

# **On Modelling, Structuring and Capturing Geometric Information**

Typeset in L<sup>A</sup>T<sub>E</sub>X.

Cover and figures designed using the Ipe extensible drawing editor.

Copyright © 2022 Ivor van der Hoog.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission of Ivor van der Hoog

ISBN 978-94-6458-049-5

# **On Modelling, Structuring and Capturing Geometric Information**

**Over het Modelleren, Structureren en Vastleggen van  
Geometrische Informatie**  
(met een samenvatting in het Nederlands)

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Universiteit Utrecht  
op gezag van de rector magnificus, prof.dr. H.R.B.M Kummeling,  
ingevolge het besluit van het college voor promoties  
in het openbaar te verdedigen op

Maandag 14 Februari 2022

des middags te

4:15 uur

door

Ivor Djinn van der Hoog

Promotor: Prof.dr. M.J. van Kreveld  
Co-promotor: Dr. M. Löffler

Dit proefschrift werd (mede) mogelijk gemaakt met financiële steun van de NWO  
grantnummer 614.001.504.



# Preface

This thesis marks the end of my PhD. This thesis compounds and summarises four years of research in computational geometry. It is the conclusion of a wonderful phase in my life which was filled with making friends, doing research and teaching. Before I present the content of this thesis, I want to look back upon how this all came to be:

I did not always aspire to do a PhD. In fact, I aspired to be *many* things, and for a long time being an academic was not amongst those options. Throughout my life I considered various career paths including (but not limited to) musician, police officer, army officer, teacher, lawyer, program developer, and (perhaps the most alluring option of all) stay-at-home gamer. My aspiration to obtain a PhD originates from high school where I was inspired by two teachers with a doctorate. They were the first to introduce me to the wonder of chasing open problems and the academic teaching ethos. Later, this aspiration was bolstered through my many encounters with academic staff. When my bachelor program progress got off to a slow start, some professors devoted a part of their personal time and attention to keeping me engaged and inspired. Throughout the remainder of my studies I enjoyed the personal touch that many faculty staff added to their work. These personal interactions exposed me to the intellectual diversity that the university houses: from numerous academic topics, to wildly varying teaching philosophies, I was slowly exposed to all facets of the academic career path. Gradually I decided that I wanted to pursue a PhD with academic teaching and by the time that I started my masters, I had the definitive aspiration to start PhD in theoretical computer science.

Whilst I was set on starting research in theoretical computer science, doing a PhD in computational geometry specifically remained unlikely: as before the start of my PhD my only encounter with geometry was high-school geometry and university topology, which I both dreaded. Thus, even though I liked algorithmic analysis, pursuing career in computational geometry did not seem appealing at the time.

However, impatient as I am, I applied to the first position that became available in theoretical computer science and this happened to be computational geometry. It may have taken a while, but now I understand and appreciate the multi-faceted beauty of computational geometry. Computational geometry is home to all aspects of theoretical computer science: from exploring of open mathematical problems, to algorithmic complexity analysis. The unique thing about computational geometry is that it combines these theoretical aspects very tangible concepts such as configurations of intersecting disks or squares. In this way it facilitates algorithmic study, without it ever becoming too abstract. It is this aspect of the field that makes it very enjoyable to do research in and, in hindsight, I could not have hoped for a better topic.

I hope that this thesis is able to illustrate (at least partly) the beauty in computational geometry and that you enjoy reading it.

Ivor van der Hoog.



# Contents

|                 |            |
|-----------------|------------|
| <b>Preface</b>  | <b>i</b>   |
| <b>Contents</b> | <b>iii</b> |

## PART I Introduction

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Introduction</b>                          | <b>1</b>  |
| <b>2</b> | <b>Modelling, structuring and capturing</b>  | <b>15</b> |
| 2.1      | Part II: modelling geometric information     | 16        |
| 2.2      | Part III: structuring geometric information  | 19        |
| 2.3      | Part IV: capturing geometric information     | 22        |
| 2.4      | Additional research performed during the PhD | 24        |

## PART II Real RAM computations

|          |                              |           |
|----------|------------------------------|-----------|
| <b>3</b> | <b>Defining the real RAM</b> | <b>29</b> |
| 3.1      | Defining the real RAM        | 30        |

|          |  |           |
|----------|--|-----------|
| <b>4</b> | <b>Smoothed analysis of the real RAM</b>     | <b>39</b> |
| 4.1      | Smoothed analysis of the real RAM            | 44        |
| 4.2      | Smoothed analysis of the real RAM with roots | 51        |
| 4.3      | Concluding remarks                           | 54        |
| 4.A      | Varieties hitting cubes                      | 56        |
| 4.B      | Varieties hitting prisms                     | 59        |

## **PART III Structuring data**

|          |   |           |
|----------|---|-----------|
| <b>5</b> | <b>Visibility between moving entities</b>         | <b>63</b> |
| 5.1      | Introduction                                      | 63        |
| 5.2      | Semi-algebraic range searching                    | 67        |
| 5.3      | Algorithms for testing visibility                 | 69        |
| 5.4      | Testing curve intersection with a convex polygon  | 72        |
| 5.5      | Two moving entities inside a simple polygon       | 76        |
| 5.6      | One moving entity in a polygonal domain           | 81        |
| 5.7      | Two moving entities crossing a polygonal domain   | 87        |
| 5.8      | Concluding remarks                                | 89        |
| 5.A      | Transforming two entities into an algebraic curve | 90        |
| 5.B      | Dealing with degeneracies                         | 92        |
| <b>6</b> | <b>Fréchet distance between trajectories</b>      | <b>93</b> |
| 6.1      | Introduction                                      | 93        |
| 6.2      | Global approach                                   | 96        |
| 6.3      | Horizontal queries                                | 98        |
| 6.4      | Horizontal queries: querying for subcurves        | 108       |
| 6.5      | Arbitrary orientation queries                     | 117       |
| 6.6      | Space time tradeoff                               | 125       |
| 6.7      | Applications                                      | 128       |
| 6.8      | Concluding remarks                                | 130       |

|          |  |            |
|----------|--|------------|
| <b>7</b> | <b>Dynamic smooth compressed quadtrees</b> | <b>131</b> |
| 7.1      | Introduction                               | 132        |
| 7.2      | Preliminaries                              | 135        |
| 7.3      | Static smooth non-compressed quadtrees     | 138        |
| 7.4      | Dynamic smooth non-compressed quadtrees    | 146        |
| 7.5      | Point location in a quadtree               | 149        |
| 7.6      | Quadtree compression                       | 151        |
| 7.7      | Concluding remarks                         | 160        |

## **PART IV Imprecise points**

|           |  |            |
|-----------|--|------------|
| <b>8</b>  | <b>Geometric imprecision</b>                         | <b>167</b> |
| 8.1       | Introduction   | 169        |
| 8.2       | Results and implications                             | 171        |
| 8.3       | Algorithmic lower bounds                             | 173        |
| <b>9</b>  | <b>Imprecise order</b>                               | <b>181</b> |
| 9.1       | Introduction   | 182        |
| 9.2       | Ambiguity  | 185        |
| 9.3       | Sorting imprecise points in $O(A(\mathcal{R}))$ time | 193        |
| 9.4       | Concluding remarks                                   | 200        |
| <b>10</b> | <b>Imprecise staircases</b>                          | <b>201</b> |
| 10.1      | Introduction   | 202        |
| 10.2      | Geometric preliminaries                              | 204        |
| 10.3      | Lower bounds   | 209        |
| 10.4      | Geometric preliminaries for reconstruction           | 213        |
| 10.5      | Preprocessing phase                                  | 215        |
| 10.6      | Reconstruction phase                                 | 220        |
| 10.7      | Concluding remarks                                   | 232        |

|                           |            |
|---------------------------|------------|
| <b>Concluding remarks</b> | <b>235</b> |
| <b>Samenvatting</b>       | <b>239</b> |
| <b>Bibliography</b>       | <b>243</b> |
| <b>Curriculum Vitae</b>   | <b>259</b> |
| <b>Acknowledgements</b>   | <b>261</b> |

PART I

# Introduction



## Chapter One

# Introduction

This thesis is about how to model, structure or quantify geometric information (in geometric data). This work is the result of four years of research in computational geometry, a subfield of algorithm theory.

**Computer science** is a broad field of study. In a nutshell, it is the study of everything that has to do with computers and their computations: from the computer itself to the things a computer does, and the interactions that you and others have with the computer. This umbrella term includes, amongst others, the study of hardware (the mechanical bits and pieces that make a computer run), software (the programs that run on a computer), visualisation (the things you see in a program), algorithms (the sequence of steps that culminate in a program), networks and protocols (the communication between computers) and human interactions (the communication between humans and computers). The various topics within computer science can be studied on a practical or theoretical level. The difference between these two can be illustrated by the following example: suppose that we are studying the route of a delivery driver and we are interested in computing a fast route along all addresses. Practical research in computer science involves (often empirical) research performed with the use of real world computers. In the example, we may use a computer to construct various routes along all addresses and compare them to find a route with short length. Theoretical computer science involves research based on a theoretical model of a computer and allows us to reason about the limitations and capabilities of real world computers. In the example, we may reason about the various operations a computer has access to and in fact conclude that given an arbitrary set of many addresses, it is highly unlikely for an actual computer to find the route that passes all addresses that has minimal length (this property is called the *hardness* of the problem). This thesis presents research into algorithm theory in computational geometry and is part of theoretical computer science.

**Algorithm Theory** is dedicated to studying algorithmic problems. All of us are continuously presented with algorithmic problems and automatically solve many on a daily basis. In a nutshell, an algorithmic problem is a description of some class of input  $x$  and some desired output  $y$ . The procedure that transforms  $x$  into  $y$  is called an algorithm. In algorithm theory we:

- *model* algorithmic problems,
- *provide solutions* to algorithmic problems, and
- *evaluate* solutions of algorithmic problems.

**Modelling algorithmic problems.** When reading the above description you may wonder: ‘what does it mean to transform some input  $x$  into an output  $y$ ?’ Specifying this statement is called *modelling* an algorithmic problem, and a problem can be modelled to various degrees. First, you can state an algorithmic problem on an intuitive level. For example, suppose I refer to the problem of ‘baking a cake’, then you probably have some intuitive grasp of what task I am referring to. When modelling algorithmic problems, we subsequently try to make the problem description as accurate and precise as possible; this is called *formalising* the problem. Our example could be made more formal as follows: the input of the ‘baking a cake’ problem is some collection  $x$  of ingredients. The desired outcome  $y$  consists of an edible cake. An algorithm may sequence operations such as heating the oven, measuring the volume of ingredients, combining (mixing) ingredients and timing how long a mixture is present in the oven. If I told you that I would be baking a cake using the above description, then you might consider that description to be quite verbose. However, for the purpose of problem analysis and evaluation, the problem is actually not nearly precise enough. Indeed, suppose that I instead asked *you* to bake the cake. Then before you can analyse or solve the baking problem, you would need a much more elaborate description of the problem statement. A more formal definition would specify that the ingredients  $x$  of the cake is a collection of items, which I have not yet gathered. Thus, when solving the problem for me, you would need to go to the supermarket and purchase these items for me. Moreover, the intended output  $y$  may not be any cake but instead be restricted to a handmade banana flavor cake with chocolate sprinkles, which takes a moderate amount of effort to make. I could even further specify the algorithmic problem by elaborating on what actions exactly are available to solve the problem (e.g. what kind of kitchen appliances you have access to). Deciding on the degree of formality and elaborating on the problem definition is the key element to *modelling* an algorithmic problem. The final degree of formality required to analyse an algorithmic problem depends on what you are studying. The above description of our algorithmic problem is probably sufficient for you to discover how to solve the baking problem and evaluate how much time it would roughly take. If instead we would be interested in programming a robot to do the baking for us, the above definition probably is not formal enough. A robot would need access to a refined description of each ingredient and access to a very precise protocol on how to purchase, transport and combine the ingredients.

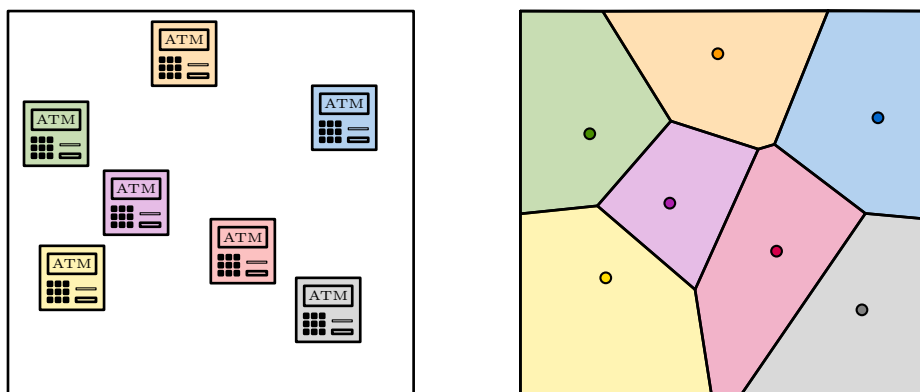
The main tool, or language, we use to further formalise the algorithmic problems that we model is mathematics. Mathematics allows us to compartmentalize an intuitive problem statement into elementary pieces where every piece is as unambiguous as possible. This thesis contains contributions to the modelling of algorithmic problems in general. We review how algorithmic problems and their solutions are mathematically modelled and we discuss how these models relate (and possibly do *not* relate) to the intuitive notion of algorithmic problems and the tools available to us for solving them that we observe in practice.

**Solving algorithmic problems.** Once we have modelled an algorithmic problem we are ready to *provide a solution*. A solution to an algorithmic problem is a handbook that for every input  $x$ , generates a sequence of steps to create the desired output  $y$ . We call this sequence of steps an *algorithm*. More precisely, a *program* is a sequence of instructions. An algorithm is a system that, given an input  $x$ , provides a program that gives the desired output  $y$ . Consider the example where we want to bake a cake. An algorithm to bake a banana cake is called a recipe: given the ingredients  $x$  the recipe specifies the sequence of steps needed to transform  $x$  into the desired cake  $y$ . The recipe could work for a variety of input, for example it could specify the input size of  $x$  (ingredients for a large or small cake) and provide an accordingly adjusted sequence of steps (bake the cake for longer or shorter). When modelling an algorithmic problem, we noted that there exists a varying degree in the formality of the modelling of the problem. Similarly, there exists a varying degree in how we describe an algorithmic solution to our problem. When describing algorithmic solutions, we make a distinction between elementary and high-level operations. Elementary operations are assumed to be well-understood actions that, intuitively, can be executed in a single step. In mathematics, these operations can be elementary mathematical operators such as determining the parity (whether a number is even or odd) or addition. In our example of baking a cake, elementary operations may be physical operations that you can execute without additional thinking such as grabbing an object and twisting your wrist. An algorithm specifies for each input  $x$  the sequence of elementary operations that result in the output  $y$  (e.g. to transform a number  $x = 2$  into the number  $y = 10$ , we can use the addition operator four times and create  $y = 2 + 2 + 2 + 2 + 2$ ). However, an algorithmic description can become rather verbose whenever we restrict ourselves to simply sequencing elementary operations. High-level operations are intuitive actions which are comprised of several elementary operations. In our mathematical example, we created the high-level operation multiplication by sequencing four additions. In our baking example, we can construct a high-level operator of ‘turning the oven on’ by constructing the correct sequence of grabbing an oven handle and turning our wrists. The key here is that once a high-level operator has been established, we and future researchers can refer to it for future algorithmic descriptions. Arguably, establishing algorithms as high-level operators (sometimes called *subroutines*) is one of the more substantial contributions one can make to the literature in theoretical computer science. The majority of this thesis is dedicated to providing solutions to various algorithmic problems. These solutions are either a standalone solution or can serve as subroutines in later work.

**Evaluating algorithmic solutions.** Finally, given a solution (algorithm) for an algorithmic problem we *evaluate* it (reason about the quality). There are various criteria that can be used to evaluate the quality of an algorithmic solution. For example, one could look at the length and readability of the algorithmic description (if you are unable to understand an algorithm, then you will not be able to implement it on a real world computer). The main criterion for an algorithmic solution is its *correctness*: does the algorithm always correctly transform the input  $x$  into an output  $y$ ? Consider the following toy example: we make an algorithm that consists of a single elementary operation: given an input  $x$ , output  $x + x$ . This algorithm may solve the following algorithmic problem: given input  $x$ , output  $y = x + 2$ . Indeed, if the input to this mathematical problem is the number 2, then the output will be  $y = x + x = x + 2 = 4$ . However, for all other inputs  $x$  this algorithm does not provide the correct solution and is thus not always correct. Throughout the thesis, we spend considerable effort towards *proving* that the algorithmic solutions always provide the correct answer. That is, we indisputably show that for all inputs  $x$  that match the problem description, our algorithm provides the correct corresponding output  $y$ . Once we have proved that our algorithmic solution is correct we can begin to evaluate other criteria. In theoretical computer science, there is an emphasis on quantifiable measures that can be objectively evaluated and proven. The main two criteria that we focus on are the *space requirement* and the algorithmic *running time*. We again provide a numerical example: consider the algorithmic problem where the input  $x$  is a collection of  $n$  numbers  $\{p_1, p_2, p_3, p_4, p_5 \dots p_n\} = \{3, 7, 2, 1, 9 \dots 1\}$  and the output  $y$  is the maximal number in  $x$ . There is a straightforward algorithm that can always solve this problem: iteratively inspect all the elements (first  $p_1$ , then  $p_2$  and so forth) and remember the largest number seen so far. So first we inspect  $p_1 = 3$ , and the largest number seen so far is 3. Then  $p_2 = 7$  and the largest number seen so far is 7. Then  $p_3 = 2$ , since 2 is less than 7 the largest number seen so far is still 7 and so forth. When we reach the final number in  $x$ , we output the largest number seen so far as  $y$ . This algorithm can be run whilst storing at most  $n + 1$  values (we have to store all the numbers in  $x$  regardless and a counter  $i$  that specifies that we are inspecting the  $i$ 'th number. We can store the largest number seen so far instead of the first number of the input). In this case, we can prove that this algorithm uses a minimal amount of space to perform the computation. That is, all algorithms that compute the maximal value of  $n$  numbers  $x$  need to store at least  $n + 1$  values to compute the maximum  $y$ . Similarly, this algorithm specifies  $n$  steps before it outputs the maximum  $y$  by inspecting every element in  $x$  once, which is the best one can do. Thus, with respect to space usage and time spent, we show that our solution is *optimal*: there can exist no solution to our algorithmic problem that is better than the one provided here. The majority of this thesis is dedicated to showing that the algorithmic solutions presented in this thesis are *efficient*. That is, the space requirement and algorithmic runtime (time spent on running the algorithm) are better than previous state-of-the-art results. In many cases, our results are optimal: there exists no other algorithmic solution with lower space usage or runtime. For a specific subset of geometric problems, we even revisit previous notions of algorithmic optimality to show a degree of optimality that is stronger (or more general) than prior results seen in the literature.

**Computational geometry** is a subfield of algorithmic research where the algorithmic problems studied have a geometric nature. Geometry is one of the oldest branches of mathematics, as the start of mathematical geometric research is often attributed to the ancient Egyptians or Greeks [184]. The classical geometric paradigm is Euclidean geometry, which revolves around the fundamental notions of points, lines, surfaces, distances and curves. In computational geometry, we study algorithmic problems where the input consist of these fundamental geometric concepts, within the context of a (real world or theoretical) computer. Studying objects such as point sets, disks or surfaces may appear at first as a theoretical exercise. However, many problems studied within computational geometry either originate from, or have applications in, real world dilemmas. Indeed, many real world algorithmic problems can be modelled as algorithmic problems where the input is a collection of geometric objects. We present three real world examples that, when modelled as an algorithmic problem, become a classic algorithmic problem within computational geometry.

In the first example, we are given the locations of a number of Automated Teller Machines (ATMs) across the city and we are interested in marking for every point on the map, the nearest ATM (refer to Figure 1.1). This intuitive problem, when formalised, becomes a problem where both the input and output is geometric. The input is a set of two-dimensional points (each point specifies the coordinates of an ATM on the map) and the goal is to subdivide the map into areas  $A$  corresponding to an ATM  $a$ , such that all points in the area  $A$  have the ATM  $a$  as the nearest ATM. This subdivision of the plane, for a given set of points  $P$ , is called the *Voronoi diagram* of  $P$  (named after Georgy Voronoi [199]).

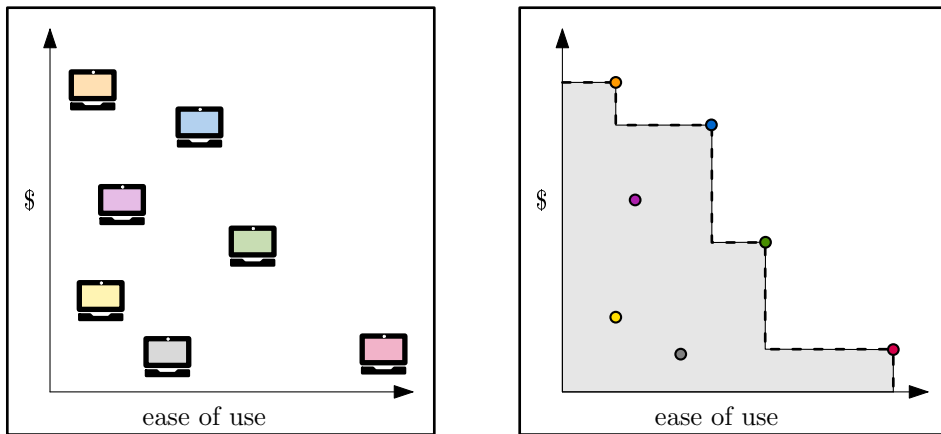


**Figure 1.1** We show a set of ATMs on a top-down view of a map. The location of each ATM can be modelled as a two-dimensional point. We subsequently want to subdivide the plane (top-down map) in a collection of cells, such that all points in a cell have the same nearest ATM. On the right, we see the Voronoi diagram of the point set corresponding to these ATMs.

The second example comes from economics and is illustrated by Figure 1.2. In this example, we want to identify the ‘best’ products on offer. Suppose that we manage an online store where products have associated (user) scores in three categories:

- the ease of use,
- the price/quality ratio, and
- the longevity.

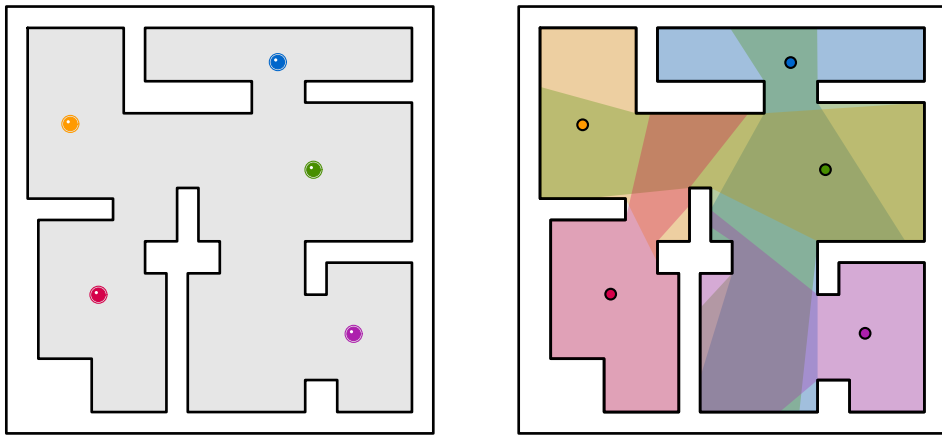
In this example, we are interested in selecting the ‘best’ products to display to our customers. It is not immediately clear what makes a product ‘good’ and which products should be displayed on the front page of our website. However, there is a clear indicator for when a product  $a$  is superior to a product  $b$ : if the product  $a$  is better in all three categories (better rated ease of use, better price/quality ratio and better rated longevity) then  $a$  is a superior choice over  $b$ . It follows that at our front page, we want to display only items  $a$  where there is no item  $b$  that is better than  $a$ . This leads to the following geometric problem where the input is a set of three-dimensional points (each point specifies the three scores associated with a single item in the store). The goal is to find the set of items  $A$  where for every  $a$  in  $A$ , there is no item  $b$  whose coordinates are greater than the corresponding coordinates of  $a$  in every direction. Identifying this staircase-like set of points  $P$  is called computing the Pareto front of  $P$  (the term originates from the notion of Pareto efficiency by Vilfredo Pareto [171]).



**Figure 1.2** We show a set of laptops and plot for each laptop in the place where the  $y$ -axis corresponds to the price/quality ratio and the  $x$ -axis is the rated ease of use (we use only two of the three presented categories because it is easier to draw in two dimensions). Each of these laptops hence corresponds to a point in the plane. We are interested in the set of points  $A$ , for which there exists no point  $b$  such that  $b$  has greater  $x$  and  $y$  coordinate (in that case, the laptop corresponding to  $b$  has a better price/quality ratio *and* is easier to use). The resulting Pareto front is a set of points that follow a staircase-like pattern.

In the third example we want to surveil a building (traditionally, an art gallery) with fisheye cameras (refer to Figure 1.3). Cameras are expensive, hence we would like to place as few as possible. At the same time, we want there to be no *blind spots*: we want that all points in the room are visible by at least one camera. This intuitive problem (when formalised) is of a geometric nature. Indeed, the input of the problem can be modelled as a polygon. A polygon consists of a set of vertices (points) connected by edges (line segments between points), such that the edges together form a closed polygonal chain (a polygonal chain is a sequence of vertices  $p_1, p_2, \dots, p_k$  such that every two consecutive vertices, and  $(p_k, p_1)$ , are connected by an edge). The top-down view of a floor in a building is such a polygon. Indeed, the corners of each room form vertices connected by edges (a wall segment of a room connects two corners).

Fisheye cameras are mounted on the ceiling of the room, hence the location of a camera  $c$  is a point in the simple polygon. Fisheye cameras have  $360^\circ$  view, but they cannot see through walls. Hence, a point  $a$  in the simple polygon (building) is visible from a camera  $c$  if and only if the line segment between  $a$  and  $c$  is not intersected by any edge. These observations allow us to formalise the algorithmic problem as follows: the input is some simple polygon  $x$ . The output is the minimal collection of points (cameras)  $y$ , such that for every point  $a$  in the simple polygon  $x$ , there is a point  $c$  in  $y$  such that the line segment between  $a$  and  $c$  is intersected by no edge in the domain. This is a classical problem within computational geometry that is called the *art gallery problem*.



**Figure 1.3** A top-down view of a building where there are five fisheye cameras mounted on the ceiling. If we model every camera as a point, we can illustrate for every camera the area of the building that it can see. The goal in the art gallery problem is to find a minimal set of points (cameras) that together see the whole floor.

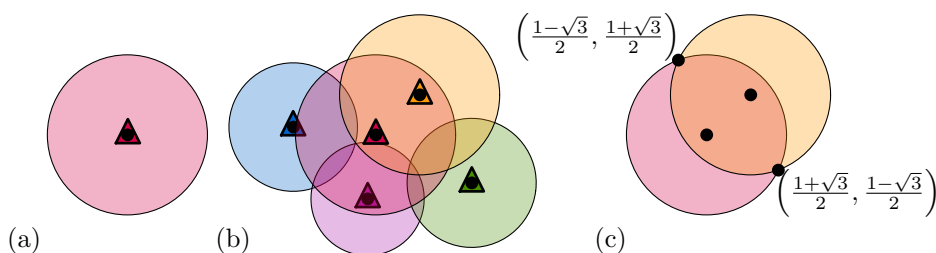
## Challenges tackled in this thesis

There are numerous challenges in computational geometry that are interesting to study. This thesis is dedicated to tackling three challenges, each of which has a dedicated part in this thesis.

**The first challenge** within computational geometry arises from the often-used model of computation. A model of computation is a mathematical model for a computer: an abstraction of the physical concept of a computer that formalises the operations that an algorithm has access to. The model of computation specifies the set of elementary operations available to an algorithm. These operations include actions such as: reading from computer memory, comparing the value of two numbers and arithmetic actions such as addition, multiplication and division. From this well-understood set of elementary operations follows a collection of known high-level operations that make our algorithmic analysis easier. For example: it is well-understood that a computer should be able to compute the maximum of a set of  $n$  numbers in  $n$  steps (see the numerical example provided earlier). If we ever require that our algorithm computes the maximum of some numbers, we can shorten our algorithmic description by referring to this well-known result. Such ‘subroutines’ or ‘abstractions’ are critical for all theoretical algorithmic analysis: without these each algorithmic description and/or publication would be unbearably long. At the same time, it is vital to always remember that our model of computation and all its subroutines and abstractions have the intention to represent the inner workings of a computer as closely as possible (we want all results and algorithms obtained to work in the real world, on a real world computer). Thus, as we introduce high-level operations and abstractions to more swiftly model processes, we must be mindful of preserving the connection between our mathematical model and the real world. For computational geometry, the friction between the demand for an ease of exposition and the need to keep our model representative of computers in the real world presents a unique challenge that we illustrate through the following setting: suppose that we want to write a computer program that provides insight into the efficacy of a cell tower. Specifically, we want to model the area of a specific town where a user has access to at least two cell towers (such areas might illustrate that certain towers are redundant and that costs can be saved by removing the tower). The top-down view of a town can be visualised by a 2D map, and the range of every tower is a disk with some fixed radius (the size of the radius represents the strength of the signal). Refer to Figure 1.4 for an example.

The challenge we encounter in this example is the following: it is well-known that real world computers have a fixed, finite memory (if you open too many tabs on your browser, your computer will eventually crash). However, the area of intersection between two circles cannot be expressed within finite memory. Indeed, consider two disks with radius 2. One centered around the point  $(0, 0)$  and one around the point  $(1, 1)$  (see Figure 1.4(c)). These disks intersect in the points:

$$\left( \frac{1 - \sqrt{3}}{2}, \frac{1 + \sqrt{3}}{2} \right) \text{ and } \left( \frac{1 + \sqrt{3}}{2}, \frac{1 - \sqrt{3}}{2} \right).$$



**Figure 1.4** (a) We can model a cell tower as a disk that represents its reach. (b) The reach of the red cell tower is (almost entirely) covered by the reach of the remaining towers. Hence, the tower may be redundant. (c) Two disks with radius 2, one centered around  $(0, 0)$  and the other around  $(1, 1)$ . Their points of intersection are irrational points that cannot be modelled using a finite sequence of numbers.

The value  $\sqrt{3}$  equals 1.7320508075688772935274463415058724... with many, many more digits (infinitely many to be precise). Thus, we cannot store  $\sqrt{3}$  as a sequence of digits within fixed, finite memory. This presents us with three choices where we can:

1. try to never represent  $\sqrt{3}$  as a sequence of digits and instead refer to the value 'symbolically' as  $\sqrt{3}$ ,
2. approximate the value of  $\sqrt{3}$  by rounding it down to a number (say 1.732), or
3. assume that we *can* store  $\sqrt{3}$  in finite space and perform all operations on  $\sqrt{3}$  as if it were a normal number (we continue our algorithmic analysis with this added assumption).

Geometric settings habitually present us with this same problem: many geometric constructions almost immediately confront us with values that cannot be stored as digits in finite space and in these situations we have to choose one of the three options. However, each of these options has a corresponding drawback:

1. If we choose option 1 it increases the difficulty of our algorithmic analysis and solutions. Whenever we choose to represent our irrational values symbolically, it is no longer straightforward to explain how a computer compares two numbers. Consider the following example, where I ask you: 'is  $\sqrt{17}$  greater than 4?'. Chances are, that you don't immediately know. If you could represent  $\sqrt{17}$  as a sequence of digits ( $\sqrt{17} = 4.123\dots$ ) you could easily deduce the answer. But if you cannot, then you have to come up with a more creative solution (in this example you could check if  $4^2 = 16 < 17$  and conclude that indeed,  $\sqrt{17}$  is greater than 4). The more complicated your algorithmic problem is, the more creative the algorithmic comparisons between values will have to be (try checking if  $\sqrt{2} + \sqrt{3} + \sqrt{5}$  is smaller than  $\sqrt{29}$ ). In this option, the price of staying truthful to a real world computer is that algorithmic results become much harder to obtain and explain.

2. If we choose option 2, where we round down irrational values, we can no longer provide a ‘provably correct’ answer. In our example, if the desired output is the area of overlap between the range of two cell towers then we automatically no longer provide the exact output, as we are immediately forced to round down results so they can be stored in memory. Moreover, it is far from trivial to quantify ‘how wrong’ the eventual solution will be: after one computational step, this rounding down creates an error between our stored values and reality. After the second step (and second rounding) this error may increase and slowly you start to deviate more and more from reality. This problem becomes worse as soon as the desired output is not numerical, but combinatorial. Consider the following example where we have a set of points and a line that represents a ‘cutoff value’ and we want to output all points above this threshold. If we round down a value, it may suddenly be placed below the ‘cutoff line’. In this case, our output is no longer an approximation of reality, but outright wrong instead.
3. The third option is the most appealing for theoretical analysis as it does not demand any additional analysis from us: we simply wrongfully assume that everything works out, and continue as normal. Its ease of use makes it an often chosen option when performing theoretical analysis within a geometric domain. This option however, does come with a significant price: as it explicitly separates our theoretical model (that our algorithmic solutions are described and analysed in) from reality.

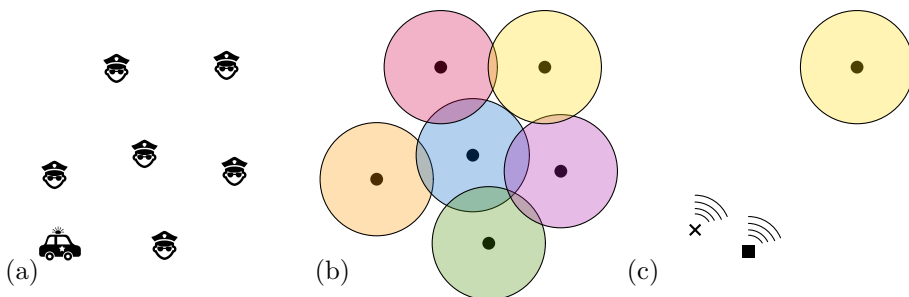
Part II of this thesis is a discussion on how to model geometric information. It is dedicated to analysing the consequences of tackling the challenge of how to represent precise values on a finite-precision machine through the often-taken third option as we ask the following question:

‘If we assume in our theoretical model that we can represent infinite precision values with finite precision (the so-called real RAM assumption) then what is the probability that the outcome of our theoretical algorithm is equal to the outcome of our theoretical algorithm, executed on a real world finite-precision machine?’

We can unpack this statement step by step. Following the third option, we assume a theoretical model of a computer where infinite precision values can be stored and compared to one another, within finite space and time. We then study algorithms on this so-called real RAM model of a computer and ask the question: what happens when we run a program defined on a real RAM on a real world computer that has finite, as opposed to infinite, precision? The answer to this question is well-known: the real RAM program becomes incorrect as the rounding-errors that occur under finite precision may create mistakes. However, we assume that the input  $x$  of our program is not any input but input sampled from some slightly random source. We show that in this case (with high probability) a real RAM computation, when executed with bounded precision, makes no ‘mistakes’. This argument illustrates that the third option, where we assume that a computer can handle arbitrary real values, works decently in practice (where input often is slightly randomised).

**The second challenge** within computational geometry is how to efficiently store and retrieve geometric information. Consider the following example where there are police officers stationed around a city (Figure 1.5). Each officer has a certain *action radius*: an area where for all points in the area they can move towards the location within five minutes. Throughout the day the police may receive several distress calls. When such a distress call occurs, we want to be able to send a police officer to that location (provided that the location is within the action radius of that officer. Otherwise, we might have to send a patrol car instead). We can model a distress call as an algorithmic problem as follows: the input  $x$  is a collection of disks (the action radii of the officers) and a location  $p$  from where the police is called. The output  $y$  is the ID of the police officer that should respond to the call. An algorithm that solves this algorithmic problem might do the following: check the location of every available officer and compute if they are within range. If so, output the officer in range. If no officer is in range, we conclude that we have to send a patrol car.

