position, indices i and j with $1 \leq i \leq j \leq |S|$ and an x-coordinate t. It rearranges $G[i\,..\,j]$ so that, for $h = \lfloor (i + j)/2 \rfloor$, all lines in $G[i\,..\,h]$ have evaluation at t smaller than the lines in $G[h + 1\,..\,j]$. We use $G[i\,..\,h](t) < G[h{+}1\,..\,j](t)$ to express this condition. Moreover, the order of the lines in $G[i\,..\,h]$ and in $G[h{+}1\,..\,j]$ is exactly the order they had in G at beginning of the procedure. The algorithm also returns the number of inversions $G[i\,..\,j]$ had among elements that are now in different halves.

The algorithm uses as an auxiliary function the function FindLineOfMedianEvaluation(G, i, j, t), that returns the index of the line in $G[i\,..\,j]$ that has median evaluation at t.

Algorithm 30 SeparateInHalf(G, i, j, t)

Input: a set G of lines, integers i and j such that $1 \leq i, j \leq |G|$ and an x-coordinate t

Output: returns the number of inversions along the elements in different halves of $G[i\,..\,j]$ when sorted by evaluation at t. The function also rearranges G so that $G[i\,..\,h](t) < G[h{+}1\,..\,j](t)$ for $h = \left\lfloor \frac{i+j}{2} \right\rfloor$

```
1: m ← FindLineOfMedianEvaluation(G, i, j, t)
2: y ← G[m](t)                          ▷ median y-coordinate in G[i..j] at x-coordinate t
3: G' ← G[i..j]
4: t1 ← i,   t2 ← ⌊(i+j)/2⌋
5: inv ← 0
6: for ℓ ∈ G' do
7:     if ℓ(t) ≤ y then
8:         G[t1] ← ℓ
9:         t1 ← t1 + 1
10:        inv ← inv + t2 − ⌊(i+j)/2⌋
11:    else
12:        G[t2] ← ℓ
13:        t2 ← t2 + 1
14: return inv
```

Similarly to PartitionInHalf, SeparateInHalf(G, i, j, t) runs in time $\mathcal{O}(j - i)$. From there, we can see that ApproximateInversionsRec has the same complexity than ApproximateSlopeSortRec and ApproximateInversions has the same complexity than ApproximateSlopeSort. Therefore, the complexity of ApproximateInversions is $\mathcal{O}(n \log \frac{1}{\epsilon})$.

Now let us discuss the accuracy of the algorithm.

What we need to do now is to bound the error on the number of inversions counted by the algorithm that was just described in relation to the actual number of inversions of π_k in relation to G, where $t_k = t$. This algorithm rearranges G using the exact same method of ApproximateSlopeSort, but with a different order. Instead of the slope order, it uses the order given by the intersection of the lines in G with the vertical line at $x = l$. Meanwhile, it counts the number of inversions undone while sorting G. As the sorting is not complete, stopping with a δ-bucket-sorted array, some inversions of the complete order at $x = l$ were not counted.

Just like in the previous algorithm, in the deepest level of the recursion, we will have no more than $\frac{1}{\epsilon}$ buckets of size at most ϵn each. This introduces $\frac{(\epsilon n)(\epsilon n - 1)}{2}$ inversions for the order of the lines.

5.5.2 Changing the sorting network

This section will describe two approximations to be applied to FindIntersection, that will result in the implementation of FindApproximateIntersection. Recall that the latter receives a set G of lines in general position, an integer k such that $1 \leq k \leq \binom{|G|}{2}$, and a real number δ with $0 < \delta < 1$, and returns an x-coordinate a for which the number s of intersections of lines in G to the left of the vertical line $x = a$ satisfies $|k - s| \leq \delta\binom{|G|}{2}$. The implementation must run in $\mathcal{O}(|G| \log^3 \frac{1}{\delta})$ time.

The first approximation reduces the $\Theta(\log |G|)$ depth of the sorting network by using a network of depth $\mathcal{O}(\log(\frac{1}{\delta}))$. The second approximation consists of using a t-tolerant ϵ-sorting network, therefore allowing us to delete some comparators from the network while it remains an ϵ-sorting network. More specifically, we will use a tolerant δ-sorting network. That way, instead of obtaining the k-th intersection, we will obtain an intersection that might be off by at most δN from the k-th intersection, where $N = \binom{|G|}{2}$.

To exploit the tolerance we will modify Algorithm 23. Similarly to such algorithm, for each parallel step of the network, we will iteratively answer half of the remaining queries in the parallel step. In Algorithm 23, we do this until all queries in the parallel step have been answered. In FindApproximateIntersection, we will leave some queries unanswered. That corresponds to deleting the corresponding comparators from the network. We will leave at most $\delta|G|$ queries unanswered from each parallel step. Since the number of queries in a parallel step is at most $|G|$, this means that the maximum number of executions of line 5 of Algorithm 23 will go from $\mathcal{O}(\log |G|)$ to $\mathcal{O}(\log(\frac{1}{\delta}))$.