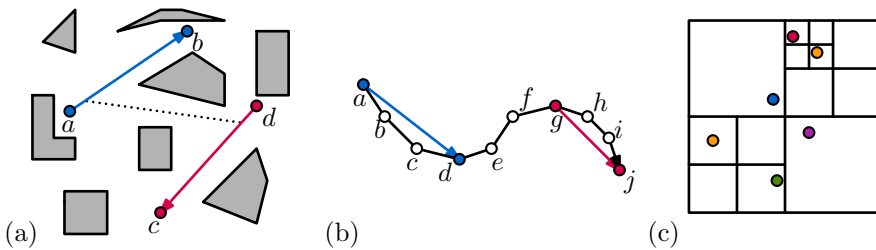
Distress calls happen throughout the day, thus we encounter this algorithmic problem frequently. At every call it is important that our computer can swiftly decide which officer to assign. The above algorithm checks for *every* distress call *every* available officer which can be computationally very costly. It turns out that in this case we can be more efficient: we notice that the set of officers does not change much throughout the day (people tend to be stationed in the same area for several hours a day). So whilst the location of the distress call may vastly change between calls, the set of action radii stays roughly the same. This means that if officers  $A$ ,  $B$  and  $C$  were all very far from the first distress call, and the second distress call is in the same area, then the three officers are *still* very far from the distress call and it would be wasteful to check the location of each officer again.



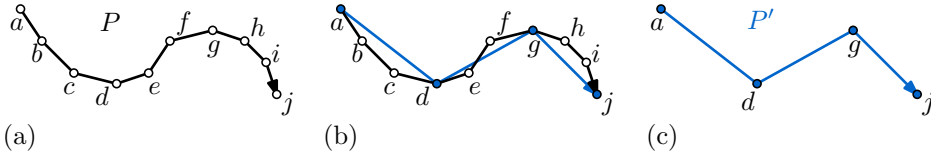
**Figure 1.5** (a) Imagine police stationed around a city and a patrol car. (b) Every agent has an action radius such that whenever there is a distress call from within their action radius, they can arrive quickly. (c) If there is a distress call from the cross, the yellow agent is nowhere near in range. If subsequently there is another distress call from the nearby square then we know that the yellow agent is too far without checking the location of the yellow agent again.

It intuitively follows that instead of checking every officer, every time, we are better off structuring the geometric information in some way. For example, we could split the city into four quadrants and place each officer in a quadrant. If there is a distress call in the northern quadrant, we know that we do not have to check all officers assigned to the southern, eastern or western quadrant which saves (computational) time. In general, given a set of geometric objects, we can store them in what is called a *data structure*: some suitable division of the space into sub-regions such that given a *query* (a distress call from some location) we can efficiently compute the solution to the algorithmic problem without explicitly processing all objects again. Research into efficient data structures is a longstanding tradition within theoretical computer science. Part III of this thesis is dedicated to showing how to structure geometric information such that given some query, we can extract the information efficiently. Specifically, we investigate three different scenarios (refer to Figure 1.6):

- (a) In the first scenario we study visibility testing between moving entities. We consider as input the top-down view of some visibility-blocking environment. We want to structure the environment such that given two entities that each follow a given trajectory, we can determine if the two entities are at any time mutually visible. We call this visibility testing. A trajectory is a representation of the movement of an entity: suppose that an entity walks from a point  $a$  to  $b$  and then to a point  $c$ . We can represent this movement as what is called a polyline trajectory: a line segment connecting  $(a, b)$  and a line segment connecting  $(b, c)$ . We study how to store the environment such that given two single segment trajectories (one entity walks from  $a$  to  $b$ , the other from  $c$  to  $d$ ), we can swiftly determine whether there is a time when the two entities were mutually visible.



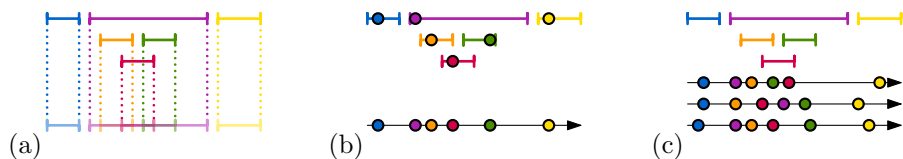
**Figure 1.6** (a) An environment (polygonal domain) shown in black. We store it, such that given two entities that each follow a segment trajectory (blue from  $a$  to  $b$  and red from  $c$  to  $d$ ) we can test if there is a time when they are mutually visible. (b) A complicated trajectory with many edges. We store the trajectory such that given a segment trajectory (blue from  $a$  to  $d$  or red from  $g$  to  $j$ ) we can compute the distance from the segment to a subcurve of  $P$  efficiently. (c) Given a set of points, we can subdivide the plane into squares such that every point has its own square. We show how to maintain this subdivision as we add and remove points.



**Figure 1.7** (a) A trajectory  $P$  that consists of many edges, traveling from  $a$  to  $j$ . (b) We can create shortcuts in  $P$  by traveling from  $a$  directly to  $d$ , from  $d$  to  $g$  and from  $g$  to  $j$ . These shortcuts do not deviate much from the original trajectory. (c) Instead of analysing the complicated trajectory  $P$ , we can study the simplification  $P'$  obtained through these shortcuts instead.

- (b) Next, we study trajectory similarities. This topic has applications in trajectory simplification. Suppose that we have some complicated trajectory  $P$  (a trajectory consisting of many edges). Then it may be that  $P$  is too large and complicated to efficiently study or analyse (refer to Figure 1.7). In such a case we would like to simplify  $P$ : create some trajectory  $P'$  that is much less complex but still resembles  $P$ . When creating this simplified trajectory  $P'$ , it is desirable to be able to swiftly query how much our new trajectory  $P'$  deviates from the original  $P$  (if we deviate too far from  $P$ , we might want to consider another simplification). To this end, we show how to store  $P$  such that given a query segment (*shortcut*) we can swiftly compute the distance between the segment and a part of the trajectory  $P$ .
- (c) Finally, we show how to efficiently store a dynamic set of points for point location. Specifically, the input to our problem is a set of points in the plane (for example, the aforementioned police officers in a city). We show how to store points in a quadtree such that given a query location (a distress call), we can efficiently determine which entities are nearby. We study this data structure problem under the assumption that occasionally, specific points get added or removed from the set (officers may leave to respond to a call). We show how to maintain this data structure subject to these alterations in worst-case constant time (time independent of the tree size), which is the best anyone can do.

**The third challenge** studied in this thesis is how to handle imprecise geometric input. We often assume that we accurately know our algorithmic input (data). In real world applications, this data commonly originates from some measurement of real world objects. For example: trajectory analysis can be performed on data from GPS trackers attached to gulls [103]. In many applications the input measurement is inherently imprecise. GPS trackers have significant measuring error, satellite imaging is subject to distortions and humans make many mistakes when filtering data. Even if the measurement is accurate, we can still have an inaccurate representation of the input. Suppose that we use an accurate GPS to track a moving object. By the time the signal reaches us, the object will have moved slightly. So no matter how accurate the measurement was, at the time of processing the input it has become inaccurate.



**Figure 1.8** (a) A set of imprecise values, represented as intervals on a line. We lift each interval and draw them at different heights so that they are easier to see. (b) The exact values in each interval imply a unique left-to-right order. (c) Depending on what the exact values are, there can be different left-to-right orders. However, the blue point always precedes all others and the orange point precedes the green.

In many of these applications it is possible to make the input more precise at a certain computational cost. For example, when working with GPS signals as your input data, you could choose to ping the location of an entity again if you want an update of its current location. However, if this computational cost is high then we would rather avoid refining every single entry. This is the inspiration for the following question:

‘Given an imprecise input  $x$  and a desired output  $y$ , which of the input entries need to be refined further to compute  $y$ ?’

Part IV of this thesis is dedicated to studying this question. We assume that for every imprecise value, we know a range (or region) that must contain the possible corresponding exact value. In Figure 1.8 we show an example with six imprecise values, each in a specified range:  $x = \{[0, 3], [5, 8], [7, 10], [9, 12], [4, 15], [16, 20]\}$ . Whenever we want to compute some output  $y$  on the imprecise input  $x$ , it may not be required to retrieve the exact location of every imprecise point. Indeed, suppose that we want to know the left-to-right order of our example. The first value may lie anywhere in the interval  $[0, 3]$  whilst all other points are greater than 3. Thus, the first value is always first in the left-to-right order of  $x$ : a fact which we can conclude without refining the data any further.

In Part IV of this thesis we formalise the above intuition that not every value needs to be further refined. We combine insights from Parts II and III to show that we can store the imprecise values in a data structure, such that given the opportunity to refine the values further we can do so efficiently. Specifically, we assume that for every imprecise value we can obtain its exact location at some fixed cost (e.g. the cost of performing a new GPS measurement on that entity). We show that our data structure can be used to identify which imprecise values need to be refined. Moreover, when refining one value we can efficiently deduce which remaining values need to be further defined. We show how to sort a set of imprecise geometric values and we prove that there can be no sorting scheme which is more efficient than the one we present here. We extend our results from one-dimensional sorting to computing the Pareto front (maximal set) of a set of two-dimensional imprecise points.

## Chapter Two

# Modelling, structuring and capturing

In the introduction we presented an intuitive description of what theoretical computer science, algorithms and algorithms theory is. We presented a high-level summary of the three challenges within computational geometry that the parts of this thesis investigate. In this chapter we elaborate on the background and definition of each challenge. Every section in this chapter is dedicated to the corresponding part of this thesis and provides a low-level overview of the concepts studied in that part. We make steps towards formalising the algorithmic problem studied and provide a glimpse of the use or implications of the results. This chapter is dedicated to presenting a low-level overview of the topic of each remaining chapter: for a full problem description, combined with references to related work, we refer to the introduction of each respective chapter. At the end of each section we specify which publications correspond to the content of the part. In the last section we briefly describe research that was performed during the PhD process that is not presented in this thesis.

## 2.1 Part II: modelling geometric information

Part II of this thesis is dedicated to studying models of computation. In algorithm theory, we study an algorithmic problem as a mathematical function that maps any desired input  $x$  to its corresponding output  $y$ . A model of computation is a mathematical model which describes how the output of such a function can be computed. The most widely known model of computation is indisputably the Turing machine by Alan Turing [192]. This model of computation is lauded for its mathematical and conceptual simplicity. The intuition behind its definition is that a computer *reads* a program. Whenever we read, we read letter-by-letter. At all times our mind is in some current state. After every letter the state gets updated with the new information and, depending on our current state, we may choose to skip ahead to the next word, go back to previous letters or simply proceed to the next letter. The Turing machine is a formalisation of this intuitive description. The Turing machine is a model of computation that assumes that a computer has access to a finite set of *states* and that all times it has selected one of these states to the ‘current’ state. A program on a specific input is an infinitely long *tape* subdivided into cells. The computer has a pointer to a single cell on the tape which is the ‘current’ cell. This pointer is called the *head* of the machine. Each cell can contain at most a single value called a *symbol* (intuitively, the symbol is a letter). The set of symbols is also assumed to be a finite set. A computer reads the symbol and evaluates it together with its current state. The outcome of this evaluation can be any combination of three options:

- write a new symbol in the current cell,
- move the head to a new location on the tape,
- adopt another state as the ‘current’ state.

This is the intuitive definition of how a Turing machine works. Jeff Erickson states that: ‘the precise definition of a Turing machine is surprisingly unimportant’ [82]. This statement references the Church-Turing thesis which states that the various precise definitions of Turing machines are all equivalent. For our intents and purposes the statement is also true when taken out of context as it is difficult to perform the algorithmic analysis of this thesis on a Turing machine for two reasons:

1. The first reason is that we study algorithmic efficiency: for a given input of  $x$  of  $n$  elements, we want to express the number of steps required to reach the output  $y$ . The ‘bare bones’ definition of a Turing machine is difficult for this kind of algorithmic analysis as singular steps in an algorithm must often be decomposed into several smaller steps on a Turing machine.
2. The second reason was alluded to in the introduction. In computational geometry, we assume that our computer can represent arbitrary, infinite-precision real values. Since there are infinitely many of these and each of these can be part of the algorithmic input or output, there cannot be a finite number of states and symbols (which a Turing machine assumes). Indeed, any real number may be part of the input and therefore part of the read states or symbols.

To remedy the first problem, we use a Random Access Machine (RAM) as the model of computation instead. At its core, a Random Access Machine has access to *registers* that store the current state, and a *central processing unit* that can perform a fixed set of instructions on the registers. A register is a memory slot that can store a *single number* (see the next paragraph). The registers together form the memory of the computer that records the current state (e.g. a set of  $n$  two-dimensional points  $P$  can be represented by using  $2n$  registers: each of which stores a coordinate of a point in  $P$ ). The CPU has some fixed, finite set of operations called *instructions* that may manipulate the registers. This includes reading a value from a register, writing a value into a register, comparing two values, arithmetic operations and referencing the location of another instruction. Given a RAM, a *program* is a list of consecutive instructions. A program on a RAM iteratively performs instructions until it reaches the HALT instruction. A HALT instruction terminates the program: the output  $y$  is then part of the current content stored in the registers. The number of instructions performed before the program halts is the *running time* and the number of registers required during execution is the *space requirement*. We aim to analyse geometric algorithms by analysing the running time and their space requirement. More precisely, recall that an algorithm for any input  $x$  provides a program that given  $x$  outputs the corresponding  $y$ . We study algorithms for algorithmic problems and show (for a fixed algorithm), assuming that the input  $x$  can be represented using  $r$  registers, an upper bound on the running time and the space requirement of the corresponding program (this upper bound is parametrized by  $r$ ).

**Word RAM and real RAM.** The registers in a RAM each store a single number. The canonical model of computation within computer science is the *word RAM*, [97, 98, 112, 131]. The word RAM is parametrized by an integer called the word size  $w$ . Every value in each register is a number that can be represented by at most  $w$  bits. The word RAM assumes that the numbers in two registers can be compared in a single CPU instruction. The word RAM models two crucial aspects of real-life computing: first, computers must store values up to finite precision. Second, operations on more values or higher precision numbers take more time (for example, we must represent a high precision number using several registers. This implies that operations on this high precision number automatically require more than a single CPU instruction). Besides the word RAM, there exists the real RAM. In the introduction we showed an example (Figure 1.4) of a geometric construction where the numbers that resulted from our geometric computations cannot be expressed as a finite set of bits (e.g. the number  $\sqrt{3}$ ). Hence, these values cannot be stored in the word RAM (we would need an infinite number of registers to store all the bits). We explained how this is problematic for our algorithmic analysis and that it is common to assume, within computational geometry, that values can be stored up to infinite precision. The real RAM formalises this intuition, as it assumes that every register can store a single real-valued number up to infinite precision. The model of computation assumes that the numbers in two registers can be compared in a single CPU instruction, regardless of their complexity.

**Word RAM vs real RAM.** The word RAM, on one hand, accurately models a real-world computer with its assumption that every register can store a number only up to bounded precision. The real RAM, however, enables us to do perform algorithmic analysis that would otherwise be impossible (the algorithm runtime would at all times be dominated by the time that a comparison between two real-valued numbers may take). An interesting question to investigate is how much, in a practical setting, the real RAM deviates from the word RAM: can we somehow parametrize the difference between the word and real RAM?

Part II of this thesis is dedicated to this question. In Chapter 3 we propose a formal definition of the real RAM that supports both the real-valued operations that are often required for algorithmic analysis within computational geometry and the bit and memory manipulation of the word RAM that we observe in real world computers. This definition allows us to ‘translate’ real RAM programs to word RAM programs. First, for a given real-valued input  $a$ , we construct a corresponding bounded-precision input  $a'$ . Second, given a real RAM program with the input  $a$ , we construct an ‘equivalent’ program on the word RAM with the input  $a'$ .

In Chapter 4 we apply the real RAM definition and the mapping between real RAM and word RAM programs. We say that the real RAM program is *correct* under bounded precision, if the executions of both equivalent programs reach the same HALT instruction (if a real RAM program can be executed with low bit precision, the program is called *robust* [113]). The intuition behind this definition is that whenever this is the case, we can run our real RAM programs with bounded precision and still reach the same conclusion. We then investigate the following question: ‘what is the likelihood, given a real RAM algorithm with random input  $a$ , that the program corresponding to  $a$  is correct under bounded precision?’. Specifically, we are given a real RAM algorithm and input and we randomly perturb every number in the input. That is, every number  $p$  in the input is replaced by another number  $p + x$  for some uniformly random  $x \in [-\delta/2, \delta/2]$  (for some sufficiently small  $\delta$ ). This perturbed input is our algorithmic input  $a$  and we study the probability that the program corresponding to  $a$  is correct under bounded precision. We show for a wide category of algorithms that, with high probability, the program corresponding to the perturbed input  $a$  is correct under bounded precision. Real world data almost always has some degree of randomness to it (GPS data has some inherent random error, humans make random errors when sampling information, precision was randomly lost during information compression, etc.). Thus, our analysis illustrates that the ‘standard’ real RAM model of computation that is used within computational geometry, for a wide category of algorithms, is usable in practice as on this real-world input it has a high probability that the corresponding program is correct under bounded precision.

The content of Chapters 3 and 4 is based on joint work with J. Erickson and T. Miltzow that appeared at the Symposium on Foundations of Computer Science (FOCS 2020).

## 2.2 Part III: structuring geometric information

In theoretical algorithmic analysis we analyse real RAM data structures and their query algorithms, where our main evaluation criteria for some algorithm  $A$  are:

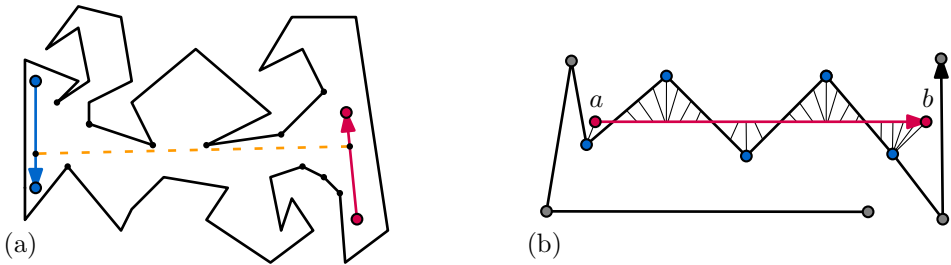
- correctness (does the algorithm  $A$  produce for every input  $x$ , a corresponding program that outputs the desired output  $y$ ?),
- the running time of  $A$  (given an input  $x$  that can be stored in  $r$  registers, can we upper bound the (maximum) number of instructions before the HALT instruction is reached?), and
- the space requirement of  $A$  (given an input  $x$  that can be stored in  $r$  registers, can we upper bound the maximum number of non-empty registers during the program execution?).

In the introduction we described an algorithmic problem in terms of an input  $x$  and a required output  $y$ . In other words: in an algorithmic problem we assume that we start with nothing, then receive the input  $x$  with a single question and compute the corresponding output  $y$  which is the answer to that question. However, many practical problems do not follow this ‘one shot’ format. Indeed, in the introduction we presented an example where we have a set of police officers stationed around a city and distress calls throughout the day (see Figure 1.5). In this example, we have continuously (almost) the same input and the same type of question. In such cases, it is interesting to see if we can preprocess (part) of our algorithmic input such that when the entire algorithmic input is given, we can compute the corresponding output more efficiently. We refer to such problems as *data structure problems* as opposed to algorithmic problems, and this paradigm can be formalised as follows: ‘Given is an input  $x$ , and a set or class of algorithmic problems  $Q$  which we call *queries*. Can we preprocess  $x$  (store some information in the registers of our RAM), such that given a query from  $Q$  where the input is  $x$  (and possibly some additional information), we can efficiently output the corresponding  $y$ ?’ Part III is dedicated to presenting solutions for data structure problems within computational geometry.

In Chapter 5 we study how to preprocess a polygonal domain  $P$  with  $n$  vertices to support sublinear visibility queries between two segment trajectories in  $P$ . This problem originates from the following real world dilemma: researchers in biology studied the movement of monkeys through GPS tracking [41]. They attached GPS trackers to monkeys to study their traversal through their habitat. From time to time, the researchers noticed that a group of monkeys would suddenly deviate from their likely trajectory. The conjecture is, that the deviating monkeys had spotted another monkey and chose to avoid possible conflicts. To test this conjecture, we encounter a data structure problem. The input  $x$  of the data structure problem is the sight-blocking terrain in the habitat of the monkeys. A single query in  $Q$  is an algorithmic problem where the input is  $x$  and the trajectories of two monkeys. The query asks if there is a time whenever the two monkeys were mutually visible. If this time corresponds to an unexpected deviation, the conjecture is supported.

Specifically, in Chapter 5, we make additional assumptions when we formalise the data structure problem. We model the top-down view of the habitat as a two-dimensional polygonal domain  $x$  (refer to Figure 2.1(a)). For simplicity, we assume that the trajectory of each monkey is a directed line segment that the monkey traverses at constant speed. Thus, every query is an algorithmic problem where the input is  $x$  plus two directed line segments. We output a simple *yes/no* depending on whether, during the trajectory traversal, there is a time when the monkeys are mutually visible. We show, in a variety of settings, how to preprocess a polygonal domain of  $n$  vertices (that can be stored using fewer than  $r = 3n$  registers) such that these queries can be answered in time sublinear in  $n$  (and  $r$ ). This chapter is based on joint work with P. Eades, M. Löffler and F. Staals that has appeared at the Scandinavian Symposium and Workshops on Algorithm Theory (SWAT 2020).

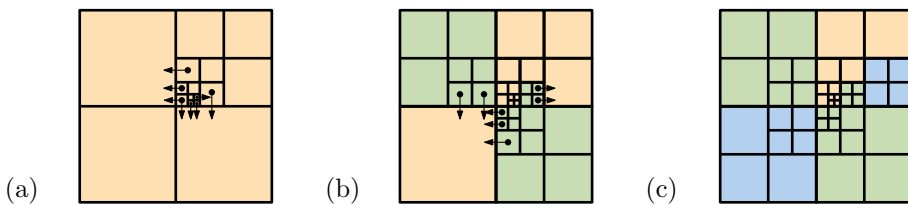
In Chapter 6 we continue studying algorithmic analysis of trajectory data. Trajectory data records the position of some moving entity. Consider for example an entity tracked by GPS. A GPS tracker regularly sends its current coordinates to some recording server. The output of this process is a sequence of points where we know that the entity traversed the points in order. We subsequently assume that the entity travels between consecutive points in a straight line. If we want to analyse the movement of the entity, the algorithmic input then becomes a polyline trajectory (a sequence of points, connected by line segments). The larger the algorithmic input is (the more points there are in the trajectory), the longer subsequent data analysis may take. To make the data analysis go faster, it may be interesting to first simplify the trajectory somewhat (replace the trajectory  $P$  with a trajectory  $P'$  that contains fewer vertices). When simplifying a trajectory we want to replace the trajectory  $P$  with a trajectory  $P'$  that is ‘similar’ to  $P$ . In Chapter 6 we use the well-studied Fréchet distance as a measure of similarity. When constructing a trajectory simplification  $P'$ , it is useful to be able to efficiently compute the similarity between (candidate) segments of  $P'$  and (sub)trajectories of  $P$ .



**Figure 2.1** (a) Given a simple polygon and two segment trajectories (red and blue), we want to determine whether there is a time when both entities are mutually visible (the yellow line). (b) Given a subtrajectory (blue) and a segment trajectory (red), we want to efficiently compute their Fréchet distance.

Specifically, in Chapter 6, we study the following data structure problem (Figure 2.1(b)). The input  $x$  is a planar polyline trajectory of  $n$  vertices (that can be stored in  $r = 2n$  registers). A query specifies a directed line segment  $\overline{ab}$  and two points  $s$  and  $t$  of the trajectory (these points are not necessarily vertices of the trajectory). The desired output is the Fréchet distance between  $\overline{ab}$  and the subtrajectory of  $P$  bounded by  $s$  and  $t$ . We show how to preprocess a trajectory, such that these queries can be answered in time sublogarithmic in  $n$  (and therefore,  $r$ ). In addition, we show how this data structure can be used to speed up state-of-the-art trajectory simplification algorithms. This chapter is based on joint work with M. Buchin, T. Ophelders, L. Schlipf, R. Silveira and F. Staals that is currently under submission.

In Chapter 7 we study a hierarchical space decomposition called a quadtree. A quadtree is obtained through the following scheme: some planar square  $R$  corresponds to the root of the tree. The root  $R$  may be split into four equal-sized squares that partition  $R$  (these are the children of  $R$ ). Subsequently, each of the children may be split and so forth. The multi-dimensional generalisation of a quadtree is a structure over a  $d$ -dimensional hypercube where every cube in the tree may be split into  $2^d$  equal-sized cubes. A quadtree is a multi-purpose data structure that can store various types of geometric data. We study the quadtree as a standalone dynamic data structure, subject to the split operation and its inverse (the merge operation). We study how to preserve a property called *smoothness* in a quadtree. For a quadtree  $T$  a leaf cube is a cube that corresponds to a leaf in the tree  $T$ . A quadtree is smooth whenever for all leaf cubes  $C$ , all leaf cubes of  $T$  that are geometrically adjacent to  $C$  (partially share a facet with  $C$ ) have comparable diameter. Quadrees do not guarantee smoothness; however, for any quadtree  $T$  it is possible to perform additional splits to create an extended tree  $f(T)$  that is smooth (refer to Figure 2.2). We show how to dynamically maintain such an extended quadtree  $f(T)$  subject to splits and merges in  $T$ , in worst case constant time (if the dimension  $d$  is constant). In addition, we show how to maintain a search structure on  $T$  and  $f(T)$  in worst case constant update time. This is joint work with E. Arseneva and M. Löffler that appeared at the International Symposium on Computational Geometry (SoCG 2018).



**Figure 2.2** (a) Given an arbitrary quadtree  $T$ , we mark the non-smooth leaf squares  $C$  that are adjacent to leaf squares  $C'$  with  $|C'| > 2|C|$ . (b) We may add squares to  $T$  to make the quadtree more smooth, but added squares are not automatically smooth themselves. (c) We can make the quadtree  $T$  smooth by adding even more squares. In a dynamic setting, the challenge is to avoid a ‘cascading’ of adding squares.

## 2.3 Part IV: capturing geometric information

In Part IV of this thesis we study algorithmic problems with input that is subject to geometric imprecision. In the introduction we explained how algorithmic input that is obtained from real world data is often imprecise. This imprecision can come from various sources, including errors inherent to the measuring of the real world data, a discrepancy between the time of measuring and the time of computation or the imprecision may be inherent to the input domain (this for example applies to data obtained from a random simulation). In Part IV of this thesis we focus on imprecise input that may be further refined. Specifically, we assume the following:

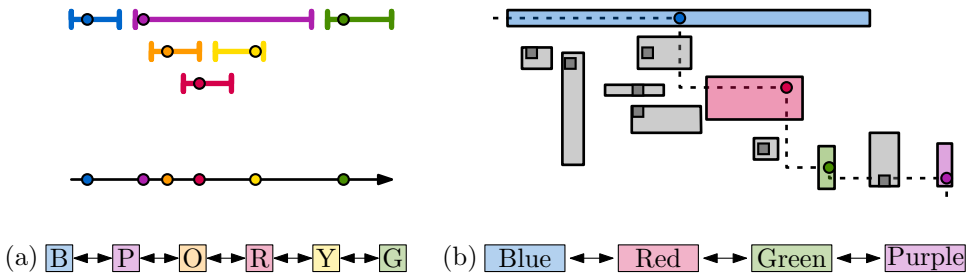
- for every input  $x$ , we know which values are imprecise,
- for every imprecise value we know an upper bound on the imprecision, and
- for every imprecise value we can retrieve a fully precise representation.

These assumptions are formalised by the preprocessing model by Held and Mitchell [117]. Here, the algorithmic input is some set of constant-complexity geometric regions  $\mathcal{R} = (R_1, R_2, \dots, R_n)$ . There is some unknown ‘true’ point set  $P$  such that for all regions  $R_i$ , there is a unique point  $p_i \in P$  with  $p_i \in R_i$ . The region  $R_i$  represents the imprecise value corresponding to the unknown point  $p_i$ . For every region  $R_i$  we may retrieve  $p_i$  for some known fixed cost  $C$  (typically, this cost  $C$  is constant but it may be any number of RAM instructions). The goal is to preprocess  $\mathcal{R}$  to subsequently construct a data structure on  $P$  faster than would be possible without preprocessing.

In Chapter 8 we further formalise the preprocessing model. We reason about what the desired output of the preprocessing may be and we propose a definition of the preprocessing model that deviates from the classical model by Held and Mitchell. We propose what we call *using indirect representations* where the output in the preprocessing model is not a structure on the point set  $P$  but rather a structure on  $\mathcal{R}$  which indirectly represents the desired outcome on  $P$ . Formally, we assume that the desired algorithmic output  $y$  is a pointer structure on  $P$ . In the preprocessing model with indirect representations, the output is a pointer structure on  $\mathcal{R}$  such that if we replace every region  $R_i$  with the point  $p_i$ , we obtain  $y$  (Figure 2.3). We provide several arguments to support this new definition of output (e.g. our new definition of output allows for sublinear algorithmic results). The remainder of this chapter is dedicated to a discussion about algorithmic optimality in this new framework. We provide a translation of the well-studied concepts of worst case and instance optimality for the preprocessing model with indirect representations. We subsequently show that for a large class of algorithmic problems the worst case lower bound is linear in the input size and the instance lower bound is provably unmatchable. Finally, we provide a new definition of algorithmic lower bound which we call the uncertainty-region lower bound. This new definition of lower bound has a granularity that falls somewhere in between worst case and instance optimality. The content of Chapter 8 is based on joint work with I. Kostityna, M. Löffler and B. Speckmann (this joint work is part of the publications and submissions corresponding to Chapters 9 and 10).

In Chapter 9 we study sorting a set of imprecise values (one-dimensional points) in the preprocessing model using indirect representations. Here, the input is a set  $\mathcal{R}$  of  $n$  uncertainty intervals with a corresponding ‘true’ point set  $P$ . We show how to preprocess  $\mathcal{R}$ , such that given access to  $P$  we can construct an indirect representation of the sorted order of  $P$  in time that matches the uncertainty-region lower bound. This indirect representation is a linked list on the regions in  $\mathcal{R}$  such that if we replace every region  $R_i$  with the corresponding point  $p_i$ , we obtain a linked list that matches the sorted order of  $P$ . We show that this problem, when formalised in the preprocessing model with indirect representation, is the geometric variant of the well-studied problem of sorting under partial information. Our results improve the state-of-the-art results on sorting under partial information in a less general geometric setting. This is based on joint work with I. Kostityna, M. Löffler and B. Speckmann that has appeared at the International Symposium on Computational Geometry (SoCG 2019).

Chapter 10 is dedicated to studying the Pareto front of a set of imprecise values (two-dimensional points) in the preprocessing model using indirect representations. In the introduction (Figure 1.2) we explained that for two points  $a$  and  $b$ , the point  $a$  dominates  $b$  if the  $x$  and  $y$  coordinates of  $a$  are both greater than the respective coordinates of  $b$ . For a point set  $P$ , the Pareto front of  $P$  is the subset of points  $p \in P$  that are not dominated by any other point in  $P$  (connected through a staircase). In Chapter 10 we show that if  $\mathcal{R}$  is a set of two-dimensional rectangles that are pairwise disjoint, we can preprocess  $\mathcal{R}$  such that we can compute an indirect representation of the Pareto front of  $P$  in uncertainty-region optimal time. This result is the first higher-dimensional result in the preprocessing model that is not trivially optimal. The content of this chapter based on joint work with I. Kostityna, M. Löffler and B. Speckmann that is to appear at the Symposium on Discrete Algorithms (SODA 2022).



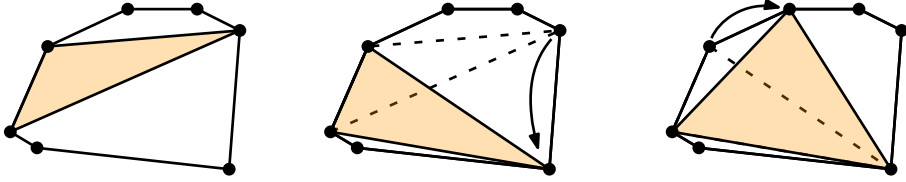
**Figure 2.3** (a) For a set of imprecise one-dimensional points, their indirect sorted order is a linked list on the regions such that if we replace the regions with their points, we obtain the left-to-right order of the points. (b) For a set of imprecise two-dimensional points, their indirect Pareto front is a linked list on a subset of the regions such that if we replace the regions with their points, we obtain the top-left to bottom-right traversal of the Pareto front.

## 2.4 Additional research performed during the PhD

In this section, we provide a short description of additional research performed during the PhD process that is not discussed further in this thesis.

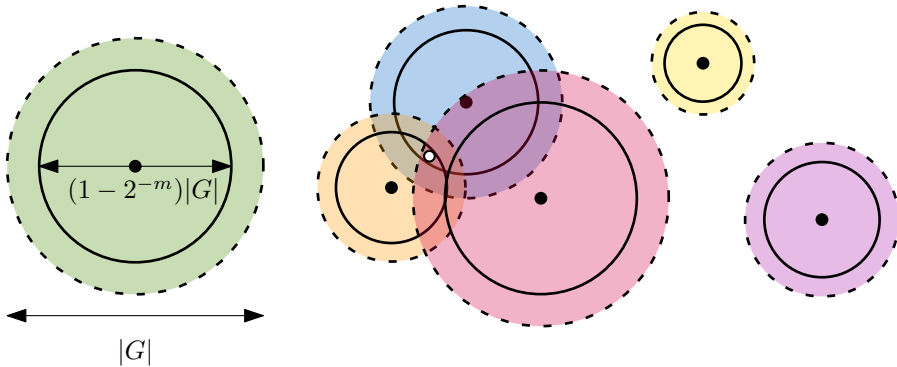
**Maximum-area triangle in a convex polygon, revisited.** We study the well-known and often cited Dobkin-Snyder algorithm for computing the maximum-area triangle in a convex polygon [72]. This algorithm supposedly finds the maximum-area triangle that is contained within a convex polygon in linear time through a simple scheme (Figure 2.4): given a convex polygon  $P$  and a triangle  $T$  contained in  $P$ , the authors suggest to see if there is a local improvement to the area of  $T$  (that is, if any of the vertices of  $T$  can be moved along the convex polygon to increase the area of  $T$ ). The authors claim that after at most  $O(n)$  local improvements, one can find the largest area contained in  $P$ . However, we show a polygon  $P$  and a starting triangle such that the algorithm encounters a triangle  $T$  where there exists no local improvement, even though  $T$  is not the maximum-area triangle. This paper is joint work with V. Keikha, M. Löffler, A. Mohades and J. Urhausen and has appeared at Information Processing Letters volume 161 in 2018. This discovery initiated renewed interest in this algorithmic problem [59, 121, 126, 139].

**Topological stability of kinetic  $k$ -centers.** The  $k$ -center problem or facility location problem asks for a set of  $k$  disks that cover a given set  $P$  of  $n$  points, such that the radii of the disks are as small as possible (by some measure of size). The problem can be interpreted as placing a set of  $k$  facilities (e.g. stores) such that the distance from every point (e.g. client) to the closest facility is minimized. Specifically, for a set  $P$  of  $n$  points and a fixed integer  $k$ , we denote by  $C_k$  any set of  $k$  disks such that every point in  $P$  is contained in a disk in  $P$ . We denote by  $|C_k|$  the size of  $C_k$ : which may be the size of the smallest radius in  $C_k$  or the sum of all disk radii. We study a kinetic variant of this problem where the input is a set  $P$  where every point in  $P$  moves at some fixed speed in some fixed direction and we want to maintain the  $k$ -center during the movement of  $P$ . We are interested in realising the transition from a minimal  $k$ -center  $C_k$  to the next minimal  $k$ -center  $C'_k$  in a topologically stable manner (a topologically stable transition obtains  $C_k$  from  $C'_k$  in a series of operations called *flips* which we denote by  $f(C_k, C'_k)$ ). For various measures for the size of a  $k$ -center  $C_k$  (and various notions of disks) we show examples of kinetic points in which there exists a time where we need to transition from the minimal  $k$ -center  $C_k$  to the next  $k$ -center  $C'_k$ , with  $C''_k \in f(C_k, C'_k)$  such that  $|C''_k| \geq c \cdot |C_k| = |C'_k|$  for some constant  $c$ . Next, we show an  $O(n^{6k+1})$  time algorithm that for any kinetic set of points  $P$  can maintain the kinetic  $k$ -center. We show an algorithm where for every transition from a  $k$ -center  $C_k$  to a  $k$ -center  $C'_k$ , the radii encountered in the sequence of flips matches our corresponding lower bounds. This paper is joint work with M. van Kreveld, W. Meulemans, K. Verbeek and J. Wulms and has appeared in Theoretical Computer Science volume 866 in 2021.



**Figure 2.4** The algorithm starts with a convex polygon and an inscribed triangle. For each of the triangle vertices, the Dobkin-Snyder algorithm considers shifting the vertex along the polygon boundary to increase the triangle area.