Since the number of deleted comparators can be as big as $\delta|G|$ per parallel step, if the network we are performing this algorithm has depth $\mathcal{O}(\log(\frac{1}{\delta}))$, the number of comparators removed will be $\mathcal{O}(\delta|G| \log(\frac{1}{\delta}))$. If the network is a t-tolerant δ-sorting network for an appropriate $t = \Omega(\delta|G| \log(\frac{1}{\delta}))$, the resulting algorithm is simulating a δ-sorting network.

Recall that TolerantSortingNetwork(n, ϵ), that returns a t-tolerant ϵ-sorting network of width n and depth at most $C \log(\frac{1}{\epsilon})$, for C given by Corollary 1, for $t < \epsilon^{C+1} n/4$, that runs in time $\mathcal{O}(n \log(\frac{1}{\epsilon}))$. We will use $t = t_0 = \frac{\delta C+1|G|}{4}$.

The existence of this network allows us to implement the modified version of the algorithm, as long as we set the correct numbers for some variables on the Algorithm, which we will do according to Section 3 of [8], with particular help from Lemma 3.4.

The algorithm after both modifications have been applied will remain largely unchanged, as can be seen in its new implementation, shown in Algorithm 31.

First of all we can see the whenever the algorithm returns some value z (line 10), this value is t_l for some l. This is because z is defined as the x-coordinate of the intersection of two lines in G. By the definition of ApproximateIntersections, we know that $|k_z - l| \leq \delta\frac{N}{3}$. With that, we know that the conditional on line assures that $|k - l| \leq |k - k_z| + |k_z - l| \leq \delta N$.

This assures that the number the algorithm returns is t_l where l is in the range $[k - \delta N, k + \delta N]$, just like we wanted. This is, of course, conditional on the correct functioning of the network. We know the functioning of the network is conditional on the number of comparators that have been deleted to be less than t, or, in our case, $t_0 = \frac{\delta C+1|G|}{4}$. Notice that the condition on line 4 assures that the number

Algorithm 31 FindApproximateIntersection(G, k, δ)

Input: a set G of lines, an integer k, and a number δ such that $0 < \delta < 1$

Output: the value of t_l where l is in the range $[k - \delta N, k + \delta N]$. As a side effect, the function rearranges G

1: $H \leftarrow$ TolerantSortingNetwork$(|G|, \delta)$ ▷ $|H| \leq C \log(\frac{1}{\delta})$
2: **for** $P \in H$ **do**
3: $\quad i \leftarrow 1, \quad j \leftarrow |P|$
4: $\quad$ **while** $j - i \geq \delta^{C+1} \frac{|G|}{4|H|}$ **do**
5: $\quad\quad$ Partition(G, P, i, j)
6: $\quad\quad m \leftarrow \left\lceil \frac{i+j}{2} \right\rceil$
7: $\quad\quad (a, b) \leftarrow P[m]$
8: $\quad\quad z \leftarrow$ Intersection$(G[a], G[b]).x$ ▷ $z = u_{a,b}$
9: $\quad\quad k_z \leftarrow$ ApproximateIntersections$(G, z, \delta/3)$
10: $\quad\quad$ **if** $|k_z - k| \leq 2\delta N/3$ **then return** z
11: $\quad\quad$ **if** $k_z < k - 2\delta N/3$ **then**
12: $\quad\quad\quad$ **for** $(p, q) \in P[i \,..\, m]$ **do** ▷ $u_{p,q} < u_{a,b} \leq t_k$
13: $\quad\quad\quad\quad$ **if** $G[p].m > G[q].m$ **then** $G[p] \leftrightarrow G[q]$
14: $\quad\quad\quad i \leftarrow m + 1$
15: $\quad\quad$ **else**
16: $\quad\quad\quad$ **for** $(p, q) \in P[(m + 1) \,..\, j]$ **do** ▷ $u_{p,q} > u_{a,b} > t_k$
17: $\quad\quad\quad\quad$ **if** $G[p].m < G[q].m$ **then** $G[p] \leftrightarrow G[q]$
18: $\quad\quad\quad j \leftarrow m - 1$

of comparators deleted per step of the network will be no more than $\delta^{C+1} \frac{|G|}{4|H|}$. Since we have $|H|$ steps, the total number of deleted operators will be at most $|H|\delta^{C+1} \frac{|G|}{4|H|} = \frac{\delta C+1|G|}{4} = t_0$, which assures the correct functioning of the network.

Something else that need to proved is that the algorithm always returns an answer. The proof for that will be delegated to Lemma 3.9 of [8].

The last thing that needs to be investigated is the running time of the algorithm. Firstly, let us see what happens in one iteration of the while in line 4. We have one call of partition, that takes $\mathcal{O}(j-i)$ time, one call of ApproximateIntersections, that takes $\mathcal{O}(|G| \log \frac{1}{\delta})$ time and some iterations, each one taking at most $|G|$ time. Therefore, that while takes at most $\mathcal{O}(|G| \log \frac{1}{\delta})$ time.