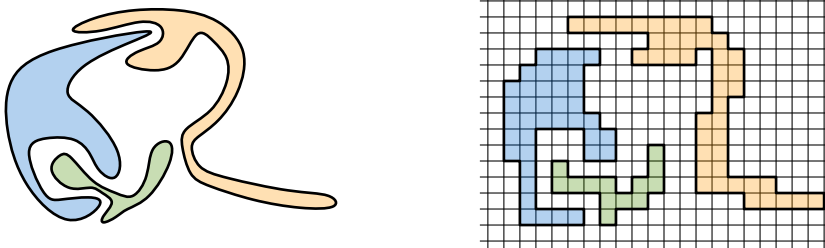
**Dynamic stabbing queries with sub-logarithmic local replacements.** We study a dynamic set of planar disks, subject to approximate stabbing queries. A point  $p \in \mathbb{R}^2$  stabs a disk  $R$  whenever it is contained in its interior. Given a fixed integer  $m$  and a disk  $R$ , we denote by  $I_m(R)$  the inner disk which is a disk centered at  $R$  with diameter  $(1 - 2^{-m})|R|$  (Figure 2.5). A  $(1 - 2^{-m})$ -approximate stabbing query on a set of disks  $\mathcal{R}$  receives a point  $p$  and returns all disks  $R \in \mathcal{R}$  where  $p \in I_m(R)$  and may return additional disks  $R' \in \mathcal{R}$  with  $p \in R'$ . We show how to dynamically store a set of disks  $\mathcal{R}$  in a quadtree such that we can efficiently perform such approximate stabbing queries. Specifically, we show how to store a set of  $n$  disks  $\mathcal{R}$  (where at most a constant number of disks overlap in a single point) using linear space, such that we support  $(1 - 2^{-m})$ -approximate stabbing queries in  $O(\log n + m)$  time. In addition, we support what are called local replacements in  $O(\frac{\log n}{\log \log n} + m)$  time (during a local replacement, we replace a disk  $R$  with a disk  $R'$  which is *similar* to  $R$ ). This is joint work with E. Arseneva and M. Löffler and has appeared at the European Workshop on Computational Geometry (EUROCG 2018).



**Figure 2.5** A set of disks where the inner disk is an unbroken curve and the outer disk is dashed. A stabbing query with the white point must return the blue and orange disks, it may return the red disk and it may not return any other disk.

**Cook-Levin for the real RAM.** The complexity class NP is defined using the SAT problem. In the SAT problem, the input is a logical formula of Boolean values and the output is an assignment of Booleans such that the formula is true. A problem is NP hard whenever it is at least as difficult as SAT and lies in NP whenever it can be written as a SAT formula. The famous Cook-Levin theorem implies that an algorithmic problem has a polynomial time word RAM verification algorithm (a polynomial time algorithm that verifies for a given input  $x$  with a solution  $y$ , whether  $y$  is indeed the solution) if and only if it lies in NP. This theorem makes showing that algorithmic problems lie in NP significantly easier. The complexity class  $\exists\mathbb{R}$  is defined using the ETR problem. In the ETR problem, the input is a logical and arithmetic formula of real values and the output is an assignment of reals such that the formula is true. A problem is  $\exists\mathbb{R}$  hard whenever it is at least as difficult as ETR and lies in  $\exists\mathbb{R}$  whenever it can be written as a ETR formula. We show a real RAM analogue to the Cook-Levin theorem as we prove that an algorithmic problem has a polynomial time real RAM verification algorithm if and only if it lies in  $\exists\mathbb{R}$ . This is joint work with T. Miltzow and J. Erickson and is part of the same publications that the content of Chapters 3 and 4 is based on.

**Mapping multiple regions to the grid with bounded hausdorff distance.** We study a problem motivated by digital geometry: given a set of disjoint regions  $\mathcal{R}$  and some fixed grid width  $w$ , we want to assign to every region a unique set of connected cells on a grid of width  $w$  such that the resulting image  $f(\mathcal{R})$  is similar to  $\mathcal{R}$  (refer to Figure 2.6). The similarity measure that we use is the Hausdorff distance. For various classes of regions  $\mathcal{R}$ , we show lower bounds for the Hausdorff distance between  $\mathcal{R}$  and any image  $f(\mathcal{R})$ . We show, for various classes of regions  $\mathcal{R}$ , how to construct an image that matches these lower bounds. This is joint work with M. van de Kerkhof, M. van Kreveld, M. Löffler, F. Staals, J. Urhausen and J. Vermeulen and has appeared at the Workshop on Algorithms and Data Structures (WADS 2021).



**Figure 2.6** The input is a set of three disjoint regions. We assign to each region a connected set of cells in a grid (pixels). The goal is to make an assignment that minimizes the Hausdorff distance between the regions  $\mathcal{R}$  and the image  $f(\mathcal{R})$ .

PART II

# Real RAM computations



## Chapter Three

# Defining the real RAM

In this chapter we give an overview of the often-used model of computation of theoretical computer science: the Random Access Machine (RAM). We revisit previous works that define the two common types of RAM: word RAM and real RAM, to formulate a precise enough real RAM definition that supports our results, and computations generally done in computational geometry. This formalization serves two purposes: first, it provides the theoretical framework in which all computations in the remainder of this thesis are done. Second, it provides a translation between real and word RAM computations, which in turn allows us to reason about the *bit-precision* required to do a computation on an actual (practical) computer.

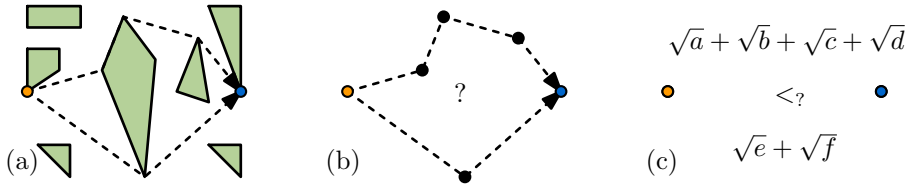
On an intuitive level, the RAM is a mathematical model of a computer which emulates how a computer can access and manipulate data. Within computational geometry, algorithms are often analyzed within the real RAM [95, 140, 178] (or the later Blum-Shub-Smale machine [29]) where values with infinite precision can be stored and compared in constant space and time. By allowing these infinite precision computations, it becomes possible to verify certain geometric primitives in constant time, which simplifies the analysis of geometric algorithms. Mairson and Stolfi [153] point out that ‘without this assumption it is virtually impossible to prove the correctness of any geometric algorithm’. However, defining the real RAM is not as simple as immediately allowing all word RAM instructions to take constant time when their input is a real value. Schönhage [181], for example, observes the following: suppose that you are able to perform constant-time arithmetic on arbitrary real numbers *and* that you are able to convert a real number to its nearest integer in constant time (this operation is often referred to as the *floor* operation), then the complexity class PSPACE is contained in  $P$ . This statement is widely believed to be false [14, Ch. 4].

### 3.1 Defining the real RAM

The Random Access Machine (RAM) is a model of computation for the standard computer architecture. At its core the RAM has a number of *registers* and a *central processing unit* (CPU), which can perform operations (instructions) on register values like reading, writing, comparisons, and arithmetic operations. A program is a sequence of instructions. The number of instructions performed before the program terminates is called the *running time* and the number of registers required during execution is the *space requirement*. The canonical model of computation within computer science is the *word RAM*, a variation on the RAM formalized by Hagerup [112] but previously considered by Fredman and Willard [97, 98] and even earlier by Kirkpatrick and Reisch [131]. The word RAM models two crucial aspects of real-life computing: (1) computers must store values with finite precision and (2) computers take more time to perform computations whenever the input of the computation (on a constant number of values) is longer. Specifically, the word RAM supports constant-time operations on  $w$ -bit integers, where the *word size*  $w$  is a parameter of the model. These constant-time operations include basic arithmetic, bit-manipulations and ways to index memory.

For certain disciplines in computer science the word RAM assumption, that each value in your memory can be specified with a finite number of bits, can be limiting for algorithmic analysis. This is because many algorithmic procedures either explicitly or implicitly require comparisons between high-precision or even real values. Consider the following example within computational geometry (Figure 3.1): fix a word size  $w$  and a polygonal domain  $P$  where every vertex of the polygon has coordinates that are specified with  $w$ -bit integers. Suppose that we would want to know the Euclidean shortest path from a vertex  $v$  to a vertex  $w$  in  $P$ . Out of all possible paths from  $v$  to  $w$ , we want to find the shortest hence we need to compare their lengths. However, the length of any path is the sum of some Euclidean distances: the sum of square roots of a collection of integers cannot be expressed with a finite number of integers and hence on a word RAM this value cannot be expressed within finite memory. Yet from an algorithmic point of view, it is interesting to study how to compute the shortest path in a polygonal domain.

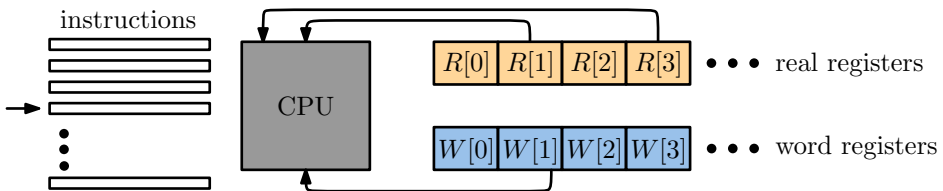
A large part of the algorithms community (either explicitly or implicitly) use a variation of the RAM called the *real RAM*, where registers may contain arbitrary real numbers, instead of just integers. The usage of the real RAM is prevalent in the field of computational geometry but also in probabilistic algorithm analysis where one wants to reason about continuous perturbations of the input. The abstraction offered by the real RAM dramatically simplifies the design and analysis of algorithms, at the cost of working in a model that is physically unrealistic. Implementations of real RAM algorithms using finite-precision data types are prone to errors, not only because the output becomes imprecise, but also because rounding errors can lead the algorithm into inconsistent states. Kettner [128] provides an overview of complications that arise from the unrealistic precision that the real RAM assumes.



**Figure 3.1** (a) An orange vertex  $v$  and blue vertex  $w$  in a polygonal domain where each vertex can be specified with constant bit precision. (b) The shortest path from  $v$  to  $w$  is either of these two paths. (c) To compute the shortest path from  $v$  to  $w$  we need to be able to compare two sums of square roots.

**A high-level overview of how we define real RAM algorithms.** The real RAM has been the standard underlying model of computation in computational geometry since the field was founded in the late 1970s [174, 184]. Despite its ubiquity, we are unaware of any published definition of the model that is simultaneously precise enough to support our results and broad enough to encompass most algorithms in the algorithm analysis literature. Specifically, many RAM proposals do not model the exact operations and input they allow.

The obvious candidate for such a definition is the real-computation model proposed by Blum, Shub, and Smale [28, 29]; this model simply assumes that all numbers in a program are real values and has a table of operations on these real values that can each be executed in unit time. However, this model does not support the integer operations necessary to implement even simple algorithms! Even though the real RAM is often presented as a RAM that stores and manipulates only exact real numbers, countless algorithms in this model require decisions based on both exact real and finite precision integer values. Consider the following example: given an array of  $n$  real values as input, compute their sum. Any algorithm that computes this sum must store and manipulate real numbers; however, a straightforward algorithm also requires indirect memory access through an *integer* array index. More complex examples include call stack maintenance, discrete symbol manipulation, and program slicing.



**Figure 3.2** The dominant model in computational geometry is the real RAM. We define it to have a central processing unit, which can operate on real and word registers in constant time, following a set of instructions.

On the other hand, real and integer operations must be combined with care to avoid unreasonable discrete computation power. A model that supports both exact constant-time real arithmetic and constant-time conversion between real numbers and integers, for example using the floor function, would also trivially support arbitrary-precision constant-time integer arithmetic (to multiply two integers, cast them both to reals, multiply them, and cast the result back to an integer.) Including such constant-time operations allows any problem in PSPACE to be solved in polynomial-time [181]; see also [26, 116, 131, 196] for similar results. We follow [125] and [67] to support both integer and real operations and extend their models by explicitly defining the input and instructions available to the machine.

To accommodate this mixture of real and integer operations, and to avoid complexity pitfalls, we define the real RAM as an extension of the standard integer word RAM [112] (refer to Figure 3.2). We define the real RAM in terms of a fixed parameter  $w$ , called the *word size*. A *word* is an integer between 0 and  $2^w - 1$ , represented as a sequence of  $w$  bits. Mirroring standard definitions for the word RAM, memory consists of two *random access arrays*  $W[0, \dots, 2^w - 1]$  and  $R[0, \dots, 2^w - 1]$ , whose elements we call *registers*. Both of these arrays are indexed/addressed by words; for any word  $i$ , register  $W[i]$  is a word and register  $R[i]$  is an exact real number. We sometimes refer to a word as an *address* when it is used as an index into a memory array.

A program on the real RAM consists of a fixed, finite indexed sequence of read-only instructions. The machine maintains an integer *program counter*, which is initially equal to 1. At each time step, the machine executes the instruction indicated by the program counter. The GOTO instruction modifies the program counter directly; the HALT and ACCEPT and REJECT instructions halt execution. Otherwise, the program counter increases by 1 after each instruction is executed.

The input to a real RAM program consists of a pair of vectors  $(a, b) \in \mathbb{R}^n \times \mathbb{Z}^m$ , for some integers  $n$  and  $m$ , which are suitably encoded into the corresponding memory arrays before the program begins.<sup>1</sup> To maintain uniformity, we require that neither the input sizes  $n$  and  $m$  nor the word size  $w$  are known to any program at ‘compile time’. The output of a real RAM program consists of the contents of memory when the program executes the HALT instruction. The *running time* of a real RAM program is the number of instructions executed before the program halts; each instruction requires one time step by definition. The *space required* by a real RAM program is the maximum number of registers in use over all steps of the program.

Given this high-level overview of input, output and computations, we are ready to formally define the instructions available to our Random Access Machine. For now, we consider the real RAM without trigonometric, exponential and logarithmic operations or the square root operator. In Section 3.1.3, we revisit our definition and discuss how the square root operator can be added to this model of computation without adding unreasonable computational power.

<sup>1</sup>Following standard practice, we implicitly assume throughout the paper that the integers in the input vector  $b$  are actually  $w$ -bit words; for problems involving larger integers, we take  $m$  to be the number of words required to encode the integer part of the input.

| Class                  | Word                                 | Real                               |
|------------------------|--------------------------------------|------------------------------------|
| Constants              | $W[i] \leftarrow j$                  | $R[i] \leftarrow 0$                |
|                        |                                      | $R[i] \leftarrow 1$                |
| Memory                 | $W[i] \leftarrow W[j]$               | $R[i] \leftarrow R[j]$             |
|                        | $W[W[i]] \leftarrow W[j]$            | $R[W[i]] \leftarrow R[j]$          |
|                        | $W[i] \leftarrow W[W[j]]$            | $R[i] \leftarrow R[W[j]]$          |
| Casting                | —                                    | $R[i] \leftarrow j$                |
|                        | —                                    | $R[i] \leftarrow W[j]$             |
| Arithmetic and Boolean | $W[i] \leftarrow W[j] \boxplus W[k]$ | $R[i] \leftarrow R[j] \oplus R[k]$ |
| Comparisons            | IF $W[i] = W[j]$ GOTO $\ell$         | IF $R[i] = 0$ GOTO $\ell$          |
|                        | IF $W[i] < W[j]$ GOTO $\ell$         | if $R[j] > 0$ GOTO $\ell$          |
| Control flow           | GOTO $\ell$                          |                                    |
|                        | HALT / ACCEPT / REJECT               |                                    |

**Table 3.1** A table showing all constant time RAM operations applicable to word or real registers. The values  $i, j, k$  are constant words used for indexing. The operator  $\leftarrow$  assigns a value to a register. By  $\boxplus$  and  $\oplus$  we denote the arithmetic operations specified in Section 3.1.1.

### 3.1.1 Instructions of a real RAM

The real RAM has access to both word and real registers. Therefore, the input to a real RAM algorithm consists of a pair of vectors  $(a, b) \in \mathbb{R}^n \times \mathbb{Z}^m$  (for some integers  $n$  and  $m$ ) which are suitably encoded into the corresponding memory arrays before the algorithm begins. Our model assumes that there is some fixed word size  $w$ . Following Fredman and Willard [97, 98] and later users of the word RAM, we assume that  $w = \Omega(\log N)$ , where  $N = n + m$  is the total size of the problem instance at hand. This so-called *transdichotomous* assumption implies direct constant-time access to the input data (each address can be specified with a constant word). Table 3.1 provides a summary of the specific instructions our model supports. Most notably, after each operation the program counter increases by 1, apart from comparisons and control flow operations. All *word* operations operate on words and produce words as output; all *real* operations produce real numbers as output. Each operation is parametrized by a small number of constant words  $i, j$ , and  $k$ , which index the address of the registers that are manipulated by the operation. Each instruction requires one time step by definition.

Our model supports the following specific word operations ( $\boxplus$ ), all arithmetic operations interpret words as non-negative integers between 0 and  $2^w - 1$ :

- addition:  $x \leftarrow (y + z) \bmod 2^w$
- subtraction:  $x \leftarrow (y - z) \bmod 2^w$
- lower multiplication:  $x \leftarrow (yz) \bmod 2^w$
- upper multiplication:  $x \leftarrow \lfloor yz/2^w \rfloor$
- rounded division:  $x \leftarrow \lfloor y/z \rfloor$ , where  $z \neq 0$
- remainder:  $x \leftarrow y \bmod z$ , where  $z \neq 0$
- bitwise nand:  $x \leftarrow y \uparrow z$  (that is,  $x_i \leftarrow y_i \uparrow z_i$  for every bit-index  $i$ )

Other bitwise Boolean operations can be implemented by composing bitwise nands. Similarly, our model supports the following *exact* real operations ( $\oplus$ ):

- addition:  $x \leftarrow y + z$
- subtraction:  $x \leftarrow x - y$
- multiplication:  $x \leftarrow y \cdot z$
- exact division:  $x \leftarrow y/z$ , where  $z \neq 0$
- (optional) exact square root:  $x \leftarrow +\sqrt{y}$ , where  $y \geq 0$

To avoid unreasonable computational power, we do not allow casting real variables to integers (for example, using the floor function  $\lfloor \cdot \rfloor$ ), or testing whether a real register actually stores an integer value, or any other access to the binary representation of a real number (each of these can be used to simulate a constant-time floor operation [181]). We *do* allow casting integer variables to reals, as this is a requirement for the real RAM equivalent of the Cook-Levin theorem [83].

### 3.1.2 Simulating real RAM computations on a word RAM

Now that we have formally defined the operations available to the real RAM, we reason about how to simulate the execution of a real RAM program on a word RAM. This simulation is the justification for our model definition, as this will show that real RAM programs can be executed on a more realistic word RAM machine. Recall that after every instruction the program counter increases by one, except for when the instruction is a comparison or control flow instruction. Imagine two inputs  $I$  and  $I'$  and an algorithm  $A$  such that after every operation their program counter is identical. These two inputs then produce the same output and are, intuitively, equivalent.

Formally, we say two inputs  $I = (a, b) \in \mathbb{R}^n \times \mathbb{Z}^m$  and  $I' = (a', b') \in \mathbb{R}^n \times \mathbb{Z}^m$  are equivalent with respect to an algorithm  $A$  on a real RAM, if every comparison operation gives the same result (we denote this by  $I \cong_A I'$ ). This implies that after every step the program counter has the same value. For each integer  $z \in \mathbb{Z}$ , we denote by  $\text{bit}(z)$  the length of its binary representation, i.e.,  $\text{bit}(z) = \lfloor \log_2(|z| + 1) \rfloor + 1$ . For each rational number  $y = p/q \in \mathbb{Q}$ , we define the length of its binary representation as  $\text{bit}(y) := \text{bit}(p) + \text{bit}(q)$ .

First we consider not real input, but rational input  $I = (a, b) \in \mathbb{Q}^n \times \mathbb{Z}^m$  for a real RAM algorithm  $A$ . For a fixed algorithm  $A$ , we denote by  $\text{bit}_{\text{IN}}(I) = \max_i \max\{\text{bit}(a_i), \text{bit}(b_i)\}$  the *input bit length* of  $I$ . We denote  $C(I)$  as the set of all values of all registers during the execution of  $A$  with  $I$ . For a fixed algorithm  $A$ , we define the *execution bit length* of  $I$  as:

$$\text{bit}(C(I)) = \max_{c \in C(I)} \{\text{bit}(c)\}.$$

Now, we are ready to define the bit precision and input precision of *real* input. The bit precision of  $A$  with input  $I = (a, b) \in \mathbb{R}^n \times \mathbb{Z}^m$  is:

$$\text{bit}(I, A) := \min\{\text{bit}(C(I')) \mid I' \in \mathbb{Q}^n \times \mathbb{Z}^m, I \cong_A I'\}.$$

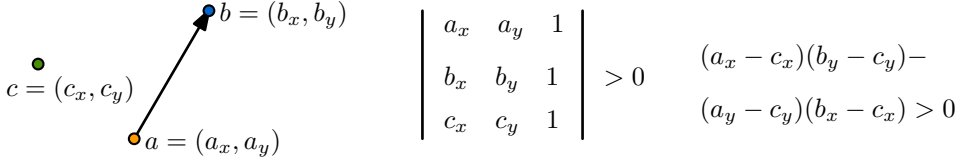
In the same manner, the input precision is defined as :

$$\text{bit}_{\text{IN}}(I, A) := \min\{\text{bit}_{\text{IN}}(I') \mid I' \in \mathbb{Q}^n \times \mathbb{Z}^m, I \cong_A I'\}.$$

If, for some input  $I$ , there is no equivalent input  $I' = (a, b) \in \mathbb{Q}^n \times \mathbb{Z}^m$ , then we say  $\text{bit}(I, A) = \text{bit}_{\text{IN}}(I, A) = \infty$ . For any input  $I$  for which  $\text{bit}(I, A) \neq \infty$ , it is now straightforward to *simulate* an execution of a real RAM algorithm  $A$  on input  $I = (a, b) \in \mathbb{R}^n \times \mathbb{Z}^m$  on a word RAM with word size  $w = O(\text{bit}(I, A))$  using its equivalent input  $I'$ .

In the introduction, we referred to the intuitive notion of *robustness*. We use the above observations to formalise the definition of robustness. We say that a real RAM algorithm  $A$  is *robust* if for all inputs  $I \in \mathbb{R}^n \times \mathbb{Z}^m$ ,  $\text{bit}(I, A) = O(\log n)$ . Showing robustness for an algorithm is not straightforward: it is difficult to reason about the bit precision of the registers of the execution of an algorithm after *any* operation of its program. Luckily, for a wide category of algorithms, we show that it is enough to reason about the input precision only.

**Arithmetic degree and input precision.** Liotta, Preparata and Tamassia [143] studied the required bit precision for real RAM proximity (nearest-neighbor) algorithms. To aid their analysis, they defined the *arithmetic degree* of an algorithm. We express their definition in our real RAM model and add the notion of algebraic dimension for our later analysis. Observe that adding, subtracting, or multiplying two rational functions yields another rational function, possibly of higher degree; for example,  $\frac{p_1}{q_1} + \frac{p_2}{q_2} = \frac{p_1 q_2 + p_2 q_1}{q_1 q_2}$ . It follows from our list of operations in Section 3.1.1 that at all times during the computation, a real register holds a value which can be described as the quotient of two polynomials  $\frac{p}{q}$  whose variables are the real input values  $a$ . We say an algorithm  $A$  has *arithmetic degree*  $\Delta$ , if  $p$  and  $q$  always have total degree at most  $\Delta$  (Figure 3.3). Similarly,  $A$  has *algebraic dimension*  $d$ , if the number of variables in  $p$  and  $q$  is always at most  $d$ . Bounded algebraic dimension and arithmetic degree give an interesting relation between the input precision and the bit precision:



**Figure 3.3** Consider the naive  $O(n^3)$  time algorithm that determines the order type of a planar point set (for every triple of points, perform an orientation test). This algorithm has total arithmetic degree  $\Delta = 2$  and dimension  $d = 6$ . This is because the orientation test dominates the arithmetic degree and dimensions of the operations required to store the solution.

**Lemma 3.1** *Let  $I \in (a, b) \in \mathbb{R}^n \times \mathbb{Z}^m$  be some input and  $A$  be a real RAM algorithm with  $\text{bit}_{\text{IN}}(I, A) = p$ , algebraic dimension  $d$  and arithmetic degree  $\Delta$ . Then its bit precision  $\text{bit}(I, A)$  is at most  $O(p\Delta^2 \log d)$ .*

**Proof** We prove a slightly stronger statement: suppose that every variable can be expressed with  $p$  bits each. Then at every time, the exact value of a register can be expressed with at most  $O(p\Delta^2 \log d)$  bits.

Indeed, if we multiply  $\Delta$  numbers of  $p$  bits each then the total number of bits needed to express this value is at most  $\Delta p$ . If we sum  $t$  numbers of  $p$  bits, the total number of bits needed to express this value is at most  $O(p \log t)$ . In a polynomial in  $d$  variables, we have at most  $O(d^\Delta)$  monomials. Thus the bit precision is upper bounded by  $O(\log(d^\Delta)\Delta p) = O(p\Delta^2 \log d)$ . ■

Hence, for a wide class of algorithms, it is enough to focus the algorithmic analysis (with respect to bit precision) on input precision.

### 3.1.3 Allowing the square root operator.

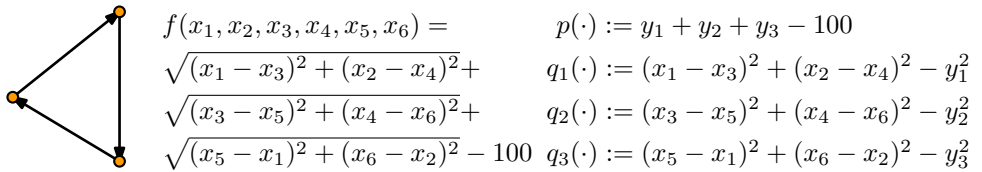
Until this point, we worked under the assumption that at all times a real register  $R[i]$  contains a value that can be expressed as the quotient of two  $d$ -variate polynomials  $f_i, g_i$  of maximal total degree  $\Delta$ . This definition achieves the following: if the input has some bounded bit precision, we know the bit precision needed to express an exact value in each register (see the proof of Lemma 3.1). For radical expressions, it is impossible to express the exact value in each register with a limited number of bits, even if the input of the radical expression has limited bit precision (for example, we cannot express the exact value of  $\sqrt{2}$  with a bounded number of bits). This makes it difficult to reason about how a word RAM would simulate a program where register values include radical expressions. At the same time, it is crucial to support real RAM programs that make use of the square root operator if we want to support algorithmic problems often considered in computational geometry.

Demaine, Hesterberg and Ku [67] also observe that the word RAM cannot immediately support evaluating radical expressions with limited bit precision. They observe that whenever a constant-size radical expression (this expression may include the addition, multiplication and division of square roots) depends on variables that can be expressed with  $w$  bits each, an algorithm can decide if the expression is greater than 0 in  $O(w)$  time. They informally note that if an algorithm makes a correct comparison decision at every step of the algorithm, then the algorithm must reach the correct control flow instructions. Following this observation the authors assume a word RAM which allows memory registers to store expressions of  $w$  bits, in addition to constant-size radical expressions. This makes their model more general than a traditional word RAM and their observation serves as a practical justification for their assumption. At the same time, their assumption is less general than the real RAM because they require all expressions to have constant size. Note that computing the shortest Euclidean path in a polygonal domain is not possible in their model, as the length of a linear-size path is not a constant-size radical expression.

In this thesis, we generalise their assumption. We allow registers to contain arbitrary radical expressions and we parametrize radical expressions to reason about the bit precision needed on a word RAM to correctly evaluate comparison instructions on the registers. Specifically, we allow arbitrary application of the square root function which includes recursive applications of the square root function. We parametrize the complexity that the square root operations introduce by some integer  $s$ .

Formally, we extend our real RAM definition so that every real register holds an expression  $f(x_1, \dots, x_d)$ , which contains multiplication, addition, subtraction, division and square roots. Let  $f$  be such an expression and consider polynomials  $p, q_1, \dots, q_s$  with variables  $[x_1, x_2, \dots, x_d, y_1, \dots, y_s]$ . We say that  $f$  is equivalent to the polynomials  $p, q_1, \dots, q_s$  if and only if (Figure 3.4):

every assignment of variables for which  $[q_1, \dots, q_s = 0] \Rightarrow \text{sign}(p) = \text{sign}(f)$ .



**Figure 3.4** An expression  $f$  that verifies if a path has length less than 100, when distance is measured as Euclidean distance. We can create four polynomials  $p, q_1, q_2, q_3$  with variables  $[x_1, \dots, x_6, y_1, y_2, y_3]$  such that for every assignment of variables where  $[q_1, \dots, q_s = 0]$  it must be that  $\text{sign}(p) = \text{sign}(f)$ .

Intuitively, we perform the comparison instruction not on  $f$  but on a set of equivalent polynomials which we will obtain with Lemma 3.2. We refer to  $p$  as the *evaluation polynomial* and  $q_1, \dots, q_s$  the *constraint polynomials*. We define the arithmetic degree of  $f$  as the maximum arithmetic degree of  $p, q_1, \dots, q_s$ . We say the algebraic dimension of  $f$  equals  $d$ , and  $s$  is denoted as the *extra algebraic dimension*. We say an algorithm has arithmetic degree  $\Delta$ , algebraic dimension  $d$  and extra algebraic dimension  $s$  whenever all real registers at time of comparison have those properties. Formally, the following lemma provides a recursive procedure to extract from each expression  $f$  its arithmetic degree, algebraic dimension and the number  $s$  of extra algebraic dimensions:

**Lemma 3.2** *Let  $f$  be a radical expression on variables  $x_1, \dots, x_d$ , with at most  $s$  square roots and divisions. Then we can transform  $f$  into equivalent polynomials through introducing at most  $s$  constraint polynomials:  $p, q_1, \dots, q_s \in \mathbb{Z}[x_1, \dots, x_d, y_1, \dots, y_s]$ .*

**Proof** We transform  $f = f_0$  step by step into expressions  $f_1, \dots, f_s$  such that the final expression  $f_s = p$  is a polynomial. At each step  $i$ , we create one additional polynomial  $q_i$ . Specifically, at step  $i$  we distinguish between two cases, and always introduce the variable  $y_i$ :

The first case is that the expression  $f_{i-1}$  has the form  $f_{i-1} = g(\sqrt{h})$ , where  $h$  is a polynomial and  $g$  is some other expression. We define  $f_i = g(y_i)$ , with the new variable  $y_i$  and add the constraint  $q_i = y_i^2 - h$  which is a polynomial expression.

The second case is that  $f_{i-1}$  has the form  $f_{i-1} = e(g/h)$ , where  $g, h$  are a polynomials and  $e$  is some other expression. Note that  $e, g$  and  $h$  may also depend on other variables. Then we define  $f_i = e(g \cdot y_i)$  and  $q_i = y_i \cdot h - 1$ . ■

In full generality, the above lemma allows us to model and parametrize nested constant order radicals. For example,  $\sqrt{\sqrt{x_1} + \sqrt[3]{x_2} + x_3^5}$ , would be a valid expression for a real register.

**Corollary 3.1** *The length of a Euclidean shortest path in a polygonal domain with  $n$  vertices is a radical expression with arithmetic degree  $\Delta = 2$ , algebraic dimension  $d = O(n)$  and extra algebraic dimension  $s = O(n)$ .*

Now that we have introduced our model of computation, and a way to parametrize algorithms based on the complexity of the expressions encountered during computations, we are ready to analyse the real RAM.

## Chapter Four

# Smoothed analysis of the real RAM

It is evident that the real RAM is instrumental for algorithmic analysis within computational geometry. The downside of algorithmic analysis in the real RAM is that it neglects the fact that a computer has limited bit precision to support the operations underlying algorithms, even though the bit precision that an algorithm has access to is very important in practice: if for a given algorithm its bit precision is not polynomial then executing the algorithm becomes infeasible. In the remainder of this chapter, we apply smoothed analysis to show that for a wide class of algorithms in computational geometry, their bit precision is bounded *in practice*. Before we formalise these concepts we briefly review the literature.

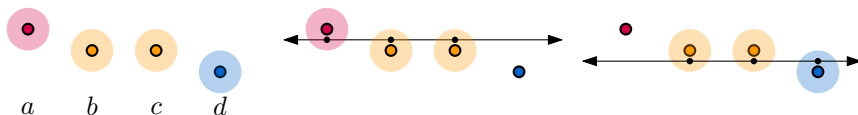
**Robust geometric computations.** The idea that some real RAM algorithms cannot be executed with bounded precision is not new. Salesin, Stolfi and Guibas [178] note that many classical examples in computational geometry are inherently nonrobust. There are even example algorithms and inputs that require a bit precision exponential in  $n$  in order to be correctly executed [102, 124].

Yao and Sharma present in the Handbook of Discrete and Computational Geometry [191, Ch. 45] an extensive overview of techniques used to obtain robust geometric computations. They describe two broad paradigms to obtain provably robust computations. In the first paradigm, called *fixed precision approaches*, the goal is to execute a geometric algorithm using fixed bit precision. With fixed bit precision, it can become impossible to correctly test certain geometric primitives (e.g. in-circle testing, collinearity testing) and it is therefore impossible to prove that the output is correct according to these geometric primitives. This is why approaches under this paradigm invent alternative (weaker) geometric primitives and they prove that they can construct output which satisfies the alternative primitives.

The fixed paradigm includes well-known approaches such as *interval geometry* [88, 88, 183],  *$\varepsilon$ -geometry* [178], *fuzzy sets* [177], *strong algorithmic stability* [94] and the more recent *topological stability* [122, 162]. Comparatively recent examples that fall under the fixed precision paradigm are papers that discuss the construction of almost-Delaunay simplices [15], Delaunay triangulations of imprecise points [149] and stability analysis of Voronoi diagram [175]. An appealing property of this paradigm is that solutions are provably correct or stable. However, the notion of stability or correctness at times may be undesirable. For example under  $\varepsilon$ -geometry it is possible to have three planar points  $(a, b, c)$  on a horizontal line, three points  $(b, c, d)$  on a horizontal line, even though the triple  $(a, b, d)$  does not lie on a horizontal line (Figure 4.1).

The second paradigm Yao and Sharma call the *exact approach*. Here, geometric primitives are given a representation that allows their precision to be lazily evaluated. Such an algorithm detects if a primitive can be correctly determined with limited bit precision and if it cannot, the algorithm can increase the bit precision of the involved variables accordingly. This approach is sometimes referred to as *exact geometric computation* (EGC); it is an active research field in experimental computational geometry [15, 30, 69, 95, 115, 204, 205, 206]. There are also theoretical results in this paradigm such as recent work which computes the expected running time of computing a Delaunay triangulation [40], verifies order type representation [46, 69] or a deterministic subquadratic precision bound for representing order types without coordinates [45]. Perhaps most notable is the fact that the CGAL Core library makes use of the EGC principle [30, 113, 115]. We propose a third paradigm which is to apply *smoothed analysis* to the bit precision of an algorithm.

**Smoothed analysis.** Smoothed analysis is a technique used in parametrized complexity that recently has received much attention. In smoothed analysis, the performance of an algorithm is studied for worst case input which is randomly perturbed by a magnitude of  $\delta$  (that is, we consider out of all inputs precisely the input that is least-beneficial to perturb from). Intuitively, smoothed analysis interpolates between average case and worst case analysis (Figure 4.2). The smaller the parameter  $\delta$ , the closer we are to true worst case input while larger  $\delta$  is closer to the average case analysis. The key difficulty in applying smoothed analysis is that one has to simultaneously argue about both worst case and average case input.



**Figure 4.1** Four points  $(a, b, c, d)$ . Under  $\varepsilon$ -geometry, a geometric predicate is true, if for every point in the predicate, there is a representative in a ball of radius  $\varepsilon$  that makes the predicate true. We can choose three points in the balls of  $(a, b, c)$  to make them coincide with a horizontal line. Similarly for  $(b, c, d)$  but not for  $(a, b, d)$ .

Spielman and Teng explain their analysis by applying it to the simplex algorithm, which was known for a particularly good performance in practice that was seemingly impossible to verify theoretically [133]. Since its introduction, smoothed analysis, has been often applied [17, 79, 84, 167], also in computational geometry [27, 155, 154].

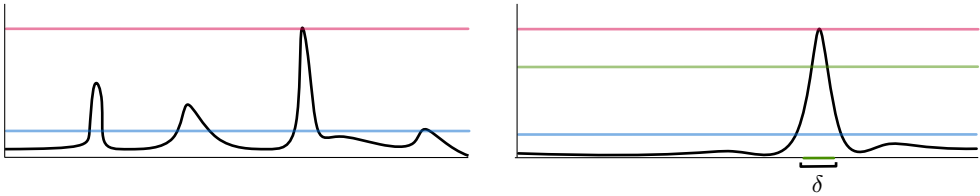
Formally we define smoothed analysis as follows: let us fix an algorithm  $A$  and some  $\delta \in [0, 1]$ , which describes the *magnitude of perturbation*. We denote by  $I = (a, b) \in [0, 1]^n \times \mathbb{Z}^m$  the input of  $A$ . We scale the real-valued input to lie in  $[0, 1]$ , to normalize  $\delta$ . Unless explicitly stated, we assume that each real number is perturbed *independently uniformly at random* and that the integers stay as they are, since we assume that they already fit into main memory. We denote by  $(\Omega_\delta, \mu_\delta)$  the probability space where each  $x \in \Omega_\delta$  defines for each instance  $I$  a new ‘perturbed’ instance  $I_x = (a + x, b)$ . We denote by  $\mathcal{C}(I_x)$  the cost of instance  $I_x$ . Traditionally in smoothed analysis, this cost is the runtime required for an algorithm in order to compute its solution. In this chapter, we consider either the input precision or the bit precision as the cost function. For a fixed  $\delta$ , the expected cost of instance  $I$  equals:

$$\mathcal{C}_\delta(I, A) = \mathbb{E}_{x \in \Omega_\delta} [\mathcal{C}(I_x)] = \int_{\Omega_\delta} \mathcal{C}(I_x) \mu_\delta(x) dx.$$

We denote by  $\Lambda_{n,m}$  the set of all instances in  $[0, 1]^n \times \mathbb{Z}^m$ . Henceforth, we implicitly assume that for all integer values  $b_i$ ,  $\text{bit}(b_i) \leq \log m$  and  $m = O(n)$  (so that we may drop  $m$  from all future complexity expressions). The smoothed precision of an algorithm  $A$ , with real input size  $n$  and perturbation  $\delta$  equals:

$$\mathcal{C}_{\text{smooth}}(\delta, n, A) = \max_{I \in \Lambda_{n,m}} \mathcal{C}_\delta(I, A).$$

This definition formalizes the intuition mentioned before: not only do we require that the majority of instances behave nicely, but actually in every neighborhood (bounded by the maximal perturbation  $\delta$ ) the majority of instances behave nicely.



**Figure 4.2** The  $x$ -axis symbolizes all  $n$ -dimensional real-valued inputs. The red line indicates the worst case cost. The blue line indicates average cost. If we pick a parameter  $\delta$  and a point  $a$  on the  $x$ -axis, then all permutations from  $a$  of at most  $\delta$  form some area centered on  $a$ . We can compute the average cost of points in this area. The cost under smoothed analysis is the maximum over all points  $a$ , of this average cost and is shown by the green line.

**Smoothed complexity for practical robustness.** Following [58, 187] we perceive an algorithm to run with logarithmic cost in practice, if the smoothed cost of the algorithm is logarithmic in the input size  $n$  and in  $1/\delta$ . If the smoothed cost is small in terms of  $1/\delta$  then we have a theoretical verification of the hypothesis that worst case examples (inputs, for which the non-smooth cost is high) are sparse. Specifically, we say that an algorithm  $A$  requires logarithmic bit precision in practice if its smoothed bit precision is logarithmic in  $n/\delta$ , or formally when:

$$\text{bit}_{\text{smooth}}(\delta, n, A) = \max_{I \in \Lambda_{n,m}} \mathbb{E}_{x \in \Omega_\delta} [\text{bit}(I_x, A)] = O(\log(n/\delta)).$$

In the remainder of this chapter, we show (with the help of Lemma 3.1) that a wide class of real RAM algorithms is robust in practice. To this end, we want to model disparities between real RAM and word RAM execution. Thus, we define an operation called *snapping* in the next paragraph.

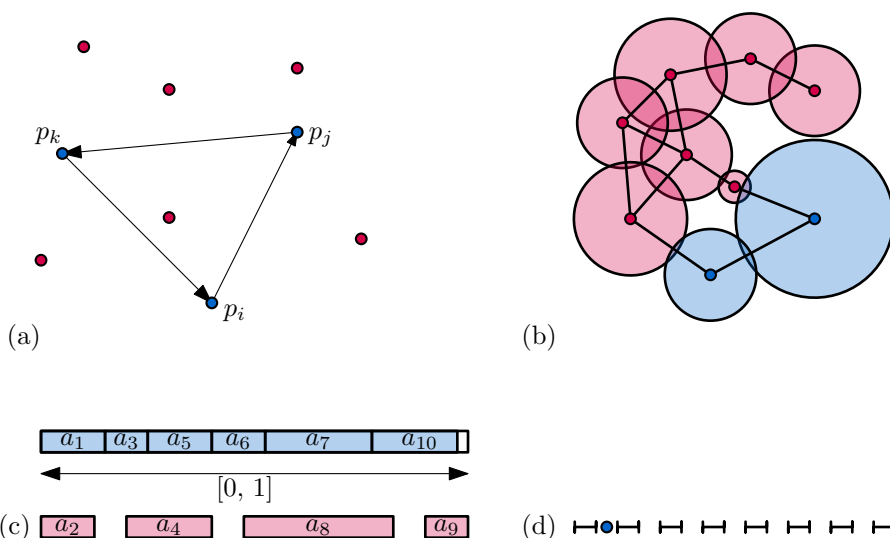
**Snapping.** Our real-valued input can be represented as a higher-dimensional point  $a \in [0, 1]^n$ . If we want to express  $a$  with only  $w \cdot n$  bits, then the corresponding integer-valued input  $a'$  with limited precision is the closest point to  $a$  in the scaled integer lattice  $\Gamma_\omega = \omega \mathbb{Z}^n$  with  $\omega = 2^{-w}$ . We call the transformation of  $a$  into  $a'$  *snapping*.

**Comparison set size.** For our algorithmic analysis, we parametrize all real RAM algorithms based on what we will call the size of their comparison set. Before we present its formal definition, we first discuss its intuition. Consider a real RAM algorithm over some input  $(a, b) \in \mathbb{R}^n \times \mathbb{Z}^m$ . For every fixed pair  $(a, b)$  there is some corresponding output. By our real RAM definition of Chapter 3, the output is obtained by iteratively performing single instructions. After every instruction, the program counter goes to a predetermined value (the counter increases by one, or gets set to some fixed predetermined value  $\ell$ ), except for comparison instructions where the program counter may either increase, or get set to some fixed predetermined value  $\ell$ . The output of a program is therefore entirely decided by the outcome of every comparison instruction. Every real comparison instruction compares a polynomial expression (or radical expression, when the square root operation is allowed) where the polynomial's variables are elements of the input  $a \in \mathbb{R}^n$  to see if the value of the expression is less than zero. We are interested in the probability that for each expression evaluates to less than zero when the input is some perturbed input  $a_x \in \mathbb{R}^n$  if and only if the expression evaluates to less than zero when the input is the snapped input  $a'$ . Specifically, in the remainder of this chapter, we perform smoothed analysis to show that (in expectation) the (perturbed) real input  $a$  and the snapped input  $a'$  have the same outcome for all comparison instructions the program encounters. Hence, the program with input  $a_x$ , or input  $a'$ , reaches the same output. To this end, we parametrize our algorithms by the number of real comparisons an algorithm *may* encounter. For a fixed real input size  $(n, m)$  we denote by  $P(n, m)$  the size of the real comparison set: the number of different comparison expressions that may be encountered by the execution of the program with input size  $n$ .

Formally, we denote for a fixed input  $(a, b) \in \mathbb{R}^n \times \mathbb{Z}^m$  by  $\text{Expr}(n, m)$  the set of all expressions considered by real comparison instructions (sets in this thesis are assumed to never contain any duplicates). The comparison set size  $P(n, m)$  is then defined as follows:

$$P(n, m) := \left| \bigcup_{(a,b) \in \mathbb{R}^n \times \mathbb{Z}^m} \text{Expr}(n, m) \right|.$$

We perform our smooth analysis, where in every theorem statement we explicitly assume that  $P(n, m) = n^{O(1)}$ . To illustrate our definition, we briefly show some algorithmic problems and the corresponding value  $P(n, m)$  (Figure 4.3). Consider the algorithmic problem of computing the order type of a set of  $n/2$  real-valued ordered points (the order type records for every triple of points  $(p_i, p_j, p_k)$  with  $i < j < k$ , the orientation of the triangle spanned by the three points). Computing the order type of a triple of points may be done by computing the sign of the determinant of some fixed matrix where the values of the matrix correspond to the coordinates of the three points. Since there are at most  $O(n^3)$  different triples of points, and algorithms that compute an order type never have to cast input integers to reals, the total number of real-valued comparison expressions of an algorithm that computes an order type is  $P(n, m) = O(n^3)$  (for all  $m$ ).



**Figure 4.3** Four examples of algorithmic problems where we want to upper bound the size of their real comparison set: (a) Computing the order type of a point set. (b) Computing a disk intersection graph. (c) The knapsack problem. (d) The even interval problem.

Similarly consider the algorithmic problem of computing a disk intersection graph (where it is sufficient to check for every pair of disks whether they intersect). There are at most  $O(n^2)$  pairs of disks and it follows that for an algorithm that computes the disk intersection graph, for every values of  $m$ ,  $P(n, m) = O(n^2)$ . Finally, we consider two algorithmic problems where  $P(n, m)$  is not polynomially bounded. First, consider the real-weighted, unit-value knapsack problem where the input  $a \in \mathbb{R}^n$  specifies  $n$  objects with some real-valued weight. The goal is to identify the largest subset of the objects where the sum of their weights is at most 1. Depending on the input  $a$ , the output may be any subset of the objects and must compare if the sum of the weights in that subset minus 1 is smaller than zero. Thus, without any additional assumptions, for all  $m$  it must be that  $P(n, m) = \Omega(2^n)$ . Second, we define what we call the even interval problem. Here, the input is a single real number  $a \in [0, 1]$  together with a single value  $b \in \mathbb{N}$  presented in unary with  $k$  bits. We ask if  $a$  is contained in one of the intervals  $I_i = [\frac{2i}{b}, \frac{2i+1}{b}]$ , for some integer  $i$ . This problem can easily be solved with binary search in  $O(\log k)$  steps where at every step we check one polynomial that verifies if  $a$  lies in some interval  $I_i$ . Yet, depending on the value of  $a$ , the set of intervals  $I_i$  that we need to check wildly varies and is roughly  $\Omega(k)$ . Hence  $P(1, \cdot) = \Omega(k)$  even though the size of the real input is constant (the input size of the integer input depends on the word size  $w$ ).

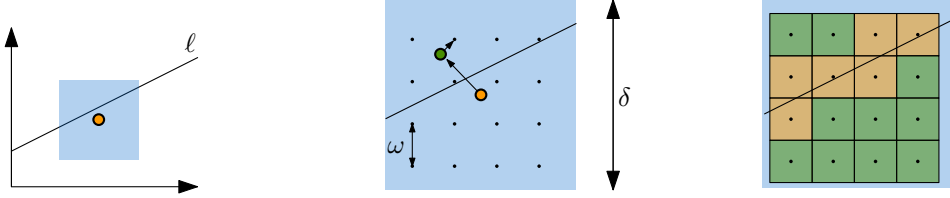
## 4.1 Smoothed analysis of the real RAM

We are now ready to apply smoothed analysis to real RAM algorithms. For ease of exposition, we consider the real RAM without the square root operator. In the next section we then generalise our results. We make use of *snapping* to upper bound the bit precision of a real RAM algorithm. Specifically, when snapping a point  $(a, b) \in [0, 1]^n \times \mathbb{Z}^m$  to a point  $(a', b)$ , we give a lower bound on the scale factor  $\omega$  for which  $(a, b) \cong_A (a', b)$  (for the definition of equivalent input  $\cong_A$  refer to Section 3.1.1). This implies an upper bound on the input precision of  $A$ . We then use Lemma 3.1 to use the bound on the input precision to bound the bit precision of an algorithm.

**Theorem 4.1** *Let  $A$  be a real RAM algorithm with arithmetic degree  $\Delta$ , algebraic dimension  $d$  and extra algebraic dimension  $s = 0$  with input  $(a, b) \in [0, 1]^n \times \mathbb{Z}^m$  and comparison set size  $P(n, m) = n^{O(1)}$ . Under perturbations of  $a \in [0, 1]^n$  by  $x \in [-\frac{\delta}{2}, \frac{\delta}{2}]^n$  chosen uniformly at random,  $A$  has a smooth bit precision of at most:*

$$\text{bit}_{\text{smooth}}(\delta, n, A) = O\left(d \log \frac{d\Delta n}{\delta} \Delta^2 \log d\right).$$

The core idea of this proof (illustrated by Figure 4.4) is to consider the algorithm  $A$  with perturbed input  $a_x = a + x$ , where  $a$  is an arbitrary value in  $[0, 1]^n$  and  $x$  is a small perturbation chosen uniformly at random in  $[-\frac{\delta}{2}, \frac{\delta}{2}]^n$ .



**Figure 4.4** Given  $a = (a_1, a_2) \in [0, 1]^2$ , we want to decide if the point  $a$  (orange) lies above or below the line  $\ell$  ( $y = x/2 + 1$ ). The point  $a$  is randomly perturbed to  $a_x$  (green) in a square of diameter  $\delta$ . The computation on the perturbed point  $a_x$  is performed correctly with low precision whenever  $a_x$  lies in a green cell in the grid.

We model the perturbed input  $a_x$  as a high-dimensional point which we snap to a fine grid to obtain  $a'$  (input which can be described using bounded precision). We then show that for any algorithm  $A$  that meets our prerequisites,  $\text{bit}_{IN}(a_x, A)$  is low with high probability. For the snapping we consider a sufficiently small  $\omega$  and we snap the point  $a_x$  to a point in  $\omega\mathbb{Z}^n$ . The Voronoi diagram of the points in  $\omega\mathbb{Z}^n$  forms a fine grid in  $[0, 1]^n$ . As we explained in Chapter 3, the content of a real RAM register for each comparison instruction is per assumption the quotient of two polynomials whose variables depend on the input. The core argument is that if the point  $a_x$  lies in a Voronoi cell of a point with limited word size which does not intersect the variety of either of the two polynomials, then the comparison instruction will be computed correctly. We upper bound the proportion of Voronoi cells that are intersected by the variety of a polynomial in the algebraic Theorem 4.2. The proof of this algebraic theorem can be found in Section 4.A to not distract from the main story line.

**Preliminaries for proving Theorem 4.1.** Per definition, the word RAM has a word size  $w$  which allows us to express  $2^w = \frac{1}{\omega}$  different values for each coordinate. We consider a fixed algorithm  $A$  with real-valued input  $a \in [0, 1]^n$ . Regardless of the input, by definition, the algorithm  $A$  provides a program that makes at most  $P(n, m)$  comparisons (Table 3.1). At the  $i$ 'th such binary decision, the algorithm looks at a real- or integer-value register and verifies if the value at the register is 0 or strictly greater than 0. For every real-valued register, per assumption the value at that register is the quotient of two  $d$ -variate polynomials  $p_i$  and  $q_i$  with maximum total degree  $\Delta$  whose variables depend only on the values in the input register.

Let  $z$  be the  $d$ -dimensional vector of input variables in  $a$  that  $p_i$  and  $q_i$  depend on ( $z$  is a subset of  $a$ ). The evaluation of any register  $R[i] = p_i(z)/q_i(z)$  depends on the evaluation of  $p_i(z)$  and  $q_i(z)$ . During smoothed analysis we perturb our real input  $a$  into new input  $a_x = (a + x)$  with  $x$  a value chosen uniformly at random in  $[-\frac{\delta}{2}, \frac{\delta}{2}]^m$ . Thereafter, we snap  $a_x$  to  $a'$  and we are interested in the probability that the execution of  $A$  under both inputs ( $a_x$  and  $a'_x$ ) is the same and thus the chance that for all comparison operations, the program ran on both inputs gives the same answer.

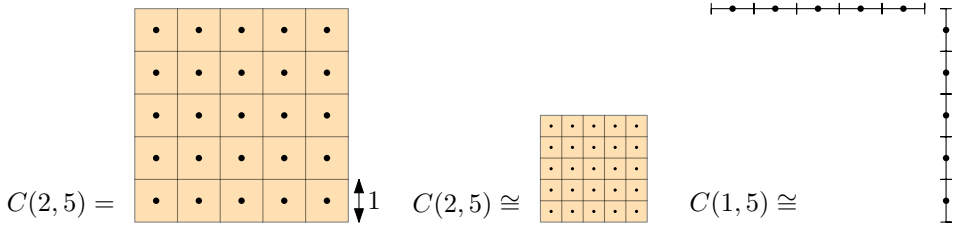
Fix a vector  $z = (z_1, \dots, z_d) \in \mathbb{Z}^d$  with integer coordinates. We denote by  $C_z$  the unit hypercube that has  $z$  as its minimal corner:  $C_z := [0, 1]^d + z$ . We denote by  $C(d, k) := \{C_z \mid z \in \mathbb{Z}^d \cap [0, k]^d\}$  a  $(k \times k \times \dots \times k)$ -grid of unit hypercubes that cover  $[0, k]^d$  (Figure 4.5). Let  $C = [0, k]^d$  be a hypercube partitioned by unit hypercubes  $C(d, k)$ . Every facet of  $C$  is a grid of  $(d - 1)$ -dimensional hypercubes  $C(d - 1, k)$ . The core of our proof in this chapter, is an argument about the proportion of hypercubes in  $C(d, k)$  that are intersected by a variety, by induction on the dimension.

To that end, we define an equivalence relation on sets of hypercubes. Let  $C$  be a  $d$ -dimensional hypercube, partitioned by  $d$ -dimensional hypercubes of equal width. We say  $C$  is *equivalent* to  $C(d, k)$ , denoted by  $C \cong C(d, k)$ , if there exists an affine transformation  $\tau$  of  $C$  such that there is a one-to-one correspondence between hypercubes in  $\tau(C)$  and  $C(d, k)$  where corresponding hypercubes coincide. We give two examples of this equivalence relation that are often used in this chapter:

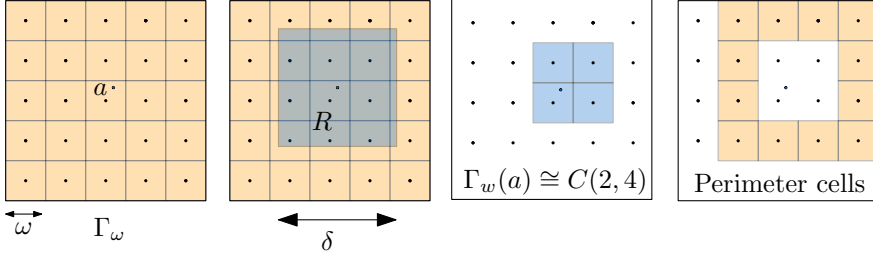
1. Consider any  $d, k$  and any  $(d - 1)$ -dimensional, orthogonal hyperplane  $H$  that intersects  $C(d, k)$  we have that  $C(d, k) \cap H \cong C(d - 1, k)$ .
2. Consider the  $d$ -dimensional grid  $\Gamma_\omega$  as defined for the snapping operation, and a hypercube  $C$  which has one corner on the origin and width  $k\omega$ . The intersection  $C \cap \Gamma_\omega$  is equivalent to  $C(d, k)$ .

In Section 4.A, we use these definitions together with a well-known upper bound on the number of connected components of a variety by Milnor [16] to show:

**Theorem 4.2** *Let  $p \neq 0$  be a  $d$ -variate polynomial with maximum degree  $\Delta$  and let  $k = 2\Delta + 2$ . Then the variety  $V(P)$  of the polynomial  $p \in \mathbb{Z}[x_1, \dots, x_d]$  intersects at most  $k^{d-1} \Delta 3^d (d + 1)!$  unit hypercubes in  $C(d, k)$ .*



**Figure 4.5** We illustrate the grid  $C(2, 5)$ , for some fixed unit size. If we shrink this grid, we obtain a grid  $C' \cong C(2, 5)$ . Observe that each of the facets of  $C(2, 5)$  is equivalent to  $C(1, 5)$ . Similarly, if we were to intersect  $C(2, 5)$  with an orthogonal line, the intersection is equivalent to  $C(1, 5)$ .



**Figure 4.6** Consider a point  $a$  in the grid  $\Gamma_\omega$ . The range  $R$  is the square centered around  $a$  with width  $\delta$ . The set  $\Gamma_\omega(a)$  of cells contained in  $R$  are blue. The remaining cells intersected by  $R$  are the perimeter cells.

**Proving Theorem 4.1.** In smoothed analysis, we start with our real input  $a$ , perturb it to a point  $a_x = (a + x)$  and snap it onto a point with limited precision  $a'$  (the closest point in  $\omega\mathbb{Z}^d \cap [0, 1]^d$ ). In other words, for all  $y \in \omega\mathbb{Z}^d$ , all points in the Voronoi cell of  $y$  are snapped to  $y$ . The Voronoi cells of all these cells (apart from those belonging to grid points on the boundary of  $[0, 1]^d$ ) are just  $d$ -dimensional hypercubes and we shall denote a hypercube of diameter  $\omega$  centered at  $y$  by  $C(y)$ . Given the grid size  $\omega$  we denote the grid by  $\Gamma_\omega = \{C(y) : y \in \omega\mathbb{Z}^d \cap [0, 1]^d\}$ . As this is a  $d$ -dimensional hypercube, partitioned by equal-size hypercubes, we observe that  $\Gamma_\omega \cong C(d, 1/\omega)$  (we use this observation later to apply Theorem 4.2).

The perturbed point  $a_x$  must lie within some hypercube in  $\Gamma_\omega$ . However the choice of original  $a$  and  $\delta$  limits the hypercubes in  $\Gamma_\omega$  where  $a_x$  can lie in. Specifically (Figure 4.6) the range  $R$  of possible locations for  $a_x$  is a hypercube of width  $\delta$  centered around  $a$ . The boundary of this range hypercube  $R$  likely does not align with the boundaries of grid cells in  $\Gamma_\omega$ . Therefore we make a distinction between two types of cells: the range hypercube  $R$  must contain a hypercube of cells which is equivalent to  $C(d, \lfloor \delta/\omega \rfloor)$  and this sub-hypercube of  $R$  we shall denote by  $\Gamma_\omega(a)$ . All others cells which are intersected by  $R$  but not contained in  $\Gamma_\omega(a)$  we call the *perimeter cells*. We can use this observation together with Theorem 4.2 to estimate the probability that for a given polynomial  $p$ :  $\text{sign}(p(a_x)) \neq \text{sign}(p(a'))$ :

**Lemma 4.1** Let  $p, q$  be two  $d$ -variate polynomials with maximum total degree  $\Delta$ . Let  $a \in \mathbb{R}^d$  be fixed and  $x \in \Omega = [-\delta/2, \delta/2]^d$  chosen uniformly at random. Assume  $a_x = a + x$  is snapped to a point  $a'$ . Then  $\text{sign}\left(\frac{p(a_x)}{q(a_x)}\right) \neq \text{sign}\left(\frac{p(a')}{q(a')}\right)$  with probability at most:

$$(4\omega\Delta 3^d(d+1)!)/\delta.$$

**Proof** It suffices to show that  $\text{sign}(p(a_x)) \neq \text{sign}(p(a'))$  with probability at most:

$$\frac{2\omega\Delta 3^d(d+1)!}{\delta}.$$

The statement for  $q$  is equivalent. The union bound on these separate probabilities then upper bounds the probability that their division does not have the same sign. For any polynomial  $p$  and points  $x, z \in \mathbb{R}^d$  with  $p(x) < 0$  and  $p(z) > 0$ , there must be a point  $y$  on the line segment between  $x$  and  $z$  for which  $p(y) = 0$ . It follows from the convexity of hypercubes, that if a hypercube  $C \in \Gamma_\omega$  does not intersect the variety of  $p$ , all points in  $C$  either have a positive or negative evaluation under  $p$ . Let  $a'$  be the closest point to  $a_x$  in  $\omega\mathbb{Z} \cap [0, 1]^d$  and let  $C(a')$  be its Voronoi cell. If  $C(a')$  does not intersect the variety of  $p$  then  $\text{sign}(p(a_x)) = \text{sign}(p(a'))$ . Therefore we are interested in the probability that  $a_x$  is contained in a Voronoi cell intersected by the variety of  $p$ .

As we discussed, the set of possible locations of  $a_x$  is a hypercube which contains  $\Gamma_\omega(a) \cong C(d, \lfloor \delta/\omega \rfloor)$  together with a collection of perimeter cells. We upper bound the probability that  $a_x$  lies within a cell that is intersected by  $p$  in two steps:

1. We upper bound the number of perimeter cells and make the worst case assumption that they are all intersected by the variety of  $p$ .
2. We upper bound the number of cells of  $\Gamma_\omega(a)$  that intersect the variety of  $p$ .

The sum of these two numbers, divided by the total number of cells in  $\Gamma_\omega(a)$ , upper bounds the probability that  $a_x$  lies in a cell for which the comparison instruction *may* give a different answer when processing  $a'$ .

1. The width of the grid  $\Gamma_\omega(a)$  is equal to  $k = \lfloor \frac{\delta}{\omega} \rfloor$  and that the perimeter of  $\Gamma_\omega(a)$  contains  $2d \cdot k^{d-1}$  cells.
2. Theorem 4.2 upper bounds the number of intersected hypercubes in  $C(d, k)$  by:

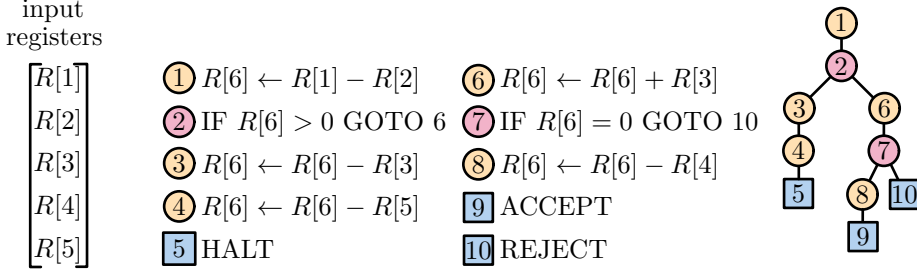
$$k^{d-1} \Delta 3^d (d+1)! \leq \left\lfloor \frac{\delta}{\omega} \right\rfloor^{d-1} \Delta 3^d (d+1)!$$

There are  $k = \lfloor \delta/\omega \rfloor^d$  hypercubes in  $\Gamma_\omega(a)$  and it follows that:

$$\Pr(\text{sign}(p(a_x)) \neq \text{sign}(p(a'))) \leq \frac{(\lfloor \frac{\delta}{\omega} \rfloor)^{d-1} \Delta 3^d (d+1)! + 2d(\lfloor \frac{\delta}{\omega} \rfloor)^{d-1}}{\lfloor \delta/\omega \rfloor^d} \leq \frac{2\omega \Delta 3^d (d+1)!}{\delta}$$

Using this, in conjunction with the union bound, finishes the proof. ■

Lemma 4.1 is used to upper bound the probability that snapping the perturbed input  $a_x$  changes a single real-valued comparison in  $A$ . The output of the program for input  $a_x$  and  $a'$ , is identical if at every real-valued comparison the outcome is the same (Figure 4.7). Therefore, we use the union bound to upper bound the probability that for the whole algorithm (which uses at most a polynomial number of comparison instructions) any real-valued comparison for  $a_x$  and  $a'$  is different:



**Figure 4.7** All programs can be transformed into a tree where every vertex has degree at most 3 and all leaves are either a halt, accept or reject instruction. For two different inputs, if at every comparison instruction (red) the program makes the same decision then the program execution must reach the same leaf.

**Lemma 4.2 (Snapping)** Let  $(a, b) \in [0, 1]^n \times \mathbb{Z}^m$  be the input of an algorithm  $A$  with comparison set size  $P(n, m)$ . Let  $x$  be a perturbation chosen uniformly at random in  $[-\delta/2, \delta/2]^n$  and let  $a_x = a + x$  be a perturbed instance of the input. For all  $\varepsilon \in [0, 1]$ , if  $a_x$  is snapped to a grid of width

$$\omega \leq \frac{\varepsilon \delta}{3^d(d+1)! \cdot \Delta \cdot 4P(n, m)},$$

then the algorithm  $A$  makes for  $a_x$  and  $a'$  the same decision at each comparison instruction with probability at least  $1 - \varepsilon$ .

**Proof** By  $E_i$  for  $i \in [P(n, m)]$ , we denote the event for the  $i$ 'th real-valued comparison expression in  $P(n, m)$  the input  $a_x$  and  $a'$  get a different outcome. Lemma 4.1 upper bounds the probability of  $E_i$  occurring by:

$$\Pr(E_i) \leq \frac{4\omega\Delta 3^d(d+1)!}{\delta}.$$

The probability that  $a_x$  and  $a'$  are not equivalent is equal to the probability that at least one event  $E_i$  occurs. In other words:

$$\Pr(g_x \text{ and } g' \text{ not equivalent}) = \Pr\left(\bigcup_{i=1}^{P(n, m)} E_i\right) \leq P(n, m) \cdot 4\omega\Delta 3^d(d+1)/\delta.$$

Finally,  $\Pr(g_x \text{ and } g' \text{ not equivalent}) < \varepsilon$  is implied by  $P(n, m) \cdot \frac{4\omega\Delta 3^d(d+1)}{\delta} < \varepsilon$ . ■

The above lemma bounds the probability that an algorithm gives a different answer after the input is snapped. To bound the *expected* input precision of  $A$ , we apply a folklore lemma about swapping the order of integration [190]:

**Lemma 4.3** *Given a function  $f : \Omega \rightarrow \{1, \dots, b\}$  it must be that*

$$\mathbb{E}[f] = \sum_{z=1}^b z \Pr(f(x) = z) = \sum_{z=1}^b \Pr(f(x) \geq z).$$

Using Lemma 4.3, we focus on the expected value of the input precision,  $\text{bit}_{IN}$ :

$$\mathbb{E}(\text{bit}_{IN}(a_x, A)) = \sum_{k=1}^{\infty} k \Pr(\text{bit}_{IN}(a_x, A) = k) = \sum_{k=1}^{\infty} \Pr(\text{bit}_{IN}(a_x, A) \geq k).$$

We split the sum at a splitting point  $l$ :

$$\mathbb{E}(\text{bit}_{IN}(a_x, A)) = \sum_{k=1}^l \Pr(\text{bit}_{IN}(a_x, A) \geq k) + \sum_{k=l+1}^{\infty} \Pr(\text{bit}_{IN}(a_x, A) \geq k).$$

Now we note that any probability is at most 1 therefore the left sum is at most  $l$ . Through applying Lemma 4.2 we obtain:

$$\Pr(\text{bit}_{IN}(a_x, A) \geq k) = \Pr(\text{GridWidth}(a_x) \geq 2^{-k}) \leq 2^{-k} \left( \frac{3^d(d+1)! \Delta P(n, m)}{\delta} \right),$$

which in turn implies:

$$\mathbb{E}(\text{bit}_{IN}(a_x, A)) \leq \sum_{k=1}^l 1 + 4 \left( \frac{3^d(d+1)! \Delta P(n, m)}{\delta} \right) \sum_{k=l+1}^{\infty} 2^{-k}.$$

Observe that  $\sum_{k=l+1}^{\infty} 2^{-k} = 2^{-l}$ . So if we choose  $l = \lceil \log \frac{3^d(d+1)! \Delta P(n, m)}{\delta} \rceil + 2$  we get:

$$\begin{aligned} \mathbb{E}(\text{bit}_{IN}(a_x, A)) &\leq l + 4 \left( \frac{3^d(d+1)! \Delta P(n, m)}{\delta} \right) 2^{-l} \\ &\leq \left\lceil \log \frac{3^d(d+1)! \Delta P(n, m)}{\delta} \right\rceil + 4 + \\ &\quad + \left( \frac{3^d(d+1)! \Delta P(n, m)}{\delta} \right) \left( \frac{\delta}{3^d(d+1)! \Delta P(n, m)} \right) \\ &= \left\lceil \log \frac{3^d(d+1)! \Delta P(n, m)}{\delta} \right\rceil + 5 = O \left( d \log \frac{d \Delta P(n, m)}{\delta} \right) \end{aligned}$$

This holds for arbitrary  $a$  with  $a_x = a + x$ . Hence we have an upper bound on the smoothed input precision required by a polynomial-time algorithm:

**Theorem 4.3** *Let  $A$  be a real RAM algorithm with arithmetic degree  $\Delta$  and algebraic dimension  $d$  with input  $(a, b) \in [0, 1]^n \times \mathbb{Z}^m$  and comparison set size  $P(n, m) = n^{O(1)}$ . Under perturbations of  $a \in [0, 1]^n$  by  $x \in [-\frac{\delta}{2}, \frac{\delta}{2}]^n$  chosen uniformly at random,  $A$  has  $O(d \log \frac{d \Delta n}{\delta})$  smoothed input precision.*

We combine Lemma 3.1 together with linearity of expectation to conclude:

**Theorem 4.1** *Let  $A$  be a real RAM algorithm with arithmetic degree  $\Delta$ , algebraic dimension  $d$  and extra algebraic dimension  $s = 0$  with input  $(a, b) \in [0, 1]^n \times \mathbb{Z}^m$  and comparison set size  $P(n, m) = n^{O(1)}$ . Under perturbations of  $a \in [0, 1]^n$  by  $x \in [-\frac{\delta}{2}, \frac{\delta}{2}]^n$  chosen uniformly at random,  $A$  has a smooth bit precision of at most:*

$$\text{bit}_{\text{smooth}}(\delta, n, A) = O\left(d \log \frac{d\Delta n}{\delta} \Delta^2 \log d\right).$$

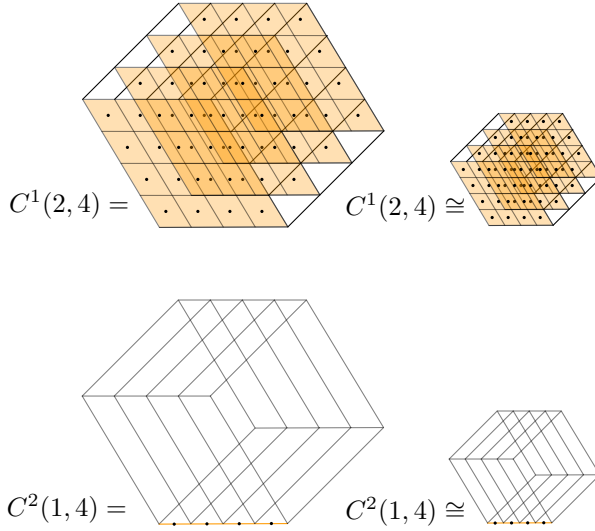
## 4.2 Smoothed analysis of the real RAM with roots

We extend our results to include our version of the real RAM that does allow use of the square root operator. Recall that for any radical expression  $f$ , Lemma 3.2 provides a recursive procedure to extract the extra algebraic dimension  $s$ . We use this parametrization to prove the following:

**Theorem 4.4** *Let  $A$  be a real RAM algorithm. Let  $A$  have arithmetic degree  $\Delta$ , algebraic dimension  $d$  and extra algebraic dimension  $s$  with input  $(a, b) \in [0, 1]^n \times \mathbb{Z}^m$  and comparison set size  $P(n, m) = n^{O(1)}$ . Under perturbations of  $a \in [0, 1]^n$  by  $x \in [-\frac{\delta}{2}, \frac{\delta}{2}]^n$  chosen uniformly at random,  $A$  has a smooth bit precision of at most:*

$$\text{bit}_{\text{smooth}}(\delta, n, A) = O\left((d + s) \log \frac{\Delta n}{\delta} \Delta^2 \log d\right).$$

**Preliminaries for proving Theorem 4.4.** We generalise our notation of a grid, to go from hypercubes to prisms. Specifically given  $z \in \mathbb{Z}^d$  and  $s \in \mathbb{N}$ , we define the *unit prism* rooted at  $z$  denoted by  $C_z^s \subset \mathbb{R}^{d+s}$  as  $C_z^s = \{x \in \mathbb{R}^{d+s} \mid z_i \leq x_i < z_i + 1, \forall i = 1, \dots, d\}$  (intuitively, we define a  $d$ -dimensional unit square rooted at  $z$ , extended into infinity in each of the  $s$  extra dimensions). We subsequently define  $C^s(d, k) := \{C_z^s \mid z \in \mathbb{Z}^d \cap [0, k)^d\}$  as a grid of prisms (Figure 4.8). Observe that for  $s = 0$ , this definition follows our definition for hypercubes. We denote by  $C^s(d, k)$  the collection of all  $d + s$ -dimensional unit prisms:  $C^s(d, k) := \{C_z^s \subset \mathbb{R}^{d+s} : z \in \mathbb{Z}^d \cap [0, k)^d\}$ . Let  $C$  be a  $(d+s)$ -dimensional prism, partitioned by  $(d+s)$ -dimensional prisms of equal width. We say  $C$  is *equivalent* to  $C(d, k)$ , denoted by  $C \cong C(d, k)$ , if there exists an affine transformation  $\tau$  of  $C$  such that there is a one-to-one correspondence between prisms in  $\tau(C)$  and  $C(d, k)$  where corresponding prisms coincide.