In each while iteration, the size of $j-i$ it deterministically cut by half. This means it will run in $\mathcal{O}(\log(\delta^{C+1} \frac{|G|}{4|H|}))$ time, which, since $|H| < |G|$ and C is a constant, is $\mathcal{O}(\log \frac{1}{\delta})$.

Lastly, there is the number of iterations of the for at line 2 the depth of the network: $\mathcal{O}(\log \frac{1}{\delta})$.

Therefore, the overall time complexity of FindApproximateIntersection is $\mathcal{O}(|G| \log^3 \frac{1}{\delta})$.

With that, the linear implementation of NewInterval, and, by consequence, HamSandwichPoint, is finalized.

Chapter 6

Final considerations

Finding linear algorithms is hard. It is famously known that one cannot sort an array of n elements using comparison based operations in $o(n \log n)$ time. The algorithms that sort arrays also provide us with other useful results, such as the number of inversions in an array, which is extensively sought throughout the algorithm described in this work.

Not being able to sort an array in linear time leaves us with a much more limited set of techniques, such as transforming the time complexity into a geometric progression or only selecting the k-th element in an array instead of sorting it. We can also use approximate versions of the more powerful $\mathcal{O}(n \log n)$ algorithms, which leads to a more complicated time complexity analysis and very slow running time, regardless of the asymptotic time complexity, often much worse than the $\log n$ factor we were trying to avoid in the first place.

This leads to algorithms that use amazingly intricate techniques but are likely very impractical to read, code and execute.

A large difficulty I had with trying to understand the linear solution for the two dimensional ham sandwich problem was that the depth of the solution was not immediately visible. Once I understood most outline of the algorithm presented by Lo, Matoušek, and Steiger in [7], which is mostly what is described in Section 3, I had to dig deeper into how to find an approximate intersection, using the method described in [8], which in turn relied heavily on the AKS sorting network described in [1], which in turn uses already complicated algorithms for building expander graphs.

Even the most low-level of those concepts, such as the algorithms for the expander graphs, are very scarce in their implementations since the constants hidden in the time complexities make them almost unusable for practical purposes. This is accentuated by the fact that there are $\mathcal{O}(n \log n)$ algorithms that work just fine and will be faster than the $\mathcal{O}(n)$ ones for any feasible size input.

I now believe the purpose of this work to be very different from what I originally intended. I originally wanted to implement the linear time algorithm and exercise the process of transforming a very mathematical and not code-oriented paper into something executable. I now know that this task is much harder than I anticipated, and would require a large amount of work to produce something very impractical.

What this work really taught me, and I hope can teach the ones interested in reading it, is just how much theory and how many fun topics can fit into a seemingly simple paper designed to solve a simple problem as fast as it can be done.

6.1 Acknowledgements

I want to thank my advisor, Cristina G. Fernandes, for introducing me to this topic, helping me figure out the algorithms, working with me to find alternative solutions, reviewing this work many, many times and teaching me how to write mathematically. I want to thank Pedro Teotonio, for working with me to figure out the original paper when this problem was originally presented to us as the final task of a Computational Geometry course two years ago. I want to thank Victor Colombo, for helping me review the text and test my explanation of novel concepts. I want to thank Yoshiko Wakabayashi, Marcel Kenji and Yoshiharu Kohayakawa, for teaching me all these computational theory concepts that I could finally apply. And finally, I want to thank the group MaratonUSP, that made me interested in algorithms in the first place.

Bibliography

[1] M. Ajtai, J. Komlós, and E. Szemerédi. An $0(n \log n)$ sorting network. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing (STOC)*, pages 1–9, 1983.

[2] Marshall W. Bern and David Eppstein. Multivariate regression depth. *CoRR*, cs.CG/9912013, 1999.

[3] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973.

[4] R. Cole, J. S. Salowe, W. L. Steiger, and E. Szemerédi. An optimal-time algorithm for slope selection. *SIAM Journal on Computing*, 18(4):792–810, 1989.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[6] Donald Knuth. *The Art Of Computer Programming, vol. 3: Sorting and Searching*. Addison-Wesley, 1973.

[7] C.-Y. Lo, J. Matoušek, and W. Steiger. Algorithms for ham-sandwich cuts. *Discrete & Computational Geometry*, 11:433–452, 1994.

[8] J. Matoušek. Construction of epsilon nets. In *Proceedings of the Fifth Annual Symposium on Computational Geometry*, SCG '89, pages 1—-10, New York, NY, USA, 1989. Association for Computing Machinery.

[9] Nimrod Megiddo. Partitioning with two lines in the plane. *Journal of Algorithms*, 6(3):430–433, 1985.

[10] Daniel G. O'Connor and Raymond J. Nelson. Sorting system with nu-line sorting switch, U.S. Patent 3029413A, Apr. 1962.