**Figure 4.8** The prism  $C^1(2, 4)$  is a two-dimensional grid (shown in yellow) extended into one additional dimension. We can scale or even translate or rotate the grid to obtain different embeddings of  $C^1(2, 4)$ . Similarly, the prism  $C^2(1, 4)$  is a one-dimensional grid, extended into two dimensions. Scaling and translating creates a different embedding of  $C^1(2, 4)$ .

**Proving Theorem 4.4.** Given an expression  $f$  and a suitable equivalent set of polynomials  $p, q_1, \dots, q_s$ , we follow the same strategy for when we analysed the real RAM without square roots. First, we show the following algebraic lemma, whose proof has been relegated to Section 4.B to not distract from the main story.

**Theorem 4.5** Let  $q_0, \dots, q_s$  be polynomials with maximum degree  $\Delta$  and  $k \geq 2\Delta + 1$ , such that the variety  $V(q_0, \dots, q_s)$  is  $(d - 1)$ -dimensional. Then the variety  $V(q_0, \dots, q_s)$  intersects at most  $k^{d-1} (2\Delta)^{s+1} 3^d (d + 1)!$  unit prisms in  $C^s(d, k)$ .

We subsequently first consider a single comparison operation:

**Lemma 4.4 (Register-Snapping)** Let  $R$  be a register, which holds some expression  $f$  with maximum degree  $\Delta$ , algebraic dimension  $d$  and extra algebraic dimension  $s$ . Let  $a \in \mathbb{R}^d$  be fixed and  $x \in \Omega = [-\delta/2, \delta/2]^d$  chosen uniformly at random. Assume  $(a + x)$  is snapped to a point  $a' \in \omega\mathbb{Z}^d \cap [0, 1]^d$ . Then  $\text{sign}(f(a + x)) \neq \text{sign}(f(a'))$  with probability at most:

$$\frac{\omega(2\Delta)^{s+1} 3^d (d + 1)!}{\delta}.$$

**Proof** This proof follows a similar structure to the proof of Lemma 4.1.

As  $f$  is a continuous function, we can upper bound the probability of  $f(a') \neq f(a)$ , by counting the number of Voronoi cells  $C \in \Gamma_\omega$ , which are intersected by the variety  $V(f)$ . As  $f$  is not a polynomial, we cannot immediately employ any of our theorems that upper bounds the intersections of a variety with a grid. Instead, we consider the polynomials  $p, q_1, \dots, q_s$  which exist by the assumption that the extra algebraic dimension is  $s$ .

We denote by  $\pi : \mathbb{R}^{d+s} \rightarrow \mathbb{R}^d$ ,  $(x_1, \dots, x_d, y_1, \dots, y_s) \mapsto (x_1, \dots, x_d)$  the orthogonal projection on the first  $d$  coordinates. From the definition of  $(p, q_1, \dots, q_s)$  it follows that the orthogonal projection of their variety is equal to  $V(f)$ , or in other words:  $V(f) = \pi(V(p, q_1, \dots, q_s))$ . We want to upper bound the number of  $d$ -dimensional Voronoi cells that are intersected by  $V(f)$ , and we obtain this by providing an upper bound on the number of  $(d+s)$ -dimensional Voronoi cells that are intersected by the higher-dimensional variety  $V(p, q_1, \dots, q_s)$ . Given a hypercube  $C \in \Gamma_\omega \subset \mathbb{R}^d$ , we define the prism  $\tilde{C} = \{(x, y) \in \mathbb{R}^{d+s} : x \in C\}$ . Per construction of  $(p, q_1, \dots, q_s)$ , it holds that:

$$V(f) \cap C = \emptyset \Leftrightarrow V(p, q_1, \dots, q_s) \cap \tilde{C}.$$

Analogous to Lemma 4.1, we consider the  $d$ -dimensional hypercube partitioned by  $\Gamma_\omega(a) \cong C^0(d, \lfloor \frac{\delta}{\omega} \rfloor)$ . The grid  $\Gamma_\omega(a)$  in our  $(d+s)$ -dimensional space becomes a collection of prisms equivalent to  $C^s(d, \lfloor \frac{\delta}{\omega} \rfloor)$ . This now allows us to apply Theorem 4.5 and bound the number of prisms that are intersected by  $V(p, q_1, \dots, q_s)$  by:

$$k^{d-1}(2\Delta)^{s+1}3^d(d+1)!.$$

Via our transformation, this implies that the number of cells in  $\Gamma_\omega(a)$  that are intersected by  $V(f)$  is upper bounded by  $k^{d-1}(2\Delta)^{s+1}3^d(d+1)!$ . Just as in Lemma 4.1, the perimeter of  $\Gamma_\omega(a)$  contains at most  $2dk^{d-1}$  cells. There are  $k^d$  hypercubes in  $\Gamma_\omega(a)$  and it follows that:

$$\begin{aligned} Pr(\text{sign}(f(a+x)) \neq \text{sign}(f(a'))) &\leq \frac{1}{k^d} (k^{d-1}(2\Delta)^{s+1}3^d(d+1)! + 2dk^{d-1}) \\ &\leq \frac{1}{k} 2(2\Delta)^{s+1}3^d(d+1)! \\ &\leq \frac{1}{\delta} \omega 2(2\Delta)^{s+1}3^d(d+1)! \end{aligned}$$

This finishes the proof of Lemma 4.4. ■

Finally, we finish the argument to prove Theorem 4.4. Note that the statement of Lemma 4.4 is identical to Lemma 4.1 of the previous section, with the exception that  $\Delta$  is replaced by  $(2\Delta)^{s+1}$ . The remainder of the proof to Theorem 4.4 is henceforth identical to the proof of Theorem 4.1, except that at each step of the analysis,  $\Delta$  is replaced by  $(2\Delta)^{s+1}$ .

### 4.3 Concluding remarks

At the beginning of this chapter, we explained that it is important that the computations done by a real RAM are robust: that algorithms can be executed with (logarithmically) bounded bit precision. Prior works that discuss the robustness of geometric computations, propose alterations to the geometric predicates and assumptions to make them robust. We propose to use smoothed analysis, to reason that our classical geometric algorithms are already robust. We want to briefly discuss how our result can be applied, its impact, and the limitations of our result.

Theorem 4.1 upper bounds the bit precision needed to correctly execute algorithms with arithmetic degree  $\Delta$  and algebraic dimension  $d$ , provided that the real RAM program does not use the square root operator. We note that the standard operations and predicates used in computational geometry all have constant arithmetic degree and algebraic dimension (in-circle testing, orientation testing and testing if two constant-complexity objects intersect), and a trivially polynomial decision set size  $P(n, m)$ . Thus, we immediately conclude the following:

**Corollary 4.1** *The following classical algorithmic problems have corresponding algorithms (see [184]) that have a smooth bit precision of  $O(\log(n/\delta))$ :*

- *Computing convex hulls.*
- *Classical inclusion problems.*
- *Computing the intersection between straight-line geometric objects.*
- *Solving closest-point problems in constant dimension.*
- *Range searching problems.*
- *Computing the shortest path in a simple polygon (not its length).*

In fact, with the exception of algorithmic problems, that explicitly compare, minimize or compute the length of a path with  $n$  vertices, it is hard to find examples in computational geometry for which the algorithmic solutions do not have constant arithmetic degree and algebraic dimension. Recall that a smooth bit precision of  $O(\log(n/\delta))$  is considered to be ‘logarithmic in practice’ hence this theorem shows that classical computational geometric algorithms are, in practice, robust. Note that some of these problems include using the Euclidean distance, even though we prohibit the usage of the square root operator. However for each of these algorithmic results, there is a constant-complexity predicate that verifies if the distance is below some desired threshold without making use of the square root (e.g., we can check if two points have distance at most 5, by checking if their squared distance is at most 25).

Whenever a problem statement requires predicates that contain more than a constant number of square roots, we can apply Theorem 4.4 as follows. Note that this analysis confirms the conjecture by Yao and Sharma [191, Ch. 45] that computing the sum of square roots can be done in practice with linear bit precision:

**Corollary 4.2** *The following algorithmic problems have corresponding algorithms that have a smooth bit precision of  $O(n \log(n/\delta))$ :*

- *Comparing a sum of square roots to a real or integer value.*
- *Comparing high-dimensional ( $d > \log n$ ) distances between points.*
- *Solving high-dimensional closest-point problems.*
- *Computing the shortest path in a polygonal domain.*

**Limitations.** We want to point out that even though our results are broadly applicable, they are not always applicable. Firstly, not all algorithms meet the conditions of Theorems 4.1 and 4.4. For example geometric algorithms that handle input of arbitrary dimension, or very high dimension. In both cases, we still get some upper bound on the bit precision, but it might be an overestimation. In addition, in the introduction we provided some examples of algorithmic problems where the comparison set size  $P(n, m)$  is not polynomially bounded. Second, perturbing the input may not always be sensible. This does not mean that our theorems do not apply in a mathematical sense, but rather that in reality the result is not usable. In particular this is the case whenever the application relies on degeneracies in one way or another. A concrete example is an algorithm to compute if any three points in a planar point set  $P$  are collinear. After a small random real-valued perturbation, none of them are. As a rule of thumb, our results are most sensible whenever the problem statement assumes that the input is a point set that lies in general position.

Having presented the concluding remarks, we present the algebraic arguments that the results in this chapter rely on.

## 4.A Varieties hitting cubes

This section is dedicated to the algebraic proof needed to upper bound the number of unit hypercubes that a  $d$ -variate polynomial  $p$  of bounded degree  $\Delta$  can intersect in  $C(d, k)$ . Before we formally prove Theorem 4.2, we present the necessary definitions and recall well-known and folklore algebraic lemmas.

Following Cox et al. [56] we define a  $d$ -variate polynomial  $p$  in  $x_1, \dots, x_d$  with real coefficients as a finite linear combination of monomials with real coefficients. Let  $p \in \mathbb{R}[x_1, \dots, x_d]$  denote the set of such polynomials.

We denote by  $V(p) := \{x \in \mathbb{R}^d : p(x) = 0\}$  the *variety* of  $p$ . For any subset  $S \subset \mathbb{R}^d$ , we say that  $p$  intersects  $S$  if  $S \cap V(p) \neq \emptyset$ . Given a set of polynomials  $p_1, \dots, p_k$ , we denote their variety as  $V(p_1, \dots, p_k) = \bigcap_{i=1, \dots, k} V(p_i)$ . For any expression  $f$  which defines a function, we also use the notation  $V(f) = \{x : f(x) = 0\}$ , although it is not necessarily a variety. We say  $f$  intersects a set  $S$ , if  $V(f) \cap S \neq \emptyset$ . We need the notion of the *dimension* of a variety. We assume that most readers have some intuitive understanding, which is sufficient to follow the arguments. It is out of scope to define the concept of dimension in this thesis, so we refer to the book by Basu, Pollack and Roy [16, Chapter 5]. Specifically, Lemma 4.5 has to be taken for granted. Given a polynomial  $p \in \mathbb{R}[x_1, \dots, x_d]$ , the linear polynomial  $\ell \in \mathbb{R}[x_1, \dots, x_d]$  is a factor of  $p$ , if there exists some  $q \in \mathbb{R}[x_1, \dots, x_d]$  such that  $\ell \cdot q = p$ .

Our proof gives a slightly stronger, but more complicated upper bound. The proof idea is to consider the intersection between a hypercube  $C_z \in C(d, k)$  and the polynomial  $p$ . Then either a connected component of  $V(p)$  is contained in  $C_z$  or  $V(p)$  must intersect one of the  $(d - 1)$ -dimensional facets of  $C_z$ . In order to estimate how often the first situation can occur, we use a famous theorem by Oleinik-Petrovski/Thom/Milnor, in a slightly weaker form. See Basu, Pollack and Roy [16, Chapter 7] for historic remarks and related results.

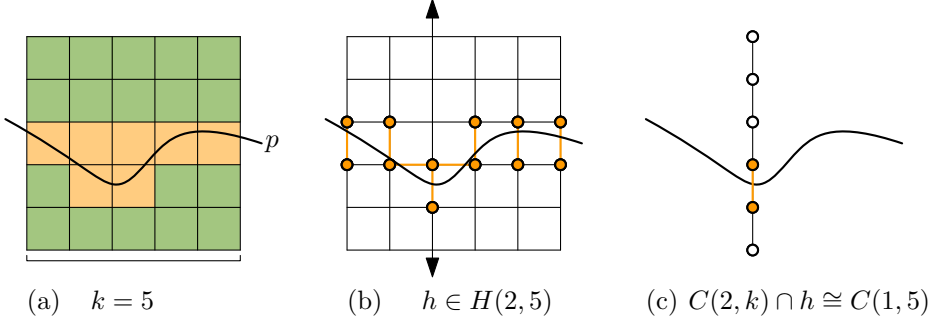
**Theorem 4.6 (Milnor [16])** *Given a set of  $d$ -variate polynomials  $q_0, \dots, q_s$  with maximal total degree  $\Delta$ . Then the variety  $V(q_0, \dots, q_s)$  has at most  $(2\Delta)^d$  connected components.*

We also need the following folklore lemma. For more background on polynomials, see the book from Cox, Little, O'Shea [57]. Specifically Hilbert's Nullstellensatz, which can be found as Theorem 4.77 in the book by Basu, Pollack and Roy [16] is important.

**Lemma 4.5 (folklore)** *Let  $p \in \mathbb{R}[x_1, \dots, x_d]$  be a  $d$ -variate polynomial and  $H = \{x \in \mathbb{R}^d : \ell(x) = 0\}$  a  $(d - 1)$ -dimensional hyperplane. Then  $V(p) \cap H$  is the variety of a  $(d - 1)$ -variate polynomial or  $\ell$  is a polynomial factor of  $p$ .*

In our applications,  $\ell$  will be of the form  $x_i = a$ , for some constant  $a$ . We are now ready to prove Theorem 4.2:

**Theorem 4.2** Let  $p \neq 0$  be a  $d$ -variate polynomial with maximum degree  $\Delta$  and let  $k = 2\Delta + 2$ . Then the variety  $V(P)$  of the polynomial  $p \in \mathbb{Z}[x_1, \dots, x_d]$  intersects at most  $k^{d-1} \Delta 3^d (d+1)!$  unit hypercubes in  $C(d, k)$ .



**Figure 4.9** (a) The polynomial  $p$  intersects some fraction of the hypercubes of  $C(2, k)$ . (b) The set  $H(2, k)$  of axis-parallel lines and the 1-dimensional facets of  $C(2, k)$  that are intersected by  $p$ . (c) The intersection of  $C(2, k)$  with a line  $h$  gives  $C(1, 5)$ .

**Proof** If  $p$  has a linear factor  $\ell$ , we decompose  $p$  into  $p = q \cdot \ell$ . Note that  $\ell$  intersects at most  $O(k^{d-1}d)$  hypercubes in  $C(d, k)$ . The maximum degree of  $q$  drops by one and we recurse until we have no more remaining factors.

Henceforth, we assume  $p$  has no linear factors. Denote by  $f(d)$  the maximum number of unit hypercubes of  $C(d, k)$  that can be intersected by a  $d$ -variate polynomial  $p \neq 0$  with maximal total degree  $\Delta$ . We first show that:

$$f(1) \leq \Delta. \quad (4.1)$$

Then we will show in a similar manner for every  $d, k$  and  $\Delta$ :

$$f(d) \leq 2f(d-1) \cdot d(k+1) + (2\Delta)^d. \quad (4.2)$$

Solving the recursion then gives the desired upper bound as follows: first, we show by induction that Equation 4.1 and Equation 4.2 imply  $f(d) \leq (k+1)^{d-1} \Delta 2^d (d+1)!$ . Equation 4.1 is the induction basis. Using  $2\Delta \leq k$ , the induction step is then:

$$\begin{aligned}
f(d) &\leq 2f(d-1) \cdot d(k+1) + (2\Delta)^d \\
&\leq 2f(d-1) \cdot d(k+1) + (2\Delta)k^{d-1} \\
&\leq 2(k+1)^{d-2} \Delta 2^{d-1}(d)! \cdot d(k+1) + (2\Delta)k^{d-1} \\
&= (k+1)^{d-1} (2\Delta) 2^{d-1}(d)! \cdot d + (2\Delta)k^{d-1} \\
&= (k+1)^{d-1} (2\Delta) (2^{d-1}(d)! \cdot d + 1) \\
&\leq (k+1)^{d-1} \Delta 2^d(d)! \cdot (d+1) \\
&= (k+1)^{d-1} \Delta 2^d(d+1)!
\end{aligned}$$

Using  $k \geq 2\Delta + 2 \geq 3$ , we can deduce that

$$(k+1)^{d-1} = \frac{(k+1)^{d-1}}{k^{d-1}} \cdot k^{d-1} \leq (1.5)^d k^{d-1}$$

This implies that  $f(d) \leq k^{d-1} \Delta 3^d(d+1)!$ . It remains to show the validity of Equation 4.1 and 4.2. If  $p$  is a univariate polynomial of degree  $\Delta$  then its variety  $V(p)$  is a set of at most  $\Delta$  points and therefore  $p$  can intersect at most  $\Delta$  disjoint unit intervals. This implies Equation 4.1.

To show the correctness of Equation 4.2, we refer to Figure 4.9. We denote by  $H(d, k)$  the  $d(k+1)$  axis-parallel,  $(d-1)$ -dimensional hyperplanes with integer coordinates that intersect the hypercube  $C(d, k)$ . For a given integer  $i$  and value  $\alpha$ , we define the hyperplane  $h(i, \alpha) = \{x \in \mathbb{R}^d : x_i = \alpha\}$ . Hence the set  $H(d, k)$  can be formally defined as  $H(d, k) := \{h(i, \alpha) : (i, \alpha) \in [d] \times [k]\}$ .

As stated at the beginning, we can assume that  $p$  has no linear factors and we apply Lemma 4.5 to  $p$  and each  $h \in H(d, k)$ . For all hypercubes in  $C(d, k)$ , all facets of such a hypercube are contained inside a  $(d-1)$ -dimensional hyperplane  $h \in H(d, k)$ . By Milnor's theorem, there are at most  $(2\Delta)^d$  cubes in  $C(d, k)$  which are intersected by  $p$  but whose boundary is not intersected by  $p$ . For any other hypercube in  $C(d, k)$  that is intersected by  $p$ , the polynomial must intersect a  $(d-1)$ -dimensional facet.

Consider one of the  $d(k+1)$  hyperplanes  $h \in H(d, k)$ . The intersection between  $h$  and  $C(d, k)$  is equivalent to  $C(d-1, k)$ . Furthermore, by Lemma 4.5, the variety  $V(p)$  restricted to  $h$  is the variety of a  $(d-1)$ -dimensional polynomial. Thus by definition, we know that  $p$  intersects at most  $f(d-1)$   $(d-1)$ -dimensional hypercubes in the intersection between  $h$  and  $C(d, k)$ . Each of these intersected  $(d-1)$ -dimensional hypercubes can coincide with a facet of at most two hypercubes in  $C(d, k)$ . It follows that  $f(d)$  is upper bound by  $(2\Delta)^d + 2 \cdot f(d-1) \cdot k(d+1)$ . This shows Equation 4.2 and finishes the proof. ■

## 4.B Varieties hitting prisms

The proof of Theorem 4.5 follows the same line of argument as Theorem 4.2, but with different constants. Similarly to Lemma 4.5, for polynomials, we have to recall a folklore lemma for varieties.

**Lemma 4.6 (folklore)** *Let  $q_0, \dots, q_s \in \mathbb{R}[x_1, \dots, x_d, y_1, \dots, y_s]$  be polynomials with maximum degree  $\Delta$ . Furthermore, the variety  $V = V(q_0, \dots, q_s)$  is  $(d-1)$ -dimensional. Let  $H = \{x \in \mathbb{R}^d : \ell(x) = 0\}$  be a  $(d+s-1)$ -dimensional hyperplane. Then either  $\ell$  is a factor of all  $q_0, \dots, q_s$  or  $V \cap H$  is a  $(d-2)$ -dimensional variety.*

**Theorem 4.5** *Let  $q_0, \dots, q_s$  be polynomials with maximum degree  $\Delta$  and  $k \geq 2\Delta + 1$ , such that the variety  $V(q_0, \dots, q_s)$  is  $(d-1)$ -dimensional. Then the variety  $V(q_0, \dots, q_s)$  intersects at most  $k^{d-1} (2\Delta)^{s+1} 3^d (d+1)!$  unit prisms in  $C^s(d, k)$ .*

**Proof** We again assume that the involved polynomials have no linear factor. Let us define  $f(d)$  as the maximum number of unit prisms of  $C^s(d, k)$  that can be intersected by  $V(q_0, \dots, q_s)$  with maximum degree  $\Delta$ . We will first show that

$$f(1) \leq (2\Delta)^{s+1}. \quad (4.3)$$

Then we will show in a similar manner for every  $d, s$  and  $\Delta$  holds that

$$f(d) \leq 2f(d-1) \cdot d(k+1) + (2\Delta)^{d+s}. \quad (4.4)$$

Solving the recursion then gives the upper bound of the theorem in three steps:

1. First, we show by induction that Equation 4.3 and Equation 4.4 imply  $f(d) \leq (k+1)^{d-1} (2\Delta)^{s+1} 2^d (d+1)!$ .
2. Equation 4.3 establishes the induction basis. Note that  $2\Delta \leq k+1$  implies  $(2\Delta)^{d-1} \leq (k+1)^{d-1}$ . (See Assumption of the Theorem.)
3. The induction step goes as follows:

$$\begin{aligned}
 f(d) &\leq 2d(k+1) \cdot f(d-1) + (2\Delta)^{s+d} \\
 &\leq 2d(k+1) \cdot f(d-1) + (2\Delta)^{s+1} k^{d-1} \\
 &\leq 2d(k+1) \cdot [(k+1)^{d-2} (2\Delta)^{s+1} 2^{d-1} (d)!] + (2\Delta)^{s+1} k^{d-1} \\
 &= (k+1)^{d-1} (2\Delta)^{s+1} 2^{d-1} (d)! \cdot d + (2\Delta)^{s+1} k^{d-1} \\
 &= (k+1)^{d-1} (2\Delta)^{s+1} \cdot (2^{d-1} (d)! \cdot d + 1) \\
 &\leq (k+1)^{d-1} (2\Delta)^{s+1} 2^d (d)! \cdot (d+1) \\
 &= (k+1)^{d-1} (2\Delta)^{s+1} 2^d (d+1)!
 \end{aligned}$$

We simplify this expression slightly, using  $k \geq 2\Delta + 2 \geq 3$ . We deduce that

$$(k+1)^{d-1} = \frac{(k+1)^{d-1}}{k^{d-1}} \cdot k^{d-1} \leq (1.5)^d k^{d-1}$$

This implies that  $f(d) \leq k^{d-1}(2\Delta)^{s+1}3^d(d+1)!$ . It remains to show the validity of Equation 4.3 and 4.4.

If  $V = V(q_0, \dots, q_s) \subset \mathbb{R}^{1+s}$  is a 0-dimensional variety of degree  $\Delta$  then it is a set of at most  $t = (2\Delta)^{s+1}$  points, by Theorem 4.6. Therefore  $V$  can intersect at most  $t$  disjoint unit prisms and this implies Equation 4.3.

Now we show correctness of Equation 4.4. We denote  $V = V(q_0, \dots, q_s) \subset \mathbb{R}^{d+s}$ . Furthermore, we note that there are  $d(k+1)$  axis-parallel  $(d-1)$ -dimensional hyperplanes with integer coordinates that intersect  $C^s(d, k)$ . We denote them by  $H(d, s, k)$ . Formally, we define the hyperplane  $h(i, a) = \{x \in \mathbb{R}^{d+s} : x_i = a\}$  and the set:

$$H(d, s, k) = \{h(i, a) : (i, a) \in [d] \times [k]\}.$$

(This is almost identical to the case with hypercubes.) For all prisms in  $C^s(d, k)$ , all facets of such a prism are contained inside a  $(d-1)$ -dimensional hyperplane  $h \in H(d, s, k)$ . By Milnor's theorem, there are at most  $(2\Delta)^{s+k}$  prisms in  $C^s(d, k)$  which are intersected by  $V$  but whose boundary is not intersected by  $V$ . For any other prism in  $C^s(d, k)$  that is intersected by  $V$ , the variety must intersect a  $(d-1)$ -dimensional facet of that prism.

Consider one of the  $d(k+1)$  hyperplanes  $h \in H(d, s, k)$ . The set  $I = h \cap C^s(d, k)$  is, up to coordinate transformations, equivalent to  $C(d-1, k)$ . Thus by definition, we know that  $V$  intersects at most  $f(d-1)$  prisms in  $I$ . (In particular,  $h \not\subseteq V$ , by the comment at the beginning of the proof.) Each of these of these  $(d-1)$ -dimensional prisms can coincide with a facet of at most two prisms in  $C(d, k)$ . It follows that  $f(d)$  is upper bound by  $(2\Delta)^{s+d} + 2 \cdot f(d-1) \cdot k(d+1)$ . This shows Equation 4.4. ■

PART III

# Structuring data



## Chapter Five

# Visibility between moving entities

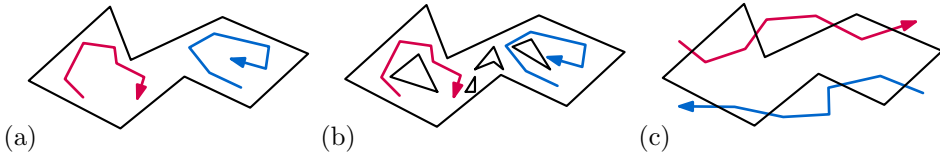
In this chapter we consider two entities that each follow a two-dimensional trajectory, with (possibly different) constant speed. Their trajectories lie in an environment with obstacles that block visibility. We study how to structure the domain of the trajectories to solve what we call *trajectory visibility testing*:

Can we efficiently compute if two moving entities, at any time, see each other?

## 5.1 Introduction

This question combines two key concepts from computational geometry, namely trajectory analysis and visibility queries. Both are well-studied topics within computational geometry and we briefly review some literature.

**Trajectory analysis.** A recent increase in the availability of low-cost, internet connected GPS tracking devices has driven considerable interest in spatio-temporal data (commonly called *trajectories*) across fields including Geographic Information Science, databases, and computational geometry. Problems studied recently in computational geometry include detecting and describing flocks [20, 137], detecting hotspots, clustering and categorising trajectories, map construction and others [36, 99, 138]. Formally, a discrete trajectory is a sequence of time-stamped locations in the plane (or more generally in  $\mathbb{R}^d$ ) which models the movement of an entity. We assume that the entity moves linearly between the time-stamped locations with constant speed. Trajectory data is obtained by tracking the movements of animals [31, 111], hurricanes [188], traffic [141], or other moving entities [73] over time. For a more extensive overview of trajectory analysis we refer the reader to the survey by Gudmundsson et al. [104].



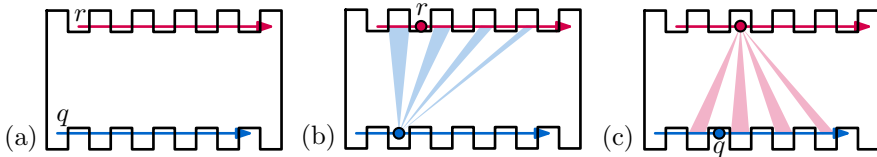
**Figure 5.1** The three different variations of the problem studied in this chapter. (a) Two trajectories inside a simple polygon. (b) Two trajectories in a polygonal domain. (c) Two trajectories intersecting a simple polygon. Our approach for the middle variant is independent of whether the trajectories intersect the domain.

**Visibility.** Two points, amidst a number of obstacles, are mutually *visible* if the line segment between them does not intersect any obstacle. Visibility is one of the most studied topics in computational geometry [163, 200] and in adjacent fields such as computer graphics [77], GIScience [66], and robotics [163]. Within computational geometry, Gosh and Giswani compiled a survey of unsolved problems in this area [100]. Visibility problems central in computational geometry include *ray shooting* [60, 52], guarding [55, 92] and visibility graph recognition [48]. For more information about visibility problems, refer to O’Rourke’s book [169, Ch. 8], and the surveys by Durant [77] and Gosh [100].

**Trajectory visibility.** In this chapter we study the following fundamental question, which we refer to as *trajectory visibility testing*. Given a simple polygon (or a polygonal domain)  $P$  and the trajectories  $q, r$  of two moving entities, is there a time  $t$  at which the entities see each other? With slight abuse of notation, we will denote both the entities and their trajectories as  $q$  and  $r$ . We assume that  $q$  and  $r$  move linearly with constant (but possibly different) speeds between trajectory vertices, and cannot see through the edges or vertices of  $P$ . We distinguish several variants (Figure 5.1) depending on whether  $P$  is a simple polygon or a polygonal domain, and whether the trajectories are allowed to intersect  $P$  (e.g. animals moving through foliage) or not (e.g. pedestrians moving amidst buildings). We further consider the same variants in the simpler scenario in which one of the entities is a point (e.g. a stationary guard and a moving intruder). Note that we are interested only whether there *exists* a time at which the two entities see each other. This implies we can temporally decompose the problem: the answer is *no* if and only if the answer is no between all two consecutive time stamps. When considering this question, two fundamentally different approaches come to mind. On the one hand, when the number  $\tau$  of trajectory vertices is small compared to the number  $n$  of polygon vertices, the best approach may be to simply solve the problem for each time interval separately. On the other hand, when  $\tau$  is large compared to  $n$ , it may be more efficient to spend some time on preprocessing  $P$  first, if this allows us to spend less time per trajectory edge. We therefore distinguish between the *algorithmic* question and the *data structure* question. Our results are discussed below and summarized in Table 5.1.

**Related work on visibility for moving entities.** There is a vast amount of research on both trajectories and visibility, but surprisingly not much previous work exists on their combination. One possible explanation for this, is that the already developed tools for visibility and trajectory analysis cannot be combined in a straightforward manner. Consider two trajectories  $q$  and  $r$  within a simple polygon  $P$ : existing visibility tools allow us to easily check if there are subtrajectories of  $q$  and  $r$  which are mutually visible (that is, a subtrajectory  $q$  is visible from  $r$  if for each point on  $q$ , there is a point on  $r$  that sees  $q$ ). However, the two moving entities see each other only if there is a *time* at which the location of the two entities are mutually visible. There can be quadratically many pairs of subtrajectories which are mutually visible; yet, it could be that the two entities are never simultaneously within such a pair (Figure 5.2). To determine visibility between moving entities, one needs to incorporate the concept of time into pre-existing tools for visibility. Early results in this direction include results by Bern et al. [23] and Mulmuley [165], who study maintaining the visibility polygon of a point that moves over a straight path. Aronov et al. [13] demonstrate a kinetic data structure that tracks the visibility polygon of a moving query point  $q$ . The most recent result on visibility and motion is by Diez et al. [70] who show how to maintain the shortest path between two moving entities using a kinetic data structure. Here  $q$  and  $r$  are mutually visible if and only if their shortest path is a line segment.

**From event based modelling to algebraic range searching.** The above attempts at answering visibility testing queries model the passage of time using a sequence of *events*. However, as  $q$  and  $r$  traverse their trajectories one may encounter a linear number of events where the visibility-status of either entity changes. This prohibits a straightforward application of these techniques to create a data structure that can answer if there is visibility between trajectories in sublinear time. In this chapter, we diverge from the classical event-based approach and model the problem in an algebraic way. Such an algebraic reformulation is not rare in computational geometry: for example, disk-containment queries are solved by reformulating them into higher-dimensional halfspace range queries. Yet we are unaware of any algebraic approaches used for visibility testing. The challenge with such an algebraic approach is that the more complicated the algebraic expression is, the worse the complexity of the runtime becomes (and with it, the complexity of the exposition of the argument).



**Figure 5.2** Two trajectories  $q$  and  $r$ . If both entities move at unit speed, then at each time either (b) the blue entity sees a linear number of red subtrajectories or (c) the red entity a linear number of blue, but they are never mutually visible.

**Summary of our approach.** We show in Section 5.3 that visibility testing can be transformed into testing for an intersection between a convex object and an algebraic curve. If both entities travel within a simple polygon, this convex object is a linear-complexity convex planar polygon and the algebraic curve is a degree-two planar polynomial. If the trajectories can intersect a simple polygon or cross a planar domain, the convex objects and algebraic curve become higher-dimensional. This transformation allows us to introduce algebraic range searching to swiftly detect intersections between the curve and object. The one-to-one correspondence between mutual visibility and object intersections gives us the first query times for visibility testing that are sublinear in the domain complexity, in a variety of settings.

**Results and structure.** We focus on trajectories of at most two vertices; any set of trajectories of  $\tau$  vertices can be handled by applying our algorithms or queries  $\tau$  times. Our results are summarized in Table 5.1. The value  $k$  is an unspecified constant.

The chapter is structured as follows: Section 5.2 presents preliminary information. In Section 5.3, we discuss our algorithmic results; we build on the structural geometric properties established in this section throughout the chapter. In Section 5.4 we consider the subproblem of preprocessing a convex polygon  $P'$  for intersection queries with quadratic curve segments. We then extend the solution in Section 5.5 to a data structure for visibility testing in a simple polygon  $P$  using multi-level data structures. In Section 5.6 we assume one of the entities is stationary. In Section 5.7 we consider the most general scenario where the trajectories lie in a polygonal domain and may intersect domain edges. Algebraic proofs are relegated to Section 5.A.

| $q$ | $r$ | $P$    | algorithm          | data structure    |                   |                                  | source      |
|-----|-----|--------|--------------------|-------------------|-------------------|----------------------------------|-------------|
|     |     |        |                    | space             | preprocessing     | query                            |             |
| •   | •   | S or I | $\Theta(n)$        | $O(n)$            | $\Theta(n)$       | $\Theta(\log n)$                 | [108]       |
|     |     | D      | $\Theta(n)$        | $O(n^2)$          | $O(n^2)$          | $\Theta(\log n)$                 | [173]       |
| •   | /   | S      | $\Theta(n)$        | $O(n)$            | $O(n \log n)$     | $\Theta(\log n)$                 | Section 5.6 |
|     |     | I      | $\Theta(n \log n)$ | $O(n^2 \log^3 n)$ | $O(n^2 \log^3 n)$ | $O(n^{\frac{3}{4}+\varepsilon})$ | Section 5.6 |
|     |     | D      | $\Theta(n \log n)$ | $O(n^4 \log^3 n)$ | $O(n^4 \log^3 n)$ | $O(n^{\frac{3}{4}+\varepsilon})$ | Section 5.6 |
| /   | /   | S      | $\Theta(n)$        | $O(n \log^4 n)$   | $O(n \log^4 n)$   | $O(n^{\frac{3}{4}} \log^3 n)$    | Section 5.5 |
|     |     | I      | $\Theta(n \log n)$ | $O(n^{3k})$       | $O(n^{3k})$       | $O(\log^k n)$                    | Section 5.7 |
|     |     | D      | $\Theta(n \log n)$ | $O(n^{3k})$       | $O(n^{3k})$       | $O(\log^k n)$                    | Section 5.7 |

**Table 5.1** The two leftmost columns specify if the query entity is a point (•) or line segment (/). The third column specifies if the domain  $P$  is:

(S) a simple polygon where the query segments may not intersect  $P$ ,

(I) a simple polygon where the query segments may intersect  $P$  or

(D) a polygonal domain with  $n$  vertices where the query segments may intersect  $P$ .

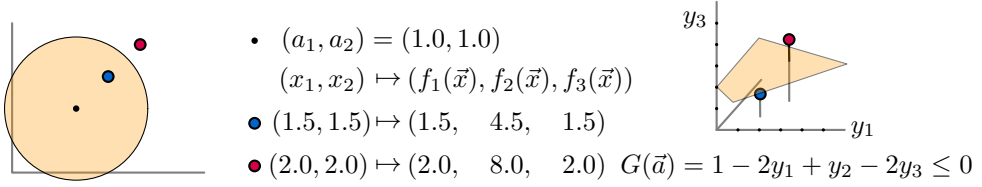
## 5.2 Semi-algebraic range searching

Throughout this chapter we make extensive use of the semi-algebraic (sometimes called linearization) techniques from Agarwal et al. [5]. In this section, we describe this technique in full detail. In a nutshell, semi-algebraic range searching allows us to perform range searching queries on areas that are bounded by a semi-algebraic curve. We use this later to detect visibility between entities. Let  $X$  be a set of  $n$  geometric objects in  $\mathbb{R}^d$ , where each object is parametrized by a vector  $\vec{x}$ . If  $X$  is a set of points, the most natural parametrization is a vector of its coordinates. However,  $X$  may be a complicated algebraic object.

We denote by  $\Gamma$  the family of geometric regions (called semi-algebraic ranges) in  $\mathbb{R}^d$  where each region  $G \in \Gamma$  is bounded by an algebraic curve  $\gamma$  which is parametrized by a vector  $\vec{a}$ . Examples include the family of all disks, or the family of all disks with radius 1. An arbitrary disk can be represented as a vector in many ways. For example, three non-colinear points define a unique circle. Thus, one could represent a circle as a six-dimensional vector which specifies the coordinates of these points. However the larger the representation vector, the more complicated the linearization process becomes. The most efficient representation of a circle is by a three-dimensional vector specifying its center and radius. The family of disks of radius 1 has fewer degrees of freedom and thus their representation can be more efficiently represented (specifically, as a 2-dimensional vector specifying only its center).

Given  $X$  and  $\Gamma$ , we are interested in preprocessing  $X$  such that given any range  $G \in \Gamma$ , we can report which objects of  $X$  intersect  $G$ . To accomplish this, we first want to derive what we have dubbed a *predicate function*  $F(\vec{x}, \vec{a}) \leq 0$ . The predicate function  $F$  takes any instance of  $X$  (parametrized by  $\vec{x}$ ) and any range  $G \in \Gamma$  (parametrized by  $\vec{a}$ ) and outputs a real number. The object intersects the range if and only if the output value is lesser than or equal to zero. We present an example (Figure 5.3): let  $X$  be a set of two-dimensional points parametrized by  $\vec{x} = (x_1, x_2)$  and  $\Gamma$  be the family of disks with radius 1, parametrized by the center coordinates  $\vec{a} = (a_1, a_2)$ . Any point  $(x_1, x_2)$  is contained in this disk if and only if  $F(\vec{x}, \vec{a}) = (a_1 - x_1)^2 + (a_2 - x_2)^2 - 1 \leq 0$  and thus we have found our predicate function.

The predicate function  $F(\vec{x}, \vec{a})$  can be seen as a map from the parameter space of our original intersection problem to the Boolean space  $\{0, 1\}$  and thus  $F$  partitions our parameter space into areas where the answer is *yes* and areas where the answer is *no*. The idea behind semi-algebraic range searching is that we search for our problem solution in this parameter space, as opposed to searching in the space where our problem lives. However, the border of each area does not have to be particularly nice. In our example, each of the *yes* areas is a quadratic-complexity surface in four dimensions. This is where *linearization* comes in. We transform the parameter space through a polynomial map into a  $k$ -dimensional space where the boundary of the *yes* spaces becomes a linear-complexity surface. We wish to mention that in full generality, this map does not have to be polynomial. But for the purpose of this chapter, we restrict the results from [5] to bounded degree polynomials.



**Figure 5.3** The mapping that transforms the two-dimensional containment query into a three-dimensional halfspace range query.

**An example of linearization.** When linearizing the predicate function  $F(\vec{x}, \vec{a})$ , we rewrite the function to the form:  $F(\vec{x}, \vec{a}) = g_0(\vec{a}) + \sum_{i=1}^k g_i(\vec{a})f_i(\vec{x})$  where  $f_i$  and  $g_i$  are polynomials dependent only on  $\vec{x}$  and  $\vec{a}$  respectively. In our example we had:

$$F(\vec{x}, \vec{a}) = (a_1 - x_1)^2 + (a_2 - x_2)^2 - 1 \leq 0.$$

To linearize this function we need to first expand the squares:

$$F(\vec{a}, \vec{x}) = a_1^2 - 2a_1x_1 + x_1^2 - 2a_2x_2 + a_2^2 + x_2^2 - 1.$$

This immediately gives a straight-forward linearization where  $k = 6$ :

$$\begin{aligned}
 g_0(\vec{a}) &= 1 & g_1(\vec{a}) &= a_1^2, & f_1(\vec{x}) &= 1, \\
 g_2(\vec{a}) &= -2a_1, & f_2(\vec{x}) &= x_1, & g_3(\vec{a}) &= 1, & f_3(\vec{x}) &= x_1^2, \\
 g_4(\vec{a}) &= -2a_2, & f_4(\vec{x}) &= x_2, & g_5(\vec{a}) &= a_2^2, & f_5(\vec{x}) &= 1, \\
 g_6(\vec{a}) &= 1, & f_6(\vec{x}) &= x_2^2.
 \end{aligned}$$

However, we can reduce the number of terms by grouping variables and writing:  $F(\vec{x}, \vec{a}) = [a_1^2 + a_2^2 - 1] + [-2a_1](x_1) + [1](x_1^2 + x_2^2) + [-2a_2](x_2)$  and obtain the following linearization where  $k = 3$ :

$$\begin{aligned}
 g_0(\vec{a}) &= a_1^2 + a_2^2 - 1 & g_1(\vec{a}) &= -2a_1, & f_1(\vec{x}) &= x_1, \\
 g_2(\vec{a}) &= 1, & f_2(\vec{x}) &= x_1^2 + x_2^2, & g_3(\vec{a}) &= -2a_2, & f_3(\vec{x}) &= x_2.
 \end{aligned}$$

Agarwal et al. show how to map any  $d$ -dimensional point  $\vec{x}$  to the  $k$ -dimensional point  $f(\vec{x}) = (f_1(\vec{x}), f_2(\vec{x}), \dots, f_k(\vec{x}))$ , and any query range to the  $k$ -dimensional halfspace  $G(\vec{a}) = \left\{ \vec{y} \in \mathbb{R}^k \mid g_0(\vec{a}) + \sum_{i=1}^k g_i(\vec{a})y_i \leq 0 \right\}$  and that  $\vec{x}$  intersects  $G$  if and only if  $f(\vec{x})$  is contained in  $G(\vec{a})$ . In our example, the map  $f$  is the well-known paraboloid projection: we map all the two-dimensional points onto a three-dimensional paraboloid and they are contained in a circle  $G \in G$  if and only if the points lie within a halfspace cutting the paraboloid.

## 5.3 Algorithms for testing visibility

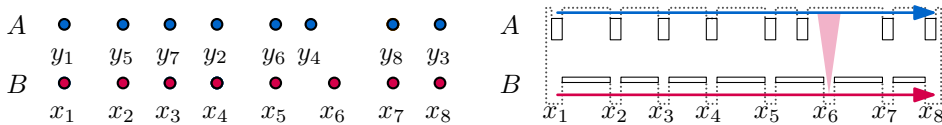
Let  $P$  be a polygonal domain with  $n$  edges and let  $q$  and  $r$  each be a line segment or a point in the plane. We first present an  $O(n \log n)$  time algorithm to solve visibility testing for  $q, r$  and  $P$  and we show this algorithm is tight whenever  $P$  is a polygonal domain, or when the trajectory segments may intersect the domain. Then we show how to solve the visibility testing in linear time whenever  $P$  is a simple polygon and  $q$  and  $r$  are contained in  $P$ ; this is trivially tight. The remaining sections depend on the notion of *hourglass* that is presented here.

**An  $O(n \log n)$  time algorithm.** The entities  $q$  and  $r$  each move along a line segment with constant speed during the time interval  $[0, 1]$ . We assume that both entities start and stop moving at the same time, and thus their speed is determined by the length of their segment. Consider the line  $g(t)$  through both entities at time  $t$ . We dualize this line to a point using classical point-line dualization (i.e. we map the line  $y = ax + b$  to the point  $(a, -b)$ ); this point  $\gamma(t)$  now traces a segment of a curve  $\gamma : [0, 1] \rightarrow \mathbb{R}^2$  in the dual space.<sup>1</sup> In Section 5.A we show the following:

**Lemma 5.1** *The segment  $\gamma$  is a segment of a quadratic curve that has 5 degrees of freedom.*

Let  $e$  be an edge of  $P$  and denote by  $L_e$  the set of lines intersecting  $e$ . The dual of  $L_e$  is a wedge  $\Lambda_e$  [61]. If the segment between  $q$  and  $r$  is blocked by  $e$  at time  $t$  then the line  $g(t)$  must lie in  $L_e$ . In the dual, this means that the curve segment  $\gamma$  must intersect  $\Lambda_e$ . There are at most three connected time intervals where a quadratic curve segment  $\gamma$  can intersect a wedge  $\Lambda_e$ ; it follows that each edge  $e$  has at most three connected time intervals where it blocks the visibility between  $q$  and  $r$ . This observation leads to a straightforward algorithm to test if there is a time at which  $q$  can see  $r$ : for each edge  $e \in P$ , we compute the at most three time intervals where it blocks visibility between  $q$  and  $r$  in constant time. We sort these time intervals in  $O(n \log n)$  time and verify if their union covers  $[0, 1]$ . Thus we conclude:

<sup>1</sup>Throughout this chapter we follow the convention of using Latin letters for objects in the primal space and Greek letters for their duals.



**Figure 5.4** The reduction when  $P$  is a polygonal domain with  $\Omega(n)$  holes or when  $P$  is a simple polygon where edges may cross the polygon edges (dotted). The set  $B$  is a sorted set of real numbers. The set  $A$  is an unsorted set of real numbers. For every number, we construct a constant-complexity component of the polygonal domain.

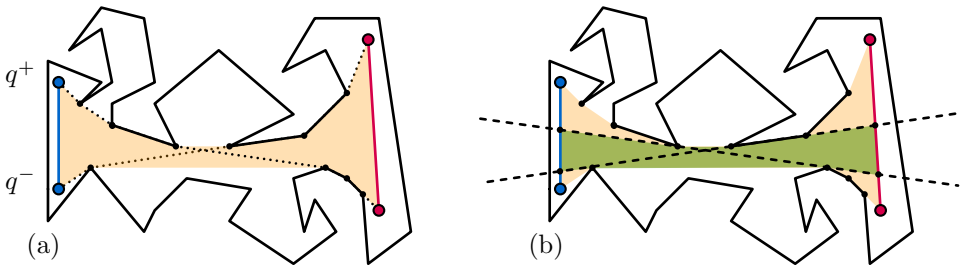
**Theorem 5.1** *Given a polygonal domain  $P$  with  $n$  vertices and moving entities  $q$  and  $r$ , we can test trajectory visibility in  $O(n \log n)$  time.*

**An  $\Omega(n \log n)$  lower bound.** If  $P$  is a polygonal domain with  $\Omega(n)$  holes, or whenever entities are allowed to cross polygon edges, this result is tight: suppose we are given a set  $A$  of  $n$  real numbers and we want to test if  $A = B$  for a given arbitrary sorted set  $B = \{x_1, x_2, \dots, x_n\}$ . Ben-Or [18] shows that this problem has an  $\Omega(n \log n)$  lower bound in the algebraic decision tree model and therefore, in our proposed real RAM model (skip ahead to Chapter 8 for details on lower bounds).

This leads to the following reduction (illustrated by Figure 5.4), which we can construct in linear time. Given the sorted set  $B$ , we construct a set of  $n$  horizontal edges. For the edge  $e_i$ , its  $y$ -coordinates are 0 and  $x$ -coordinates are  $[x_i + \varepsilon, x_{i+1} - \varepsilon]$  where  $\varepsilon$  is smaller than half of the minimal difference between two consecutive numbers in  $B$ . The value for  $\varepsilon$  can be computed in linear time since  $B$  is sorted. Given the set  $A$ , we construct for each value  $y \in A$  an axis-aligned rectangle centered at the segment between  $(y, 1)$  and  $(y, 2)$  with a width of  $2\varepsilon$ . The entity  $q$  walks from the point  $(x_1, -1)$  to  $(x_n, -1)$  and entity  $r$  walks from  $(x_1, 3)$  to  $(x_n, 3)$ . Suppose the number  $x_j$  from  $B$  is not in  $A$ , then  $q$  can see  $r$  at the  $x$ -coordinate  $x_j$ .

This construction trivially extends to the scenario where  $P$  is a simple polygon and trajectories may cross edges of  $P$ , by connecting all components. Note that this construction also extends to the case where one of the two entities is stationary: consider the cone between the stationary entity  $q$  and a horizontal line segment trajectory  $r$ : we can transform the set  $B$  into a set of  $n$  horizontal edges that cut in the cone between  $q$  and  $r$ . Each rectangle modelling a number  $a \in A$  gets stretched such that it intersects a ray from  $q$  to  $r$ . Hence we conclude:

**Theorem 5.2** *There exists a polygonal domain  $P$  with  $n$  vertices, and entities  $q$  and  $r$  moving inside  $P$ , or a simple polygon  $P$  with  $n$  vertices and entities  $q$  and  $r$  that move through  $P$ , for which testing trajectory visibility requires  $\Omega(n \log n)$  time.*



**Figure 5.5** (a) Two segments  $q$  and  $r$  in a simple polygon with their hourglass  $H(q, r)$  in orange. (b) The visibility glass  $L(q, r)$  is the hourglass between two subsegments of  $q$  and  $r$ , that are bounded by two tangents.

**A linear-time algorithm for when  $P$  is a simple polygon and  $q, r \subset P$ .** Any segment between  $q$  and  $r$  that is contained in  $P$  is a geodesic shortest path in  $P$  between a point on  $q$  and a point on  $r$ . Guibas and Hershberger [108] define, for any two segments  $q$  and  $r$  in a simple polygon  $P$ , the *hourglass*  $H(q, r)$  to be the union of all shortest paths between points on  $q$  and  $r$ . The hourglass  $H(q, r)$  is a subset of  $P$  and bounded by the segments  $q$  and  $r$  and by two shortest paths. The “upper” chain<sup>2</sup> is the shortest path between the upper end points  $q^+$  and  $r^+$  of  $q$  and  $r$ , and the “lower” chain is the shortest path between  $q^-$  and  $r^-$  (refer to Figure 5.5). We define the *visibility glass*  $L(q, r)$  as the (possibly empty) union of line segments between  $q$  and  $r$  that are contained in  $P$ . Notice that  $L(q, r)$  is a subset of  $H(q, r)$ .

**Observation 5.1** For any two segments  $q, r \subset P$  either  $L(q, r)$  is empty or there exist segments  $q' \subseteq q, r' \subseteq r$  such that  $L(q, r) = H(q', r')$ . Moreover if  $q' \neq q$  or  $r' \neq r$ ,  $q'$  and  $r'$  are bounded by the bitangent on a shortest path between endpoints of  $q$  and  $r$ .

**Proof** Suppose that the interior of the upper and lower chains of  $H(q, r)$  intersect. Then the visibility glass  $L(q, r)$  is either a single segment or empty. Thus we can either find two points  $q'$  and  $r'$  on  $q$  and  $r$  whose line segment forms  $H(q', r') = L(q, r)$  or  $L(q, r)$  is empty. If the interior of the upper and lower chains are disjoint then they are semi-convex [108]. Consider the shortest path from  $q^+$  to  $r^-$ , it has one edge  $(u, v)$  connecting the upper and lower chains. This is the unique edge for which the path makes a clockwise turn at  $u$  and a counterclockwise turn at  $v$  or vice versa (Figure 5.6). There exists an edge with similar properties on the shortest path between  $q^-$  and  $r^+$ . The extension of these two edges bounds  $q'$  and  $r'$  [53]. ■

Chazelle and Guibas [53] note that (the supporting lines of) all line segments in  $L(q, r)$  can be dualized into a convex polygon of linear complexity which we denote by  $\Lambda(q, r)$ . The shortest path between two points in  $P$  can be computed in linear time [109]. Finding the bitangents also takes linear time. It follows that we can compute  $L(q, r)$  and its dual  $\Lambda(q, r)$  in linear time. Suppose that we are given two entities  $q$  and  $r$  contained in a simple polygon  $P$ . Recall that the line  $g(t)$  through  $q$  and  $r$  traces a quadratic segment  $\gamma$  in the dual.

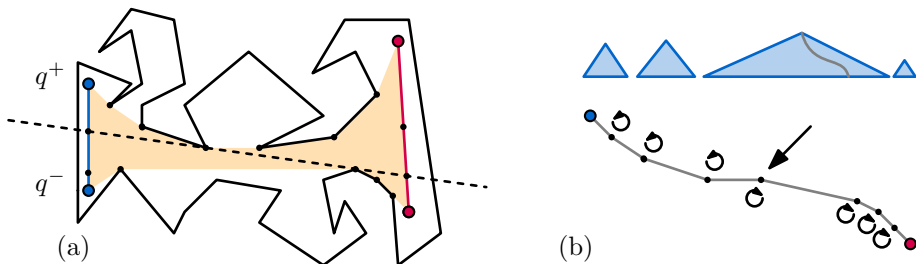
**Observation 5.2** Entities  $q$  and  $r$  are mutually visible at time  $t$  if  $\gamma(t)$  lies in  $\Lambda(q, r)$ .

**Proof** The entities can see one another at time  $t$  if and only if  $g(t) \in L(q, r)$ . ■

We can derive  $\gamma$  in constant time and construct  $\Lambda(q, r)$  in linear time. Then we can check if a quadratic curve intersects a convex polygon in linear time and we conclude:

**Theorem 5.3** Given a simple polygon  $P$  with  $n$  vertices and two entities  $q$  and  $r$  moving linearly inside  $P$ , we can test trajectory visibility in  $\Theta(n)$  time.

<sup>2</sup>We use “upper” and “lower” since they intuitively correspond to our figures.



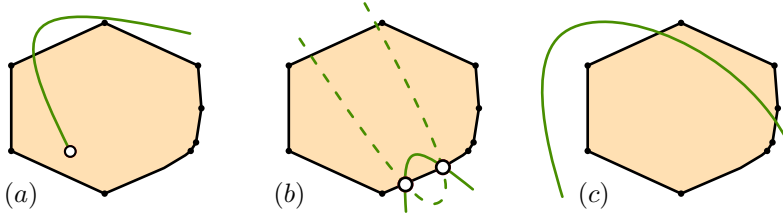
**Figure 5.6** (a) We consider the path from  $q^+$  to  $r^-$  and we want to find the bitangent that bounds the visibility glass  $L(q, r)$ . (b) This bitangent is the extension of the edge where the path from  $q^+$  to  $r^-$  changes rotation around the vertex. We get this path as a collection of search trees and perform binary search to find this vertex.

## 5.4 Testing curve intersection with a convex polygon

We now turn our attention to the data structure question: can we preprocess a simple polygon  $P$  such that trajectory visibility may be tested efficiently (i.e. in sublinear time) for a pair of query segments  $q, r$ ? By Observation 5.2, we can phrase such a query as an intersection between a quadratic curve segment and a convex polygon. Note, however, that both the curve segment and the convex polygon depend on the query segments  $q$  and  $r$ . As an intermediate step, we study in this section a simplified problem where the convex polygon is independent of  $q$  and  $r$ . In particular, the question that we study is: let  $P'$  be a convex polygon with  $n$  edges. Is it possible to preprocess  $P'$  such that for any quadratic curve segment  $\gamma$ , we can quickly test if  $\gamma$  intersects  $P'$ ? We then use our solution to this question as a subroutine in Section 5.5.

**Halfspace range searching.** We showed in Section 5.2 how to construct a predicate function that determines if an algebraic range parametrized by  $\vec{a}$  intersects an object parametrized by  $\vec{x}$ . We also showed how to linearize such a predicate into  $k$  dimensions such that each object becomes a point in  $\mathbb{R}^k$  and the algebraic range becomes a range in  $\mathbb{R}^k$  bounded by a halfspace.

After linearization, the resulting set of  $n$   $k$ -dimensional points can be stored in a data structure of linear size such that the points in a query halfspace can be counted in  $O(n^{1-\frac{1}{k}})$  time using partition trees [50]. Testing if a query halfspace is empty can be done in expected  $O(n^{1-\frac{1}{k/2}})$  time. If we are willing to use much more space  $O(n^{k+\epsilon})$ , we can test if a query halfspace is empty in  $O(\log^k n)$  time [50, 157]. A combination of partition trees and cutting trees allows one to interpolate between these space requirements and query times.



**Figure 5.7** The cases (a), (b) and (c) of intersection between  $\gamma$  and  $P'$  where for case (b) the hyperbolic segment can either go ‘outwards’ or ‘inwards’. We detect case (a) and (b) with conventional means.

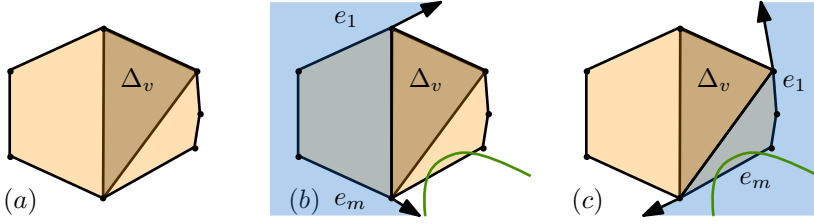
**Semi-algebraic range searching and our intersection query.** We now apply semi-algebraic range searching to test if a convex polygon  $P'$  of  $n$  edges is intersected by a quadratic curve segment  $\gamma$ . We parametrize each edge  $e \in P'$  with a four-dimensional vector  $\vec{x}_e$  that specifies its start and end points. A quadratic curve segment  $\gamma$  per definition is a semi-algebraic range parametrized by its own parameters  $\vec{a}_\gamma$ . To apply semi-algebraic range searching, we need to design a predicate function  $F(\vec{x}_e, \vec{a}_\gamma)$  that outputs a negative number if and only if an edge  $e$  is intersected by  $\gamma$ .

It is tempting to immediately construct such a predicate function using the parameters  $\vec{x}_e$  and  $\vec{a}_\gamma$  only (ignoring geometric facts such as the convexity or connectedness of  $P'$ ). However, we have to linearize the resulting predicate function  $F(\vec{x}_e, \vec{a}_\gamma)$  into  $k$  terms. The more complex the description of the objects, the query, and their intersection, the more complex the predicate will be and therefore the higher this number  $k$  will be.

**Geometry of our intersection query.** Let  $P'$  be a convex polygon with  $n$  edges. Let  $\Gamma$  denote the family of degree-2 curves  $\Gamma := \{a_1x^2 + a_2x + a_3xy + a_4y + a_5y^2 + a_6 = 0 \mid (a_1, a_2, a_3, a_4, a_5, a_6) \in \mathbb{R}^6\}$ . We denote by  $\gamma$  be a quadratic curve segment ending in the points  $s$  and  $z$  and say that the segment  $\gamma$  is a segment of the full curve  $G \in \Gamma$  parametrized by some vector  $\vec{a} = (a_1, \dots, a_6)$ . Observe (Figure 5.7) that if  $e$  is an edge of  $P'$  that is intersected by  $\gamma$  then either:

- (a) at least one endpoint  $s$  or  $z$  lies in  $P'$ , or otherwise
- (b) the segment  $\gamma$  intersects  $e$  twice (we call this *dipping*) or
- (c) the edge  $e$  has has one endpoint in the interior of  $G$   
(the area  $\{a_1x^2 + a_2x + a_3xy + a_4y + a_5y^2 + a_6 \leq 0\}$ ) and one in its complement.

We show how to detect each case, one by one, with three separate Lemmas.



**Figure 5.8** (a) The root triangle  $\Delta_v$  that splits the convex polygon into roughly equal-complexity halves. (b) For the left subpolygon,  $\gamma$  can dip an edge only if it is in the blue area. (c) For the right subpolygon we can make a similar area.

**Lemma 5.2** *We can preprocess a convex polygon  $P'$  consisting of  $n$  edges in  $O(n)$  time and using  $O(n)$  space, such that given a degree-2 curve segment  $\gamma$  we can detect case (a) in  $O(\log n)$  time.*

**Proof** To detect case (a) we refer to Kirkpatrick's well-known data structure for planar point location in a simple polygon [130]. This structure can store  $P'$  using  $O(n)$  space and  $O(n)$  preprocessing time, such that given any point we can determine if it lies in  $P'$  in  $O(\log n)$  time. ■

**Lemma 5.3** *We can preprocess a convex polygon  $P'$  consisting of  $n$  edges in  $O(n)$  time and using  $O(n)$  space, such that given a degree-2 curve segment  $\gamma$ , such that cases (a) and (c) do not apply. Then we can find an edge  $e$  of  $P'$  intersected through case (b) in  $O(\log n)$  time (if such an edge  $e$  exists).*

**Proof** We use the Dobkin-Kirkpatrick hierarchy [71]: which is a balanced decomposition of  $P'$  into triangles. Specifically, each node in the decomposition represents a subpolygon  $P''$  of  $P'$  and stores a triangle  $\Delta_v$  that splits  $P''$  in roughly-equal parts. We will refer to these parts as the 'left' and 'right' part and say that these subpolygons are bounded by a left or right chain respectively. Given this hierarchy and a segment  $\gamma$ , we detect if there is at least one edge  $e$  in  $P'$  that is dipped by  $\gamma$  as follows (Figure 5.8): for the root node  $v$ , we check if the triangle  $\Delta_v$  intersects  $\gamma$  in constant time. if so, we found a point of intersection. Otherwise, we denote by  $R = (e_1, e_2, \dots, e_m)$  the border of the right chain and by  $R_v$  the union of the halfspaces right of the supporting line of:  $e_1, e_m$  and one edge of  $\Delta_v$  (refer to the blue area in the figure). The area  $L_v$  is defined symmetrically. We test if  $\gamma$  is contained in  $R_v$  or  $L_v$ . If  $\gamma$  is contained in both then the segment cannot intersect  $P'$ . We show that if  $\gamma$  intersects an edge of  $P'$  through case (b) then it can intersect either  $L$  or  $R$  and must be contained in  $L_v$  or  $R_v$  respectively. This allows us decide in constant time whether we can discard the edges in  $L$  or  $R$  (or both). Since the decomposition has height at most  $O(\log n)$ , this proves that this recursive procedure can detect an intersection of case (b) in logarithmic time.

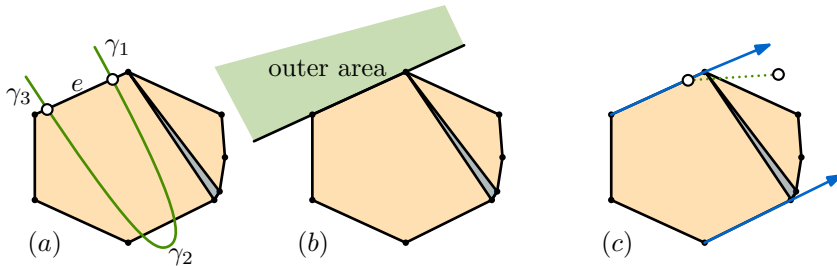
To this end, we consider the set of all edges from  $P'$  that are intersected through case (b) and denote by  $e$  the edge which cuts off the largest segment of  $\gamma$  (Figure 5.9).

Per assumption, edges of  $P'$  are intersected by  $\gamma$  only through case (b) hence this edge  $e$  is unique. We assume that  $e$  is contained in  $L$  (the argument for when  $e$  is contained in  $R$  is symmetrical).

There are two halfspaces which have the line supporting  $e$  as its boundary. One of these two halfspaces is contained in  $L_v$  and we call this the ‘outer’ halfspace. We call its complement the ‘inner’ halfspace. The edge  $e$  partitions  $\gamma$  into three segments:  $\gamma_1, \gamma_2$  and  $\gamma_3$ . We claim that  $\gamma_2$  is contained in the inner halfspace. Indeed, suppose for the sake of contradiction that  $\gamma_2$  is contained in the outer halfspace then because  $P'$  is a convex polygon and  $e$  is one of its edges, the segments  $\gamma_1$  and  $\gamma_3$  are (partially) contained in  $P'$ . However, per assumption the segment  $\gamma$  contains no endpoints in  $P'$  and  $\gamma_1$  and  $\gamma_3$  must thus ‘leave’  $P'$  at some point. However, since per assumption there is no intersection of case (c),  $\gamma_1$  and  $\gamma_3$  must leave  $P'$  through the same edge  $e'$  of  $P'$ . This edge  $e'$  contradicts the definition of  $e$ .

Thus,  $\gamma_2$  is contained in the inner halfspace. Per definition,  $\gamma_1$  and  $\gamma_3$  are contained in  $L_v$ . We claim that  $\gamma_2$  is also contained in  $L_v$ : observe that the area bounded by  $\gamma_2$  and  $e$  is convex. Because  $P'$  is convex, any line outside of  $R_v$  to a point on  $e$  must cross the triangle  $\Delta_v$ . Thus, if  $\gamma_2$  is not contained in  $L_v$  it must cross  $\Delta_v$  which is a contradiction. The above analysis shows that if  $\gamma$  intersects an edge of  $P'$  through case (b) but no edge through case (a) and (c), then  $\gamma$  either intersects  $\Delta_v$  or is contained in  $R_v$  or  $L_v$  and this concludes the proof. ■

The curve  $G$  of which  $\gamma$  is a segment divides the plane into two areas,  $G^- := \{a_1x^2 + a_2x + a_3xy + a_4y + a_5y^2 + a_6 \leq 0\}$  and its complement  $G^+$ . For an intersection of type (c) an edge  $((x_1, x_2), (x_3, x_4))$  of  $P'$  is intersected by  $\gamma$  with one endpoint in  $G^-$  and the other in  $G^+$ . Since  $G$  is a curve with  $k + 1 \leq 5$  degrees of freedom, the formulation of  $G^-$  and  $G^+$  is a predicate that specifies whenever a point  $\vec{x} = (x_1, x_2)$  lies in  $G^-$  or  $G^+$  with  $k$  linearized terms. Therefore, we can detect if an edge has two endpoints on opposite sides of  $G$  with two consecutive halfspace range queries in  $\mathbb{R}^k$ . We build a three level data structure where the first two levels are 5-dimensional partition trees [50, 156]. On each node in the second level we build a binary tree on the clockwise ordering (with respect to  $P'$ ) of the edges in that node.



**Figure 5.9** The construction for the proof of Lemma 5.3. The line between any point of  $e$  and any point outside of  $L_v$  must intersect  $\Delta_v$ .

During query time, we map the degree-2 curve  $G$  to its two  $k$ -dimensional halfspaces  $G^+$  and  $G^-$ . With two consecutive halfspace range queries we obtain the collection  $E_G(P)$  of edges which have one endpoint on either side of  $G$  in  $O(n^{1-\frac{1}{k}})$  time. Note that the set  $E_G(P)$  does not have to be a connected set of edges of  $P$  (refer to Figure 5.7 (b)). The set  $E_G(P)$  consists of edges that are intersected by the full curve  $G$ , but not necessarily edges that are intersected by the curve segment  $\gamma$ .

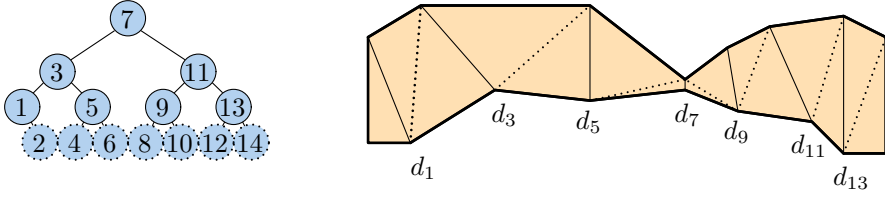
Consider the subset of  $E_G(P)$  that is intersected by  $\gamma$ . When we order  $E_G(P)$  based on the clockwise ordering of  $P'$ , this subset must be a connected interval. We identify this interval as follows: from the partition trees, the set  $E_G(P)$  is returned as  $O(n^{1-\frac{1}{k}})$  subtrees  $\{T_1, T_2, \dots, T_m\}$ . For each subtree  $T_i$ , we had constructed an associated binary search tree on the edges stored in the root. Because of the earlier discussed property, the subset of  $E_G(P)$  that is intersected by the segment  $\gamma$  must be a consecutive subset of the leaves of  $T_i$ . Thus using  $T_i$  we can obtain these consecutive leaves in  $O(\log n)$  time by testing if the segment  $\gamma$  lies before or after the point of intersection between  $G$  and an edge in  $E_G(P)$ . The time and space needed for detecting case (c) dominates the time and space needed for cases (a) and (b). In full generality, we assume that  $\gamma$  comes from a curve with  $a + 1$  degrees of freedom (at most six) and we conclude:

**Theorem 5.4** *Let  $P'$  be a convex polygon with  $n$  vertices. In  $O(n \log^2 n)$  time we can build a data structure of size  $O(n \log^2 n)$  with which we can test if an arbitrary degree-2 query curve segment  $\gamma$  with  $a + 1 \leq 6$  degrees of freedom intersects  $P'$  in  $O(n^{1-\frac{1}{a}} \log n)$  time.*

## 5.5 Two moving entities inside a simple polygon

In this section we build a data structure to answer trajectory visibility queries when both the entities  $q$  and  $r$  move linearly, possibly at different speeds, inside a simple polygon  $P$ . Our main approach is the same as in our algorithm from Theorem 5.3: we obtain the convex polygon  $\Lambda(q, r)$  that is the dual of the visibility glass  $L(q, r)$ , and test if the curve segment  $\gamma$  tracing the line through  $q$  and  $r$  in the dual space intersects  $\Lambda(q, r)$ . By Observation 5.2 this allows us to answer trajectory visibility testing. The main challenge is that we cannot afford to construct  $\Lambda(q, r)$  explicitly. Instead, our data structure will allow us to obtain a compact representation of  $\Lambda(q, r)$  that we can query for intersections with  $\gamma$ .

To obtain  $\Lambda(q, r)$  we use a variation of the two-point shortest-path query data structure of Guibas and Hershberger [108]. Their data structure (Figure 5.10) compactly stores a collection of hourglasses that can be concatenated to obtain a shortest path between two arbitrary points  $p, p' \in P$ . All shortest paths, in particular the boundaries of the hourglasses, are represented using balanced binary search trees storing the vertices on the path. By reusing shared subtrees these  $O(n)$  hourglasses can be stored using only  $O(n)$  space. To report the shortest path between two query points their report query concatenates  $O(\log n)$  of these hourglasses (Figure 5.11). The result of such a query is thereby represented by a collection of  $O(\log n)$  subtrees.

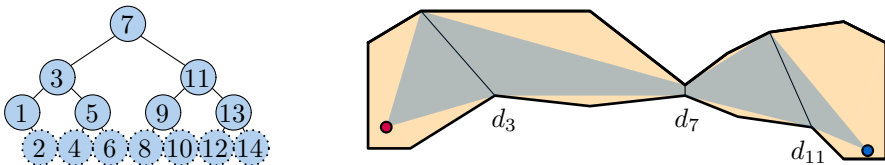


**Figure 5.10** A simple polygon  $P$ , triangulated with diagonals. Even-indexed diagonals have dotted lines. The data structure maintains a balanced decomposition of the triangulation, where the diagonal stored in a node splits the associated polygon into roughly two equal halves.

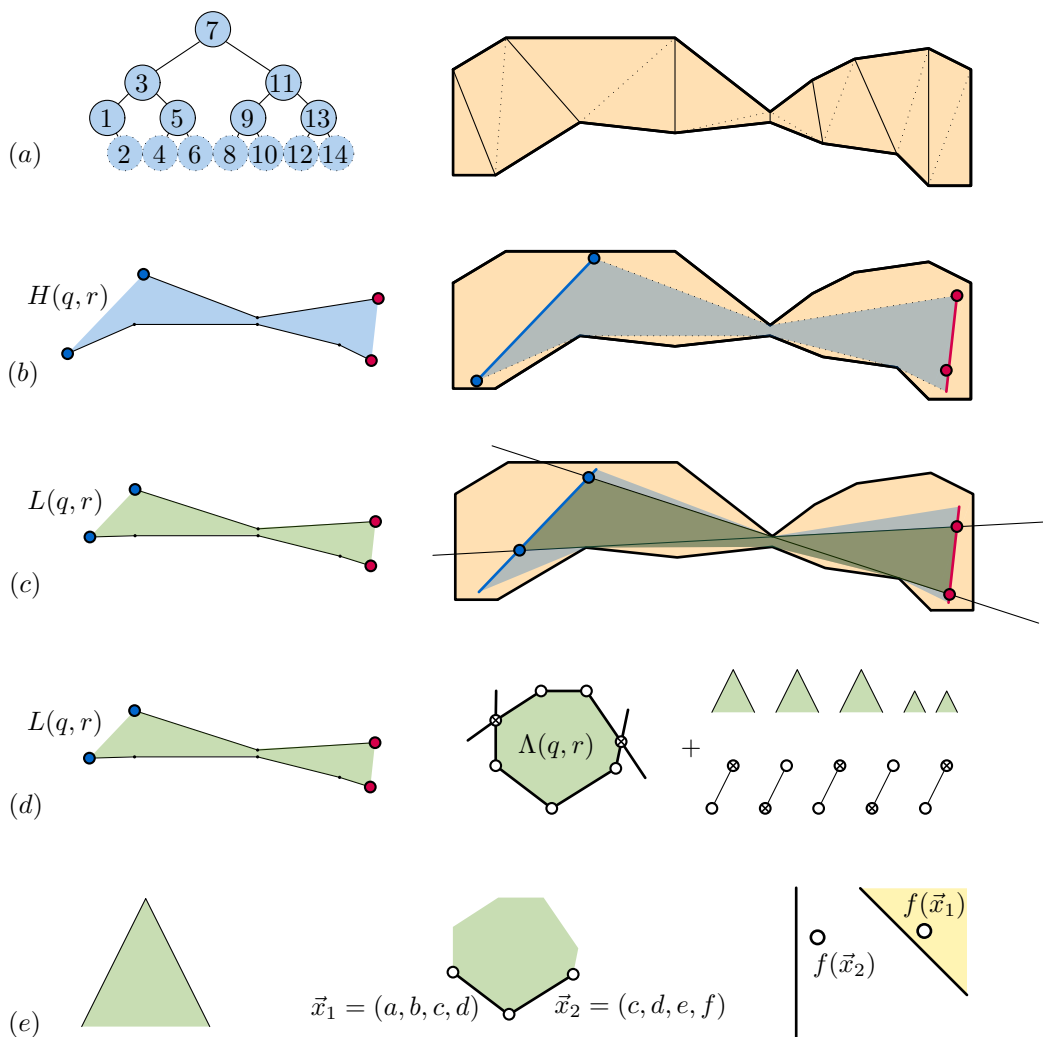
We now present an overview of our data structure used to determine visibility between two trajectories. For a sketch of our query approach, we refer ahead to Figure 5.12. The structure is a three level data structure:

1. The first level is the Guibas and Hershberger structure, where we store the hourglasses explicitly. In particular, the vertices on the boundary of an hourglass are stored in the leaves of a balanced binary search tree. The internal nodes correspond to semi-convex subchains. Let  $T$  denote the collection of all these nodes. Each node  $v \in T$  stores its subchain  $C_v$  in an associated data structure:
2. Specifically, we dualize the supporting lines of the edges in  $C_v$  to points (refer to Figure 5.13). Two consecutive edges produce two points in the dual, which we again connect into semi-convex polygonal chains. The second level stores each of these line segments in a partition tree, so that given a query segment  $\gamma$  in the dual, we can find all edges intersected by the full curve  $G$  (see Theorem 5.4).
3. For each node in the partition tree, we store the associated edges in the dual in their clockwise order so that given the set  $E_G$  of edges intersected by  $G$ , we can obtain the subset of edges intersected by  $\gamma$ .

**Lemma 5.4** *The first level of our data structure stores at most  $O(n \log^2 n)$  edges.*



**Figure 5.11** Given two points  $q$  and  $r$ , their shortest path can be represented as a concatenation of at most  $O(\log n)$  pre-stored hourglasses in the polygon.



**Figure 5.12** (a) The base level of our data structure is the hierarchical triangulation. (b) Given  $q$  and  $r$ ,  $H(q, r)$  is bounded by two shortest paths. (c) We search for the bitangents that bound  $L(q, r)$  in  $O(\log^2 n)$  time. (d) We obtain its dual  $\Lambda(q, r)$  as a set of  $O(\log n)$  subtrees and  $O(\log n)$  additional new edges in  $O(\log n)$  time. (e) For each of the subtrees, each root stores a partition tree that supports the semi-algebraic search. For each of the  $O(\log n)$  additional edges we check for an intersection with the query segment in constant time.

**Proof** The Guibas and Hershberger data structure is essentially a balanced hierarchical subdivision that recursively partitions the polygon into two roughly equal size subpolygons. For every diagonal associated with a node in the decomposition, we explicitly store an hourglass with every diagonal associated with an ancestor [108]<sup>3</sup>. It follows that all hourglasses of a subpolygon of size  $m$  use at most  $O(m \log m)$  space. The height of the balanced hierarchical subdivision is  $O(\log n)$ . So each edge is present in at most  $O(\log n)$  nodes, and subsequently in  $O(\log^2 n)$  hourglasses per node. We conclude that the total number of edges stored in the first level of our data structure is at most  $O(n \log^2 n)$ . ■

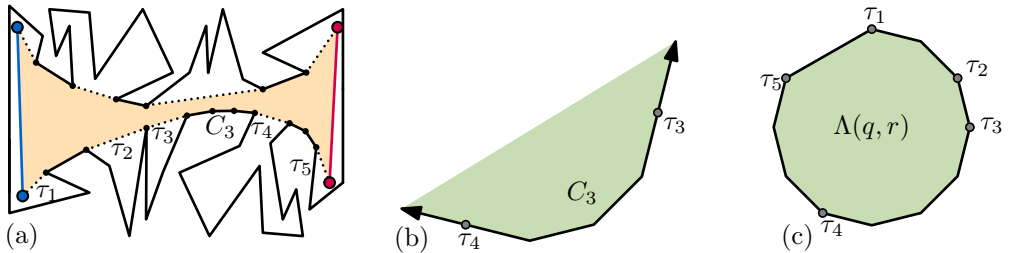
For a level one node  $v$  of size  $m$  in the data structure, the associated data structure (the partition tree  $\Delta_v$ , and the binary tree in each node of  $\Delta_v$ ) has size  $O(m \log^2 m)$  and can be built in  $O(m \log^2 m)$  time. It follows that our data structure uses  $O(n \log^4 n)$  space in total, and can be built in  $O(n \log^4 n)$  time.

**Querying the data structure.** When we get two trajectories  $q$  and  $r$  for visibility testing, we have to test if the curve  $\gamma$  (traced by the point dual to the line through  $q$  and  $r$ ) intersects the visibility glass  $\Lambda(q, r)$ . By Observation 5.1 the primal representation  $L(q, r)$  of  $\Lambda(q, r)$  is an hourglass  $H(q', r')$ . We now argue that (Figure 5.12):

- we can find the subsegments  $q'$  and  $r'$  in  $O(\log n)$  time,
- our data structure can report  $O(\log^2 n)$  nodes from  $T$  that together represent an hourglass  $H(q', r') = L(q, r)$ , and
- we can then test if  $\gamma$  intersects  $\Lambda(q, r)$  by using the associated data structures of these reported nodes. This will result in  $O(n^{\frac{3}{4}} \log^3 n)$  query time.

**Lemma 5.5** *Given our data structure, we can detect if  $L(q, r)$  is empty, or compute the subsegments  $q' \subseteq q$  and  $r' \subseteq r$  such that  $L(q, r) = H(q', r')$  in  $O(\log n)$  time.*

<sup>3</sup>We use the version of Guibas and Hershberger's structure that achieves only  $O(\log^2 n)$  query time.



**Figure 5.13** (a) An hourglass between  $q$  and  $r$  in orange. The lower chain consists of four chains that coincide with  $P$ , joined by outer tangents in dotted lines labelled  $\tau_1 \dots \tau_5$ . (b) The area bounded by  $P$  and the dualized chain  $C_3$ . Note that this chain has four edges since in the primal  $C_3$  has four vertices. (c) A simplified version of  $\Lambda(q, r)$ . Outer tangents become vertices of  $\Lambda(q, r)$ .

**Proof** By Observation 5.1 the visibility glass  $L(q, r)$  is either empty or equal to the hourglass  $H(q', r')$  for two subsegments  $q'$  and  $r'$ , and these two subsegments are bounded by the two bitangents of  $H(q, r)$ . These bitangents are the extension of two edges, from the shortest paths between the edges of  $q$  and  $r$  (Figure 5.12(b)). We explained in the proof of Observation 5.1 that the hourglass  $H(q, r)$  has an upper and lower semi-convex chain which may or may not share a point. The upper and lower chains are both a shortest path between endpoints of  $q$  and  $r$ . We can obtain them using the data structure **D** from [108] as a balanced binary search tree and we can verify if they share a point using this tree. If that is the case then  $L(q, r)$  is either empty or a single segment and we can verify this using an additional  $O(\log n)$  time.

If the upper and lower chains do not share a point then we want to identify the subsegments  $q'$  and  $r'$  for which  $L(q, r) = H(q', r')$ . Recall that  $q'$  and  $r'$  are bounded by the bitangents of  $H(q, r)$ . Such a bitangent is the extension of an edge  $(u, v)$  on the shortest path between two endpoints of  $q$  and  $r$  (Figure 5.12(c)). The edge  $(u, v)$  is the unique edge on this path for which the path makes a clockwise turn at  $u$  and a counterclockwise turn at  $v$  or vice versa. Using **D** we can obtain any path as a balanced binary search tree. We perform a binary search on this tree to identify the edge  $(u, v)$  whose endpoints have this unique clockwise ordering.

Given  $(u, v)$ , we compute in constant time the intersection of its extension with  $q$  and  $r$  to obtain  $q'$  and  $r'$ . With the same procedure we obtain  $L(q, r) = H(q', r')$  in  $O(\log n)$  time with two shortest path queries. ■

We use Lemma 5.5 to find the endpoints  $q_1, q_2$  of  $q'$  and  $r_1, r_2$  of  $r'$ , respectively. We can obtain the shortest paths  $\pi(r_1, q_1)$  and  $\pi(r_2, q_2)$  bounding  $L(q, r) = H(q', r')$  by concatenating  $O(\log n)$  of the pre-stored hourglasses. We concatenate all  $O(\log n)$  hourglasses in the dual: two contiguous subchains in the primal become two contiguous subchains in the dual. In constant time, we compare the four halflines ending the subchains and compute their point of intersection (in the primal, these points of intersection are the bridge segments on the shortest path). This way, we obtain  $\Lambda(q, r)$  as  $O(\log n)$  subtrees (each representing a subchain) and  $O(\log n)$  additional edges (each ending in a manually computed point of intersection). See Figure 5.12(d)+5.13.

To check if the quadratic query segment  $\gamma$  intersects  $\Lambda(q, r)$  we check if one of the endpoints of  $Q$  lies in  $\Lambda(q, r)$ ; this is the case only if the shortest path  $\pi(r_1, q_1)$  or  $\pi(r_2, q_2)$  is actually a single segment, or when  $\gamma$  intersects the boundary of  $\Lambda(q, r)$ . To this end, we query for each root  $v$  of a subtree, its associated second level data structure  $\Delta_v$  (Figure 5.12(e)). Since  $\gamma$  has  $k + 1 = 5$  degrees of freedom (Lemma 5.1), querying the structure of Theorem 5.4 takes  $O(n^{\frac{3}{4}} \log n)$  time per root  $v$ , and thus  $O(n^{\frac{3}{4}} \log^2 n)$  total time. We test for each of the  $O(\log n)$  additional segments if they intersect  $\gamma$  manually in constant time per segment. Therefore, we obtain the following:

**Theorem 5.5** *Let  $P$  be a simple polygon with  $n$  vertices. We can store  $P$  in a data structure of size  $O(n \log^4 n)$  that allows us to answer trajectory visibility testing in  $O(n^{\frac{3}{4}} \log^2 n)$  time. Building the data structure takes  $O(n \log^4 n)$  time.*

## 5.6 One moving entity in a polygonal domain

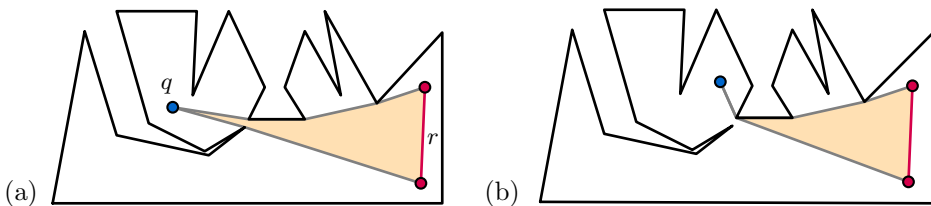
In this section we develop data structures that can efficiently answer trajectory visibility testing in case one of the entities  $q$  is stationary, while  $r$  travels along a line segment. If  $r$  is restricted to be contained in a simple polygon  $P$ , the results of the previous section apply to this case. However in this section we present a simpler solution using linear space with  $O(\log n)$  query time.

We consider three variants of this setting:

1. The domain  $P$  is a simple polygon and  $r$  is contained in  $P$ ,
2. The domain  $P$  is a simple polygon but the trajectory  $r$  may intersect edges, or
3.  $P$  is a polygonal domain and  $r$  may intersect edges of  $P$ .

**Case 1: the entity  $r$  is contained in a simple polygon.** Consider the shortest paths  $\pi_1, \pi_2$  from  $q$  to the end points of  $r$ . Observe that if edges of  $\pi_1$  and  $\pi_2$  coincide, they coincide in a connected chain from  $q$  [108]. Moreover (Figure 5.14) if more than one line segment of  $\pi_1$  and  $\pi_2$  coincide, then any shortest path from  $q$  to a point on  $r$  cannot be a single line segment. If no edges of  $\pi_1$  and  $\pi_2$  coincide then there is at least one point on  $r$ , whose shortest path to  $q$  is a line segment. If exactly one line segment of  $\pi_1$  coincides with a segment of  $\pi_2$ , then that segment must be connected to  $q$  and if there is a line-of-sight between  $q$  and  $r$ , it has to follow that line segment. This observation allows us to answer a visibility test by considering only the first three vertices of  $\pi_1$  and  $\pi_2$ . These vertices can be found in  $O(\log n)$  time using the two-point shortest path data structure of Guibas and Hershberger [108]. We conclude:

**Theorem 5.6** *Let  $P$  be a simple polygon with  $n$  vertices. We can store  $P$  in a data structure of size  $O(n)$  that allows us to answer trajectory visibility testing between a static and a linearly moving entity in  $O(\log n)$  time. Building the data structure takes  $O(n)$  time.*

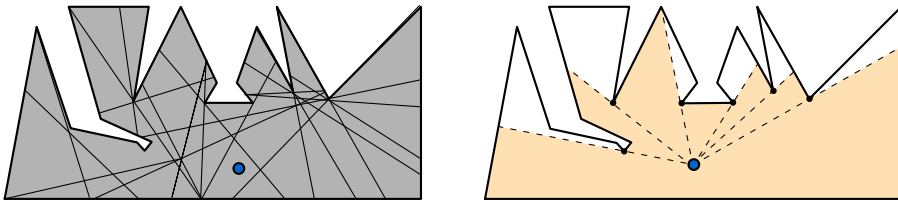


**Figure 5.14** A query point  $q$  and trajectory  $r$ . (a) The funnel is open, and thus the entities are mutually visible. (b) When the funnel is closed, entities are mutually visible if and only if the first two funnel edges from  $q$  are straight.

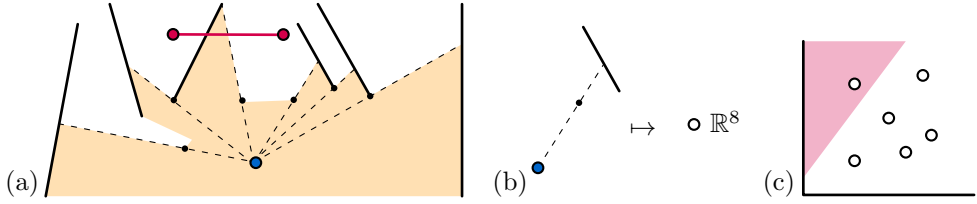
**Case 2: the entity  $r$  can cross a simple polygon.** If  $r$  is able to move through edges of  $P$  then its trajectory may intersect the boundary of  $P$  linearly often. Inspecting each of the resulting subsegments explicitly would thus require at least  $\Omega(n)$  time. Hence, we use a different approach. Let  $V_q$  denote the *visibility polygon* of point  $q$ : the set of all points visible from  $q$ . There is a time at which  $q$  can see  $r$  if and only if the trajectory of  $r$  intersects  $V_q$ . We build a data structure to find such an intersection.

Aronov et al. [13] developed an  $O(n^2)$  size data structure that can be built in  $O(n^2 \log n)$  time and can report the visibility polygon of an arbitrary query point  $q \in P$  in  $O(\log^2 n)$  time. The visibility polygon  $V_q$  is returned in its combinatorial representation, that is, as (a pointer to) a balanced binary search tree, storing the vertices of  $V_q$  in order along the boundary. It is important to note that this combinatorial representation does not explicitly store the locations of all vertices of  $V_q$ . Instead, a vertex  $v$  of  $V_q$  may be represented by a pair  $(e, w)$ , indicating that  $v$  is the intersection point of polygon edge  $e$  and the line through vertex  $w \in P$  and the query point  $q$  (Figure 5.16). If so desired, computing the explicit location of all vertices of  $V_q$  thus takes  $O(|V_q|)$  time by traversing the tree. We now extend the results of Aronov et al. in such a way that we can efficiently test if a line segment intersects  $V_q$  *without* spending the  $O(|V_q|)$  time to compute the explicit locations.

We briefly review the results of Aronov et al. first. They build a balanced hierarchical decomposition of  $P$  [53]. Each node  $v$  in the balanced hierarchical decomposition represents a subpolygon  $P_v$  of  $P$  (the root corresponds to  $P$  itself) and a diagonal of  $P_v$  that splits  $P_v$  into two roughly equal size subpolygons  $P_\ell$  and  $P_v \setminus P_\ell$ . For subpolygon  $P_\ell$  the data structure stores a planar subdivision  $S_\ell$  (of the area outside  $P_\ell$ ) such that for all points in a cell of  $S_\ell$  the part of the visibility polygon inside  $P_\ell$  has the same combinatorial representation. Moreover, for each cell it stores the corresponding combinatorial representation. These representations can be stored compactly by traversing  $S_\ell$  while maintaining (the representation of) the visibility polygon in  $P_\ell$  in a partially persistent red black tree [13]. The data structure stores an analogous subdivision for  $P_v \setminus P_\ell$ . The complete visibility polygon of  $q$  can be obtained by concatenating  $O(\log n)$  subchains of these pre-stored combinatorial representations (one from every level of the hierarchical decomposition).



**Figure 5.15** A simple polygon split in  $O(n^2)$  cells. For each cell, there exists a red-black tree that represents a visibility polygon. For each point  $q$  in a cell, its visibility polygon consists fixed vertices, and vertices defined by a reflex vertex of the polygon and an edge (the vertices at the end of the dashed segments).



**Figure 5.16** (a) We store the combinatorial structure of visibility polygon  $V_q$ . (b) Every variate vertex is defined by a reflex vertex of the polygon and an edge of the polygon. These parameters are mapped to a point in  $\mathbb{R}^8$ . (c) Given the query  $r$  and  $q$ , we map their variables to a range in  $\mathbb{R}^8$  such that edges attached to a variate vertex are intersected by  $r$  if and only if their points lie in the range.

We use the same approach as Aronov et al. [13], but we use a different representation of  $V_q$ . Our representation will be a weight-balanced binary search tree ( $BB[\alpha]$ -tree [168]) whose leaves store the vertices of  $V_q$  in order along the boundary. An internal node of this tree corresponds to a subchain of vertices along  $V_q$ , which is stored in an associated data structure. We distinguish two types of vertices in such a chain: *fixed vertices*, for which we know the exact location, and *variate vertices*, which are represented by an polygon-edge, polygon-vertex pair  $(e, w)$ . We store the fixed vertices in a linear size dynamic data structure that supports halfspace emptiness queries, that is, a dynamic convex hull data structure [34]. This data structure uses  $O(m)$  space, and supports  $O(\log m)$  time updates and queries, where  $m$  is the number of stored points. The variate vertices (Figure 5.15) are mapped to a point in  $\mathbb{R}^8$  using a function  $f$ . It is important to note that  $f$  is independent of  $q$ : so each vertex can be mapped to this point during preprocessing. We give the precise definition later. We store the resulting points in a dynamic data structure that can answer halfspace emptiness queries [4]. This data structure uses  $O(m \log m)$  space, answers queries in  $O(m^{\frac{3}{4}+\varepsilon})$  time (where  $\varepsilon$  is an arbitrarily small positive constant) and supports updates in  $O(\log^2 m)$  time, where  $m$  is the number of points stored. It follows that our representation of  $V_q$  uses  $O(n \log^2 n)$  space, and supports updates in amortized  $O(\log^3 n)$  time.

Since all nodes in the data structure have constant in-degree we can make it partially persistent at the cost of  $O(\log^3 n)$  space per update [76]. It follows we can represent the visibility polygons for all cells in a subpolygon  $S_\ell$  in  $O(n^2 \log^3 n)$  space.

**Querying.** Given a query point  $q$  and segment  $r$ , we test if the segment  $r$  intersects  $V_q$ . The core idea is as follows: we isolate the part of  $V_q$  that is intersected by the wedge defined by  $q$  and  $r$ . We then extend  $r$  into a line  $\rho$ , and test if this line intersects an edge of  $V_q$ . The segment  $r$  intersects  $V_q$  if and only if the line  $\rho$  intersects this subpolygon of  $V_q$ . Hence finding this intersection answers the visibility testing.

Since  $V_q$  is a star-shaped polygon, we can obtain the part of  $V_q$  that lies in the wedge defined by  $q$  and  $r$ , represented by  $O(\log^2 n)$   $BB[\alpha]$ -tree nodes. For each of these nodes we consider the line  $\rho$  and query for the following: we test if the halfspace bounded by  $\rho$  that does not contain  $q$  is empty (we denote this area by  $\rho^{-q}$ ):  $\rho$  intersects an edge of this subpolygon of  $V_q$  if and only if this is not the case. We can directly query the data structure storing the fixed vertices with this halfspace. To test if there is a variate vertex that lies in  $\rho^{-q}$ , we map it to a halfspace in  $\mathbb{R}^8$  using a function  $g$ .

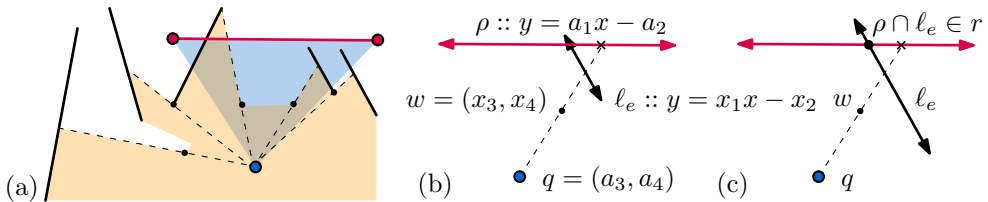
**Lemma 5.6** *There are functions  $f$  and  $g$  such that  $f$  maps each variate vertex  $(e, w)$  to a point  $f(e, w) \in \mathbb{R}^8$  and  $g$  maps each  $\rho^{-q}$  to a halfspace  $g(\rho^{-q})$  in  $\mathbb{R}^8$  such that  $f(e, w) \in g(\rho^{-q})$  if and only if the location of the variate vertex  $(e, w)$  in  $V_q$  lies in  $\rho^{-q}$ .*

**Proof** Let  $q = (a_3, a_4)$ , and let  $\rho = \{x, y \mid y = a_1x - a_2\}$  be the supporting line of the trajectory of  $r$ . We describe the construction for the case that  $q$  lies below  $\rho$  and  $\rho$  is non-vertical. The other cases can be handled analogously.

Refer to Figure 5.17 for an illustration of the proof. For each variate vertex  $(e, w)$  in a chain we know that the line  $qw$  intersects the line  $\ell_e$  supporting  $e$  on the domain of  $e$  (this property is guaranteed since  $(e, w)$  is a vertex of  $V_q$ ). Moreover, it is guaranteed that the intersection point between  $qw$  and  $\rho$ , lies on the trajectory  $r$  (this is because we algorithmically performed a binary search on  $V_q$  to make sure this is the case). It follows that  $q$  can see  $r$  if and only if, the intersection point  $(x, y)$  between  $qw$  and  $\ell_e$  lies above  $\rho$ . Given  $\rho, q, w$  and  $\ell_e$ , we can algebraically compute this intersection point  $(x, y)$ . We then substitute the equation for  $(x, y)$  into the equation for  $\rho$ , and the point  $(x, y)$  lies above this line if and only if the result is greater than 0:

$$wq := \left\{ x, y \mid 0 = \frac{x_4 - a_4}{x_3 - a_3}x - \frac{x_4 - a_4}{x_3 - a_3}x_3 + x_4 \right\}$$

The lines  $wq$  and  $\ell_e$  intersect at the point where their  $y$ -coordinate is equal:



**Figure 5.17** (a) A visibility polygon of  $q$  where two edges associated to variate vertices intersect the wedge. The remaining three variate vertices are not considered. (b) The query point  $q$ , variate point  $(w, e)$  and the line  $\rho$  that supports  $r$ . (c) We compute the point of intersection between  $\rho$  and  $\ell_e$ : the supporting line of  $e$ .

$$\begin{aligned}
x_1x - x_2 &= \frac{x_4 - a_4}{x_3 - a_3}x - \frac{x_4 - a_4}{x_3 - a_3}x_3 + x_4 \\
(x_3 - a_3)(x_1x - x_2) &= (x_4 - a_4)x - (x_4 - a_4)x_3 + (x_3 - a_3)x_4 \\
(x_3 - a_3)x_1x - (x_4 - a_4)x &= x_2(x_3 - a_3) - (x_4 - a_4)x_3 + (x_3 - a_3)x_4
\end{aligned}$$

From this equation we can extract the coordinates of the intersection point  $(x, y)$  between  $wq$  and  $\ell_e$ :

$$\begin{aligned}
x &= \frac{x_2(x_3 - a_3) - (x_4 - a_4)x_3 + (x_3 - a_3)x_4}{(x_3 - a_3)x_1 - (x_4 - a_4)} \\
y &= x_1 \frac{x_2(x_3 - a_3) - (x_4 - a_4)x_3 + (x_3 - a_3)x_4}{(x_3 - a_3)x_1 - (x_4 - a_4)} - x_2
\end{aligned}$$

Lastly we substitute the above algebraic expression for  $(x, y)$  into the formula for  $\rho$ . This gives us a predicate that determines whether the intersection between  $qw$  and  $\ell_e$  appears above or below  $\rho$ . We linearize the predicate predicate as follows:

$$\begin{aligned}
0 &\geq a_1(x_2(x_3 - a_3) - (x_4 - a_4)x_3 + (x_3 - a_3)x_4) - \\
&\quad x_2 - x_1(x_2(x_3 - a_3) - (x_4 - a_4)x_3 + (x_3 - a_3)x_4) + x_2 \\
0 &\geq [-a_1a_3](x_2) + [a_3](x_1x_2) + [a_1](x_2x_3) + [a_1a_4](x_3) + \\
&\quad [-a_4](x_1x_3) + [-a_1a_3](x_4) + [a_3](x_1x_4) + [-1](x_1x_2x_3)
\end{aligned}$$

Thus we found a predicate  $F(\vec{x}, \vec{a})$  with:

$$\begin{aligned}
(f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8) &= (x_2, x_1x_2, x_2x_3, x_3, x_1x_3, x_4, x_1x_4, x_1x_2x_3) \\
(g_0, g_1, g_2, g_3, g_4, g_5, g_6, g_7, g_8) &= (0, -a_1a_3, a_3, a_1, a_1a_4, -a_4, -a_1a_3, a_3, -1)
\end{aligned}$$

It follows that we can map every variate vertex to a point in  $\mathbb{R}^8$  using the  $f$ -maps provided by the predicate. Any query consisting of the halfplane  $\rho^{-q}$  defined by  $\rho$  and  $q$  gets mapped to a halfspace in  $\mathbb{R}^8$ . The halfplane  $\rho^{-q}$  contains the variate vertex defined by  $q$ ,  $w$ , and  $e$  if and only if its representative point lies in this halfspace. ■

This theorem now immediately follows:

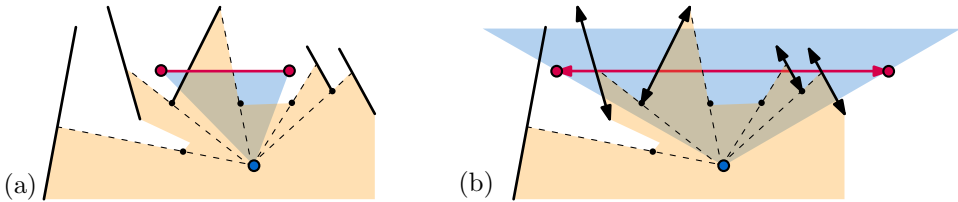
**Theorem 5.7** *Let  $P$  be a simple polygon with  $n$  vertices and  $\varepsilon > 0$  be an arbitrarily small constant. We can store  $P$  in a data structure of size  $O(n^2 \log^3 n)$  that allows us to answer trajectory visibility testing between a static and a linearly moving entity that may cross  $P$  in  $O(n^{\frac{3}{4}+\varepsilon})$  time. Building the data structure takes  $O(n^2 \log^3 n)$  time.*

**Moving through edges.** Next, we investigate the variants where the entities can walk through edges of a simple polygon  $P$ . We then note that all our techniques for detecting if a line segment intersects a visibility polygon were not dependent on the line  $r$  not being able to cross the domain  $P$ . Specifically our approach consists of two steps: testing if edges attached to fixed vertices are intersected and testing if edges attached to variate vertices are intersected by  $r$ . The first can still be done by storing for every visibility polygon the radial ordering of the fixed edges. The second approach (Figure 5.18) uses only the parameters of the supporting lines of involved polygon edges and these supporting lines already intersected the simple polygon  $P$ . Thus we immediately obtain the following result:

**Theorem 5.8** *Let  $P$  be a simple polygon with  $n$  vertices and  $\varepsilon > 0$  be an arbitrarily small constant. We can store  $P$  in a data structure of size  $O(n^2 \log^3 n)$  that allows us to answer trajectory visibility testing between a static and a linearly moving entity that may cross  $P$  in  $O(n^{\frac{3}{4}+\varepsilon})$  time. Building the data structure takes  $O(n^2 \log^3 n)$  time.*

**Case 3: the entities go through a polygonal domain.** Whenever  $P$  is a polygonal domain we cannot use the data structure by Aronov et al. [13] for simple polygons. To be able to use the same global approach, we use a much simpler but costly construction to obtain the visibility polygon  $V_q$  of our query point  $q$ . Instead of an efficient data structure that can retrieve visibility polygons, we explicitly build a subdivision  $S$  in which all points  $r$  in a cell have a visibility polygon  $V_r$  with the same combinatorial structure. To obtain  $S$  we simply take all  $O(n^2)$  lines defined by pairs of polygon vertices. We then traverse  $S$  while maintaining  $V_q$  in a partially persistent data structure. The subdivision  $S$  is the arrangement of these lines and has  $O(n^4)$  complexity. We obtain a traversal of  $S$  by computing an Euler tour of a spanning tree of the dual of  $S$ . We conclude:

**Theorem 5.9** *Let  $P$  be a polygonal domain with  $n$  vertices and  $\varepsilon > 0$  be an arbitrarily small constant.. We can store  $P$  in a data structure of size  $O(n^4 \log^3 n)$  that allows us to answer trajectory visibility testing between a static and a linearly moving entity that may cross  $P$  in  $O(n^{\frac{3}{4}+\varepsilon})$  time. Building the data structure takes  $O(n^4 \log^3 n)$  time.*



**Figure 5.18** (a) Earlier, we filtered for the subpolygon of  $V_q$  contained in an infinite wedge, and then filtered edges attached to variate vertices based on their parametrization. (b) Both steps do not depend on the  $r$  being disjoint from  $P$ .

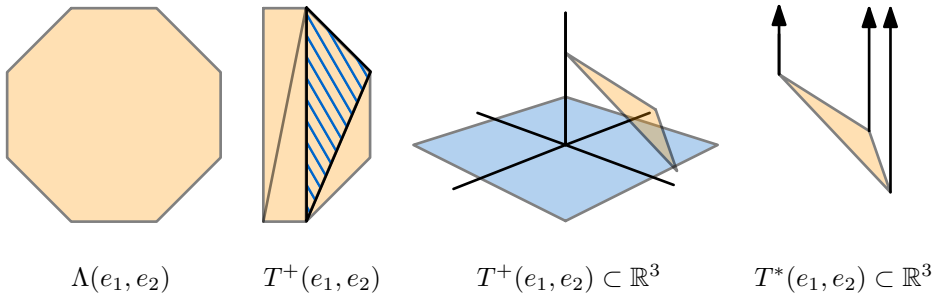
## 5.7 Two moving entities crossing a polygonal domain

Finally, we investigate the following variants where both  $p$  and  $q$  are moving:

1. both entities can walk through edges of a simple polygon  $P$ ,
2. both entities are contained within a polygonal domain  $P$ , and
3. both entities can walk through edges of a polygonal domain  $P$ .

We investigate these cases simultaneously to show that, even though this is the most general setting, it is possible to preprocess  $P$  so that given  $q$  and  $r$  we have sublinear query time. We wish to note that this section is more of a proof-of-concept, as the required space and preprocessing time will be too large to be practical. In previous sections we showed that the visibility glass (the collection of straight-line shortest paths) between two edges  $q$  and  $r$  could be dualized to a single convex, connected area. In all three scenarios in this section this is no longer true and this increases the difficulty of the problem. In this section, we use considerably more preprocessing time and space to store the visibility glass  $V(e_1, e_2)$  between any pair of edges  $e_1, e_2 \in P$ . We devise a generic projection (which we refer to as dualization) to dualize a line segment to a four-dimensional point instead of a line to a two-dimensional point. The information that we gain by not discarding endpoint information can be leveraged to solve all these three more complicated problem variants.

Let  $k$  be an unspecified integer constant. We prove that it is possible to solve visibility testing using halfspace range queries in  $\mathbb{R}^k$  among  $O(n^3)$  points. Using cutting trees [51] we can preprocess the  $O(n^3)$   $k$ -dimensional points using  $O(n^{3k})$  space and in  $O(n^{3k})$  time such that halfspace range queries (and thus our visibility testing queries) can be answered in  $O(\log^k n)$  time.



**Figure 5.19** We consider for two edges  $e_1, e_2$ , their visibility glass  $\Lambda(e_1, e_2)$  in the dual and map it  $\mathbb{R}^4$ . Specifically, we map not the whole polygon at once, but rather triangulate it and map each triangle of constant complexity separately. Since we cannot draw in  $\mathbb{R}^4$ , our illustration is constrained to lift it to  $\mathbb{R}^3$ .

**Storing visibility glasses for visibility testing.** Let  $P$  be a polygonal domain. We construct two near-identical data structures to test if  $q$  and  $r$  are mutually visible, where the line-of-sight has either positive or negative slope respectively. To any polygonal domain  $P$  we add an infinite size bounding box that contains the rest of  $P$ , consisting of four edges. Let  $t$  be a time when  $q$  and  $r$  are mutually visible and the line  $g(t)$  through  $q$  and  $r$  has positive slope. The segment between  $q$  and  $r$  must be contained in at least one visibility glass  $L(e_1, e_2)$  for two edges  $e_1, e_2 \in P$ . We denote by  $p_1$  and  $p_2$  the points of intersection between  $g(t)$  and  $e_1$  and  $e_2$  respectively.

Since  $q$  and  $r$  are both segments, we can find at most a constant number of time intervals where for each time  $t$  in the interval,  $q(t)$  is below  $r(t)$  or vice versa. We denote by  $x_p$  and  $y_p$  the respective  $x$ - and  $y$ -coordinate of a point  $p$ . We store a separate data structure for all four options based on whether  $y_{p_1} \leq y_{p_2}$  and  $r$  is below  $q$ , and query each structure with the appropriate segments. Henceforth, we assume that  $p_1$  has a lower  $y$ -coordinate than  $p_2$ , and for all times  $t$ ,  $q(t)$  has a lower  $y$ -coordinate than  $r(t)$ . It follows that (because  $g(t)$  has a positive slope)  $x_{p_1} \leq x_q \leq x_r \leq x_{p_2}$ .

This observation inspires the following approach illustrated in Figure 5.19: we consider for each pair of edges  $e_1, e_2 \in P$ , their dualized visibility glass  $\Lambda(e_1, e_2)$ , which we then triangulate. This results in a set  $\mathcal{T}$  of  $O(n^3)$  triangles in the dual. Let  $T \in \mathcal{T}$  be a triangle of  $\Lambda(e_1, e_2)$ , we lift  $T$  to a two-dimensional surface in  $\mathbb{R}^4$  with the map  $(a, b) \in T \mapsto (a, b, x_{p_1}, x_{p_2})$ , where now  $p_1$  is the intersection of  $e_1$  with the line  $y = ax - b$  and  $p_2$  is the intersection of  $e_2$  with the line  $y = ax - b$ . The two entities  $q$  and  $r$  have a unique associated curve segment  $\gamma$  which we also lift to  $\mathbb{R}^4$  with the map  $(a, b) \in \gamma \mapsto (a, b, x_q, x_r)$  where  $x_q$  is the  $x$ -coordinate of entity  $q$  at the time that realises the point  $(a, b)$  on  $\gamma$ , and  $x_r$  is defined symmetrically.

**Lemma 5.7** *Entities  $q$  and  $r$  are mutually visible, if and only if their 4-dimensional curve intersects a 4-dimensional volume that is bound by a triangle  $T \in \mathcal{T}$  (lifted to four dimensions) and two orthogonal 3-dimensional hyperplanes.*

**Proof** Let there be a time  $t$  where  $q$  and  $r$  are mutually visible. The line extending  $g(t)$  must, when dualized, be contained in a triangle  $T \in \mathcal{T}$  formed by edges  $e_1, e_2$ , with intersection points  $p_1$  and  $p_2$ . Per assumption,  $q(t)$  has a lower  $y$ -coordinate than  $r(t)$  and thus:  $x_{p_1} \leq x_q \leq x_r \leq x_{p_2}$ . It now immediately follows that the point corresponding to  $g(t)$  when lifted to  $\mathbb{R}^4$  is in the volume corresponding to  $T$ .

Similarly let there be a time  $t$  where  $g(t)$  mapped to  $\mathbb{R}^4$  is contained the volume corresponding to some triangle  $T$  in the dual, with associated edges  $e_1, e_2$  which are intersected by the corresponding line in the points  $p_1$  and  $p_2$ . The points  $p_1$  and  $p_2$  are mutually visible. Per definition, the supporting line of the segment between  $q(t)$  and  $r(t)$  coincides with the line through  $p_1$  and  $p_2$ . Per construction of the mapping, the segment itself is contained in the segment between  $p_1$  and  $p_2$ . ■

For each of these  $O(n^3)$  constant-complexity volumes, we can create a constant-description predicate for intersection that can be linearized to  $k$  terms for some constant  $k$ . And we thus conclude:

**Theorem 5.10** *Let  $P$  be a polygonal domain with  $n$  vertices. We can store  $P$  in a data structure of size  $O(n^{3k})$ , for some sufficiently large constant  $k$ , that allows us to answer trajectory visibility testing in  $O(\log^k n)$  time. Building the data structure takes  $O(n^{3k})$  time.*

## 5.8 Concluding remarks

In this chapter we studied how to efficiently compute the visibility between two entities that each traverse a segment trajectory in a polygonal domain. Specifically, we studied how (given a polygonal domain  $P$  of  $n$  vertices and two segment trajectories) to efficiently decide if there is a time where the two entities that traverse the segment trajectories are mutually visible. In addition, we studied the special case where either of the two segment trajectories is a single point.

We first showed how to answer visibility testing in the algorithmic (one-shot) variant, where the input is the polygonal domain  $P$  together with two segment trajectories. For all studied variants of this problem, we showed an algorithmic solution which we prove is worst case optimal. Our approach relied on a novel dualisation that maps the line-of-sight between the two entities to an algebraic curve (segment) of bounded degree. We leveraged the properties of this algebraic curve (and the dualisation of the line-of-sight between two segments) to construct our algorithm.

Second, we studied the data structure variant of these problems where we are allowed to preprocess the polygonal domain  $P$ . The majority of this chapter focused on the special case where  $P$  is a simple polygon. We deviated from the classical event-based approaches that (kinetically) maintain the visibility between moving entities. Instead, we again relied on constructing algebraic predicates that decide visibility between the two entities. This shift in the approach for solving visibility between moving entities allowed us to obtain the first sublinear query times for visibility between moving entities in a simple polygon (or polygonal domain). Throughout this chapter, the algebraic analysis that support our results was not overly involved: when one entity is stationary we essentially only parametrize the point of intersection between a line and a line segment, and in the case where both entities are moving we parametrized their line-of-sight as a hyperbolic curve segment. It is important to note that the application of the linearization technique is not overly complex, because at times the technique itself may be considered ‘cumbersome’, or even ‘overkill’.

We wish to note that as the data structure problem becomes more involved (i.e. as we move from a simple polygon to a polygonal domain, or from visibility between a point and an entity to visibility between two moving entities) our preprocessing time and required space moves from computationally viable to high-polynomial. That is, a computer may be able to preprocess a simple polygon in  $O(n^2 \text{ polylog } n)$  time and space; however, the  $O(n^4 \text{ polylog } n)$  (or greater) time and space required for preprocessing a polygonal domain may be considered computationally infeasible. We therefore wish to note that our results for when  $P$  is a polygonal domain, may be best regarded as a *proof of concept* that sublinear query time is indeed possible.

Finally, we wish to review possible directions for future research. First, it would be interesting to see if the preprocessing time and space required to answer visibility testing queries in a polygonal domain can be improved. When  $P$  is a simple polygon, our approach relies on obtaining the shortest path between two points in  $P$  as a canonical subset, and to transform this into what we call a visibility glass that represents all lines-of-sight between the two line segments. This approach does not transfer well to when  $P$  is a polygonal domain. This is because within the visibility glass between two entities, there may be a linear number of components of the polygonal domain. To efficiently solve visibility testing in this setting, one would need a radically different approach.

Another interesting future research direction is to study more variants of the visibility testing data structure problem. In this chapter we preprocessed a polygonal domain  $P$  subject to visibility testing queries between two trajectories  $p$  and  $q$ . A natural variant of the problem is to preprocess a polygonal domain  $P$  and some trajectory  $p$ , subject to visibility testing between  $p$  and some query trajectory  $q$ . As a last suggestion, we propose to further apply and develop the linearization technique used in this chapter. Classically, the linearization technique was used to decide halfspace containment queries. This chapter is an example of how, with some additional observations, the technique can be used to decide intersection queries between either a line and a line segment, or a hyperbolic curve and a simple polygon. It is not inconceivable, or it is perhaps even likely, that more ‘traditional’ geometric problems can be reformulated into an intersection query that can be efficiently answered through a combination of geometric observations and linearization.

## 5.A Transforming two entities into an algebraic curve

Throughout this chapter, we study the line-of-sight between two entities that each move along a linear trajectory (possibly at different but constant speeds) during the time  $t \in [0, 1]$ . Consider the line  $g(t)$  through the two entities at time  $t$ . We can dualize  $g(t)$  to a point using classical point-line dualization. In Section 5.4 we claim that the continuous dualization of the line through the two entities traces a degree-2 curve segment denoted by  $\gamma$  in the dual. Here we show why this is the case:

**Lemma 5.1** *The segment  $\gamma$  is a segment of a quadratic curve that has 5 degrees of freedom.*

**Proof** For algebraic convenience we say that entity  $q$  walks from  $(a_1, a_2)$  to the point  $(a_1 + a_3, a_2 + a_4)$  and that entity  $r$  walks from  $(a_5, a_6)$  to  $(a_5 + a_7, a_6 + a_8)$ . Note that the speed of entity  $q$  is  $\|(a_3, a_4)\|$  and that the speed of entity  $r$  is  $\|(a_7, a_8)\|$ . We can parametrize the position of entity  $q$  and  $r$  at time  $t \in [0, 1]$  as follows:

$$q(t) = \begin{pmatrix} x_{q(t)} \\ y_{q(t)} \end{pmatrix} = \begin{pmatrix} a_1 + a_3t \\ a_2 + a_4t \end{pmatrix} \quad r(t) = \begin{pmatrix} x_{r(t)} \\ y_{r(t)} \end{pmatrix} = \begin{pmatrix} a_5 + a_7t \\ a_6 + a_8t \end{pmatrix} \quad (5.1)$$

At all times, the line  $g(t)$  is the line through the points  $q(t)$  and  $r(t)$ . We say that at all times,  $g(t)$  has slope and offset  $(\alpha(t), \beta(t))$ . The parametrization of  $g(t)$  then becomes:

$$g(t) = \begin{pmatrix} \alpha(t) \\ \beta(t) \end{pmatrix} = \begin{pmatrix} \frac{y_{r(t)} - y_{q(t)}}{x_{r(t)} - x_{q(t)}} \\ \alpha(t) \cdot x_{q(t)} - y_{q(t)} \end{pmatrix} = \begin{pmatrix} \frac{a_6 - a_2 + (a_8 - a_4)t}{a_5 - a_1 + (a_7 - a_3)t} \\ \alpha(t)(a_1 + a_3t) - a_2 - a_4t \end{pmatrix} \quad (5.2)$$

If the time  $t$  lies between 0 and 1, this parametric equation traces our curve segment  $\gamma$  and if we take  $t$  over all of  $\mathbb{R}$ , the parametric equation traces a full curve which we denote by  $\Gamma$ . To show the degree of the curve  $\Gamma$  we rewrite the parametrized curve to a canonical form where we lose the dependence on  $t$ . First we take the formula for the  $\beta$ -coordinate and isolate  $t$ :

$$t = \frac{\alpha(t)a_1 - a_2 - \beta(t)}{a_4 - \alpha(t)a_3}$$

We then take the formula for the  $\alpha$ -coordinate and remove the fraction by multiplying both sides with  $((a_5 - a_1) + (a_7 - a_3)t)$ . Note that this expression is only zero if the line  $g(t)$  is vertical. Refer to the next section on how to avoid such degeneracies.

$$\alpha(t)(a_5 - a_1) + \alpha(t)(a_7 - a_3)t = (a_6 - a_2) + (a_8 - a_4)t$$

We substitute the value for  $t$  into this equation, and remove the fraction by multiplying both sides with  $(\alpha(t)a_3 + a_4)$ :

$$\begin{aligned} \alpha(t)(a_5 - a_1) + \alpha(t)(a_7 - a_3) \left( \frac{\alpha(t)a_1 - a_2 - \beta(t)}{a_4 - \alpha(t)a_3} \right) &= \\ (a_6 - a_2) + (a_8 - a_4) \left( \frac{\alpha(t)a_1 - a_2 - \beta(t)}{a_4 - \alpha(t)a_3} \right) &\Rightarrow \\ \alpha(t)(a_5 - a_1) \cdot (a_4 - \alpha(t)a_3) + \alpha(t)(a_7 - a_3) \cdot (\alpha(t)a_1 - a_2 - \beta(t)) &= \\ (a_6 - a_2) \cdot (a_4 - \alpha(t)a_3) + (a_8 - a_4) \cdot (\alpha(t)a_1 - a_2 - \beta(t)) &\Rightarrow \\ -\alpha(t)^2 a_3(a_5 - a_1) + \alpha(t)a_4(a_5 - a_1) + & \\ \alpha(t)^2 a_1(a_7 - a_3) - \alpha(t)a_2(a_7 - a_3) - \alpha(t)\beta(t)(a_7 - a_3) &= \\ -\alpha(t)a_3(a_6 - a_2) + a_4(a_6 - a_2) + \alpha(t)a_1(a_8 - a_4) - a_2(a_8 - a_4) - \beta(t)(a_8 - a_4) & \end{aligned}$$

Lastly we show that this equation provides a linearization as defined in Section 5.2 by separating polynomials based on  $\alpha(t)$  and  $\beta(t)$  from polynomials based on  $a_1 \dots a_8$ .

$$\begin{aligned}
& [\alpha(t)^2](a_1(a_7 - a_3) - a_3(a_5 - a_1)) + \\
& [\alpha(t)](a_4(a_5 - a_1) - a_2(a_7 - a_3) + a_3(a_6 - a_2) - a_1(a_8 - a_4)) + \\
& [-\alpha(t)\beta(t)](a_7 - a_3) + [\beta(t)](a_8 - a_4) + [1](a_2(a_8 - a_4) - a_4(a_6 - a_2))
\end{aligned}$$

This gives us the following linearization where  $k = 4$ :

$$\begin{aligned}
g_0(\vec{a}) &= a_2(a_8 - a_4) - a_4(a_6 - a_2) & f_1(t) &= \beta(t), \\
g_1(\vec{a}) &= a_8 - a_4, & f_2(t) &= -\alpha(t)\beta(t), \\
g_2(\vec{a}) &= a_7 - a_3, & f_3(t) &= \alpha(t), \\
g_3(\vec{a}) &= a_4(a_5 - a_1) - a_2(a_7 - a_3) + a_3(a_6 - a_2) - a_1(a_8 - a_4), & f_4(t) &= \alpha(t)^2. \\
g_4(\vec{a}) &= a_1(a_7 - a_3) - a_3(a_5 - a_1),
\end{aligned}$$

The degrees of freedom is defined as  $k$  plus one and this concludes the proof. ■

## 5.B Dealing with degeneracies

Observe that in Equation 5.2 it is possible to divide by zero. Note that this occurs only if there is a moment in time where the line-of-sight between the two entities is a vertical segment. This situation occurs throughout this chapter and in this case the dual of their line-of-sight is also not well defined. Generally we cannot dualize a visibility glass if in the primal it contains any vertical lines.

This is a common degeneracy with visibility queries and dualization algorithms in computational geometry and it can be solved as follows: For any two query segments segment one can split the time interval into two disjoint intervals, such that in the first interval the line-of-sight between them is never vertical and in the second interval the line-of-sight between them is never horizontal. Note that only one split is needed, which can be calculated in constant time, because the entities move linearly. Therefore we solve the algorithmic question or the data structure question by solving two separate inputs or queries, where for the time interval that can contain vertical but not horizontal lines-of-sight, we consider a rotated version of the plane. Similarly, whenever we consider a visibility glass, we split it into lines that are steeper than  $y = x$  and lines that are not. One such set will never contain horizontal lines and the other will never contain vertical lines. Therefore, for each set of lines we can construct the dual of the visibility glass in an appropriately rotated version of the plane.

## Chapter Six

# Fréchet distance between trajectories

In this chapter we consider a polygonal curve  $P$  of linear complexity. We are interested in preprocessing  $P$ , such that for a query segment  $\overline{ab}$  we can determine the Fréchet distance between  $P$  and  $\overline{ab}$  efficiently.

## 6.1 Introduction

Comparing the shape of polygonal curves is an important task that arises in many contexts such as GIS applications [8, 37], protein classification [120], curve simplification [36], curve clustering [6] and even speech recognition [136]. Within computational geometry, there are two well studied distance measures for polygonal curves: the Hausdorff and the Fréchet distance. The Fréchet distance has proven particularly useful as it takes the course of the curves into account. However, the Fréchet distance between curves is costly to compute as its computation requires roughly quadratic time [9]. When a large number of Fréchet distance queries are required, we would like to have a data structure to answer these queries more efficiently. One such setting is when we are given a long curve  $P$  and we want to reason about shortcuts of  $P$  that have small distance to the subtrajectory they shortcut [74].

Here we study the problem of preprocessing a polygonal curve  $P$  to determine the Fréchet distance between  $P$  and a query segment. Specifically, we study preprocessing a polygonal curve  $P$  of  $n$  vertices in the plane, such that given a query segment  $\overline{ab}$ , traversed from  $a$  to  $b$ , the Fréchet distance between  $P$  and  $\overline{ab}$  can be computed in sublinear time. Note that without preprocessing, this problem can be solved in  $O(n \log n)$  time with the algorithm by Alt and Godau [9].

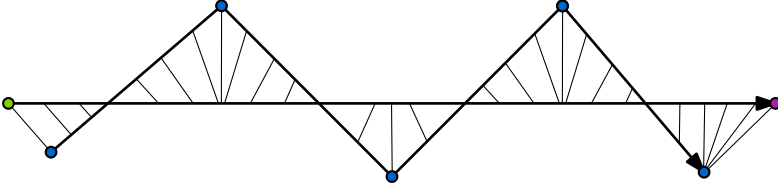
**Related work.** Data structures that support (approximate) nearest neighbor queries with respect to the Fréchet distance have received considerable attention throughout the years, see for instance the two recent papers by Driemel and Psarros[75] and by Filtser, Filtser and Katz [89] and the references therein. In exact Fréchet distance problems the goal is typically to store a set of polygonal curves such that given a query curve and a query threshold  $\Delta$  one can quickly report (or count) the curves that are within (discrete) Fréchet distance  $\Delta$  of the query curve. In approximation problems there is some fixed  $\varepsilon > 0$  and you are additionally allowed to report (or count) curves within distance  $(1 + \varepsilon)\Delta$ . Some of these data structures even allow approximately counting the number of curves that have a subcurve within Fréchet distance  $\Delta$  [62]. Also highlighting its practical importance, the nearest neighbor problem using Fréchet distance was posed as ACM Sigspatial GIS Cup in 2017 [201].

Here, we consider the problem of computing the exact Fréchet distance of (part of) a curve to a query segment. Driemel and Har-Peled [74] present an  $O(n\varepsilon^{-4} \log \varepsilon^{-1})$  size data structure that given a query segment  $\overline{ab}$  can compute a  $(1 + \varepsilon)$ -approximation of the Fréchet distance between  $P$  and  $\overline{ab}$  in  $O(\varepsilon^{-2} \log n \log \log n)$  time. Gudmundsson et al. [105] present an  $O(n \log n)$  size data structure that can *decide* if the Fréchet distance to  $\overline{ab}$  is smaller than a given value  $\Delta$  in  $O(\log^2 n)$  time. However, their result holds only when the length of  $\overline{ab}$  and all edges in  $P$  is relatively large compared to  $\Delta$ . De Berg et al. [63] presented an  $O(n^2)$  size data structure that does not have any restrictions on the length of the query segment or the edges of  $P$ . However, the orientation of the query segment is restricted to be horizontal. Queries are supported in  $O(\log^2 n)$  time. They extend their queries to vertex-to-vertex subcurves of  $P$ . That is, we denote as query input by  $P[s, t]$  the subcurve of  $P$  in between  $s$  and  $t$  (if the curve is self intersecting, we assume that the query somehow specifies the intended subcurve). De Berg et al. compute the Fréchet distance between the query segment and  $P[s, t]$  in  $O(\log^2 n)$  time, using  $O(n^2 \log^2 n)$  space. Very recently, Gudmundsson et al. [106] extended this result to allow the subcurve to start and end anywhere within  $P$ . Their data structure has size  $O(n^2 \log^2 n)$ , and queries take  $O(\log^8 n)$  time. Our results improve and extend these results, as we significantly decrease the space usage and the query times, as well as present data structures that allow arbitrarily oriented query segments.

**Problem statement & results.** Let  $P$  be a polygonal curve in  $\mathbb{R}^2$  with  $n$  vertices  $p_1, \dots, p_n$ . We assume that the vertices of  $P$  are in general position, i.e., all  $x$ - and  $y$ -coordinates are unique, no three points lie on a line, and no four points are cocircular. We consider  $P$  as a function mapping any time  $t \in [0, 1]$  to a point  $P(t)$  in the plane. Our ultimate goal is to store  $P$  such that we can quickly compute the *Fréchet distance*  $D_{\mathbb{F}}(P, Q)$  between  $P$  and a query curve  $Q$ . The Fréchet distance is defined as

$$D_{\mathbb{F}}(P, Q) = \inf_{\alpha, \beta} \max_{t \in [0, 1]} \|P(\alpha(t)) - Q(\beta(t))\|,$$

where  $\alpha, \beta: [0, 1] \rightarrow [0, 1]$  are nondecreasing surjections (also called reparameterizations of  $P$  and  $Q$ ) respectively and for any two points  $p, q$ ,  $\|p - q\|$  denotes the Euclidean distance between  $p$  and  $q$ . See Figure 6.1.



**Figure 6.1** Given a polygonal curve  $P$  of blue vertices and a polygonal curve  $Q$  we look for a continuous matching between points on  $P$  and points on  $Q$ .

In this chapter we focus on the case where  $Q$  is a single line segment  $\overline{ab}$  starting at  $a$  and ending at  $b$ . Note that  $P$  may self-intersect and  $\overline{ab}$  may intersect  $P$ .

Our first main result deals with the case where  $\overline{ab}$  is horizontal:

**Theorem 6.1** *Let  $P$  be a polygonal curve in  $\mathbb{R}^2$  with  $n$  vertices. There is an  $O(n \log n)$  size data structure that can be built in  $O(n \log^2 n)$  time such that given a horizontal query segment  $\overline{ab}$  it can report  $\mathbf{D}_{\mathbb{F}}(P, \overline{ab})$  in  $O(\log n)$  time.*

This significantly improves over the earlier result by de Berg et al. [63], as we reduce the required space and preprocessing time from quadratic to near linear. We simultaneously improve the query time from  $O(\log^2 n)$  to  $O(\log n)$ .

We further extend our results to allow queries where the input includes two points  $s, t$  on  $P$  that bound some subcurve  $P[s, t]$  (if  $P$  is self-intersecting, we assume the query somehow specifies which subcurve is between  $s$  and  $t$  and that we receive additional pointers to the edges of  $P$  that contain  $s$  and  $t$ ). We denote by  $|P[s, t]|$  the number of vertices on  $P[s, t]$  and we show the following result:

**Theorem 6.2** *Let  $P$  be a polygonal curve in  $\mathbb{R}^2$  with  $n$  vertices. There is an  $O(n \log^2 n)$  size data structure that can be built in  $O(n \log^2 n)$  time such that given a horizontal query segment  $\overline{ab}$  and two query points  $s$  and  $t$  on  $P$  the data structure can report  $\mathbf{D}_{\mathbb{F}}(P[s, t], \overline{ab})$  in  $O(\log^3 |P[s, t]|)$  time.*

De Berg et al. presented a data structure that could handle such queries in  $O(\log^2 n)$  time (using  $O(n^2 \log^2 n)$  space), provided that  $s$  and  $t$  were vertices of  $P$ . Compared to their data structure we again significantly improve the space usage, while allowing more general queries but possibly slower queries. The recently presented data structure of Gudmundsson et al. [106] does allow  $s$  and  $t$  to lie on the interior of edges of  $P$  (and thus supports queries against arbitrary subcurves). Their data structure uses  $O(n^2 \log^2 n)$  space and allows for  $O(\log^8 n)$  time queries. Compared to their result we again use significantly less space, while also improving the query time.

Using the insights gained in this restricted setting, we then present the first data structure that allows exact Fréchet distance queries with arbitrarily oriented query segments in sublinear time. At only a small additional cost we can also support subcurve queries and we obtain the following:

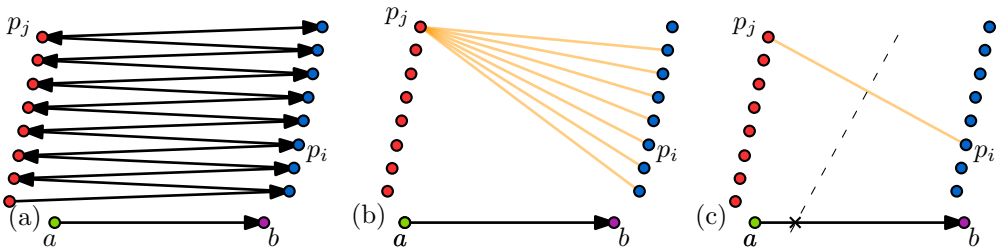
**Theorem 6.3** Let  $P$  be a polygonal curve in  $\mathbb{R}^2$  with  $n$  vertices, and let  $\varepsilon > 0$  be an arbitrarily small constant. There is an  $O(n^{4+\varepsilon})$  size data structure that can be built in  $O(n^{4+\varepsilon})$  time such that given an arbitrary query segment  $\overline{ab}$  and two query points  $s$  and  $t$  on  $P$  it can report  $D_{\mathbb{F}}(P[s, t], \overline{ab})$  in  $O(\log^4 |P[s, t]|)$  time.

To achieve our results, we also develop data structures that allow us to efficiently query the directed Hausdorff distance  $\vec{D}_H(P[s, t], \overline{ab}) = \max_{p \in P[s, t]} \min_{q \in \overline{ab}} \|p - q\|$  from (a subcurve  $P[s, t]$  of)  $P$  to the query segment  $\overline{ab}$ . For an arbitrarily oriented query segment  $\overline{ab}$  and a query subcurve  $P[s, t]$  our data structure uses  $O(n \log n)$  space and can answer such queries in  $O(\log^2 n)$  time. Using more space, queries can be answered in  $O(\log n)$  time, see Section 6.5.

Finally, in Section 6.7 we describe two problems that we can solve efficiently using our data structure. First, we show how to compute a local  $\delta$ -simplification of  $P$ —that is, a minimum complexity curve whose edges are within Fréchet distance  $\delta$  to the corresponding subcurve of  $P$ —in  $O(n^{5/2+\varepsilon})$  time. This improves existing  $O(n^3)$  time algorithms [101]. Second, given a query segment  $\overline{ab}$  we show how to efficiently find a translation of  $\overline{ab}$  that minimizes the Fréchet distance to (a given subcurve of)  $P$ . This extends the work of Gudmundsson et al. [107] to arbitrarily oriented segments.

## 6.2 Global approach

We illustrate the main ideas of our approach, in particular for the case where the query segment  $\overline{ab}$  is horizontal, with  $a$  left of  $b$ . We can build a symmetric data structure in case  $a$  lies right of  $b$ . We now first review some definitions based on those in [63].



**Figure 6.2** (a) A polygonal curve and query segment. (b) The red vertex  $p_j$  forms a backward pair with all but one blue vertex. (c) For a fixed backward pair  $(p_i, p_j)$ , we consider the distance between the intersection (cross) of their bisector (dashed) and  $\overline{ab}$ , and either  $p_i$  or  $p_j$ .

Let  $P^\leq \subset P \times P$  be the set of ordered pairs of vertices where for each pair  $(p, q) \in P^\leq$ ,  $p$  precedes or equals  $q$  along  $P$ . An ordered pair  $(p, q) \in P^\leq$  forms a *backward pair* if  $x_q \leq x_p$ . Here, and throughout the rest of the chapter,  $x_p$  and  $y_p$  denote the  $x$ - and  $y$ -coordinates of point  $p$ , respectively. The set of all backward pairs of  $P$  will be denoted  $\mathcal{B}(P)$ . A backward pair  $(p, q)$  is *trivial* if  $p = q$ . See Figure 6.2 for an example of backward pairs (omitting trivial pairs).

For two points  $p, q \in P$ , we then define  $\delta_{pq}(y) = \min_x \max \{ \|(x, y) - p\|, \|(x, y) - q\| \}$ . That is,  $\delta_{pq}(y)$  is a function that for any  $y$ -coordinate gives the minimum possible distance between a point at height  $y$  and both  $p$  and  $q$ . We will use the function  $\delta_{pq}$  only when  $(p, q) \in \mathcal{B}(P)$  is a backward pair. We then define the function  $\mathcal{D}_B(y) = \max \{ \delta_{pq}(y) \mid (p, q) \in \mathcal{B}(P) \}$ , which we refer to as the *backward pair distance* of a horizontal segment at height  $y$  with respect to  $P$ . Note that  $\mathcal{D}_B(y)$  is the upper envelope of the functions  $\delta_{pq}$  for all backward pairs  $(p, q)$  of  $P$ .

De Berg et al. [63] prove for horizontal query segments  $\overline{ab}$  that the Fréchet distance to a polygonal curve  $P$  is the maximum of four terms:

$$\mathbf{D}_{\mathbb{F}}(P, \overline{ab}) = \max \left\{ \|p_1 - a\|, \|p_n - b\|, \overrightarrow{\mathbf{D}}_H(P, \overline{ab}), \mathcal{D}_B(y_a) \right\}. \quad (6.1)$$

Given  $P$  and the points  $a$  and  $b$ , the first two terms are trivial to compute in  $O(1)$  time. Like de Berg et al., we build separate data structures that allow us to efficiently compute the third and fourth terms.

A key insight in our approach is that we can compute the directed Hausdorff distance  $\overrightarrow{\mathbf{D}}_H(P, \overline{ab})$  by building the furthest segment Voronoi diagrams (FSVD) of two sets of horizontal halflines, and querying these diagrams with the endpoints  $a$  and  $b$ . See Section 6.3.1. This allows for a linear space data structure that supports querying  $\overrightarrow{\mathbf{D}}_H(P, \overline{ab})$  in  $O(\log n)$  time, improving both the space and query time over [63].

However, in [63] the data structure that supports computing the backward pair distance dominates the required space and preprocessing time required to compute the directed Hausdorff distance. Note that there exist trajectories  $P$  where there are  $\Omega(n^2)$  backward pairs (see Figure 6.2).

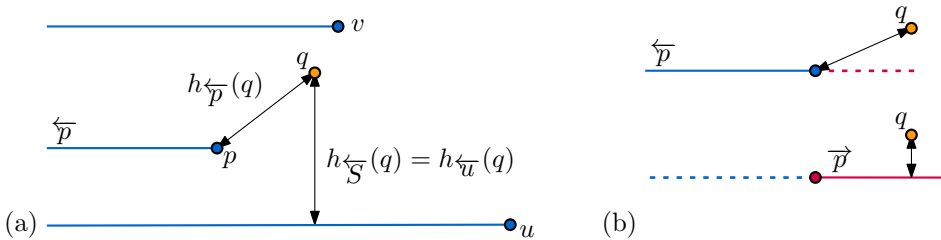
**The approach for efficiently computing backward pair distance.** Let  $\overline{ab}$  be a query segment which is known to be horizontal. In Section 6.3.2 we show that the number of backward pairs that show up on the upper envelope  $\mathcal{D}_B$  is only  $O(n \log n)$  despite there being possibly  $O(n^2)$  backward pairs. The crucial ingredient is that there are only  $O(n)$  backward pairs  $(p, q)$  contributing to  $\mathcal{D}_B$  in which  $p$  is a vertex among the first  $n/2$  vertices of  $P$ , and  $q$  is a vertex in the remaining  $n/2$  vertices. Surprisingly, we can again argue this using furthest segment Voronoi diagrams of sets of horizontal halflines. This allows us to build a search structure over the function  $\mathcal{D}_B$  in  $O(n \log^2 n)$  time. In Section 6.4 we show that we can extend these results to support queries against an arbitrary subcurve  $P[s, t]$  of  $P$ .

For arbitrarily oriented query segments we similarly decompose  $D_{\mathbb{F}}(P, \overline{ab})$  into four terms, and build a data structure for each term separately, see Section 6.5. The directed Hausdorff term can still be queried efficiently using an  $O(n \log^2 n)$  size data structure. However, our initial data structure for the backward pair distance uses  $O(n^{4+\epsilon})$  space. The main reason for this is that functions  $\delta_{pq}$  expressing the cost of a backward pair are now bivariate, depending on both the slope and intercept of the supporting line of  $\overline{ab}$ . The upper envelope of a set of  $n$  such functions may have quadratic complexity. While our divide and conquer strategy does not help us to directly bound the complexity of the (appropriately generalized function)  $\mathcal{D}_B$  in this case, it does allow us to support queries against subcurves of  $P$ . Moreover, we can use it to obtain a query time vs. space trade off. In Section 6.7 we then apply our data structure to efficiently solve various Fréchet distance related problems.

### 6.3 Horizontal queries

Let  $\overline{ab}$  be an arbitrary horizontal query segment that represents a trajectory that is traversed from  $a$  to  $b$ . We show how to preprocess  $P$  to compute  $D_{\mathbb{F}}(P, \overline{ab})$  efficiently. First, we introduce some definitions that will be used throughout the chapter.

For a point  $p \in \mathbb{R}^2$ , define  $\overleftarrow{p}$  to be the “leftward” horizontal halfline starting at  $p$  and containing all points left of  $p$  (refer to Figure 6.3). Analogously, we define  $\overrightarrow{p}$  as the “rightward” horizontal halfline starting at  $p$ , so that  $p = \overleftarrow{p} \cap \overrightarrow{p}$ . We extend this notation to any set of points  $S$ , that is,  $\overleftarrow{S} = \{\overleftarrow{s} \mid s \in S\}$  denotes the set of “leftward” halflines starting at the points in  $S \subseteq \mathbb{R}^2$ . We define  $\overrightarrow{S}$  analogously.



**Figure 6.3** (a) A collection  $S$  of three points:  $S := \{p, u, v\}$ . The set  $\overleftarrow{S}$  is the set of leftward pointing halflines whose apex lie on a point of  $S$ . The Hausdorff distance from a point  $q$  to a halfline may be the distance to the apex, or the vertical distance between the point and the line supporting the halfline. (b) the distance between a point  $q$  and  $p$  is the maximum of the distance to the left and right halfline from  $p$ .

### 6.3.1 The Hausdorff term

First, we show how to construct a data structure in  $O(n \log n)$  time such that given  $\overline{ab}$  we can compute  $\vec{D}_H(P, \overline{ab})$  in  $O(\log n)$  time. Let  $S$  and  $T$  be two (not necessarily disjoint) point sets in the plane. We define the following distance functions for rays  $\overleftarrow{p}$ ,  $\overleftarrow{S}$  (definitions for  $\overrightarrow{p}$ ,  $\overrightarrow{S}$  are analogous):

$$h_{\overleftarrow{p}}(q) = \vec{D}_H(\{q\}, \overleftarrow{p}) = \min \{\|p' - q\| \mid p' \in \overleftarrow{p}\}, \quad h_{\overleftarrow{S}}(q) = \max \{h_{\overleftarrow{p}}(q) \mid p \in S\}$$

Note that for any set  $S$  the function  $h_{\overleftarrow{S}}$  (resp.,  $h_{\overleftarrow{S}}$ ) is the upper envelope of the distance functions to the halflines in  $\overleftarrow{S}$  (resp.,  $\overleftarrow{S}$ ). Since  $h_{\overleftarrow{S}}$  and  $h_{\overleftarrow{S}}$  map each point in the plane to a distance, the envelopes live in  $\mathbb{R}^3$ . We observe the following property of this upper envelope:

**Observation 6.1** *Let  $S$  be a set of points in the plane. The (graph of the) function  $h_{\overleftarrow{S}}$  is an upper envelope whose orthogonal projection is the furthest segment Voronoi diagram of the set of halflines  $\overleftarrow{S}$ . The same property holds for  $h_{\overleftarrow{S}}$ .*

Since  $\overleftarrow{p}$  and  $\overrightarrow{p}$  are horizontal halflines we have for any two points  $p = (x_p, y_p)$  and  $q = (x_q, y_q) \in \mathbb{R}^2$  the following property (refer to Figure 6.3(a)):

$$h_{\overleftarrow{p}}(q) = \begin{cases} \|p - q\| & \text{if } x_p \leq x_q \\ |y_p - y_q| & \text{if } x_p \geq x_q, \end{cases} \quad \text{and} \quad h_{\overrightarrow{p}}(q) = \begin{cases} \|p - q\| & \text{if } x_p \geq x_q \\ |y_p - y_q| & \text{if } x_p \leq x_q. \end{cases}$$

This property implies the following observation:

**Observation 6.2** *For any fixed  $y$  and  $p \in S$ , the function  $x \mapsto h_{\overleftarrow{p}}(x, y)$  for a point  $p$  is monotonically increasing, and  $x \mapsto h_{\overrightarrow{p}}(x, y)$  is monotonically decreasing. Consequently, also for any point set  $S$ , the function  $x \mapsto h_{\overleftarrow{S}}(x, y)$  is monotonically decreasing, and  $x \mapsto h_{\overrightarrow{S}}(x, y)$  is monotonically increasing.*

Suppose that we have two functions  $x \mapsto h_{\overleftarrow{S}}(x, y)$  and  $x \mapsto h_{\overrightarrow{S}}(x, y)$  (for some fixed value of  $y$ ). Since the first function is monotonically decreasing and the second is monotonically increasing, these two functions intersect in a single connected interval. Henceforth, we assume that they intersect in a single point as we assume the point set  $S$  to lie in general position. The following lemma shows that, in fact, this interval realises the directed Hausdorff distance  $\vec{D}_H(S, \overline{ab})$  (Figure 6.3(b)):

**Lemma 6.1** *For any horizontal segment  $\overline{ab}$  and any point set  $S \subseteq \mathbb{R}^2$ , it must be that:*

$$\vec{D}_H(S, \overline{ab}) = \max \{h_{\overleftarrow{S}}(a), h_{\overrightarrow{S}}(b)\}.$$

**Proof** Assume without loss of generality that  $x_a \leq x_b$ , and let  $y = y_a = y_b$ . Partition  $S$  into three disjoint subsets  $L$ ,  $R$ , and  $M$ , where  $L \subseteq S$  contains all points in  $S$  strictly left of  $a$ ,  $R$  all points strictly right of  $b$ , and  $M$  all (remaining) points that lie in the vertical slab defined by  $x_a$  and  $x_b$ . We then have (Figure 6.4):

$$\begin{aligned} \vec{D}_H(S, \overline{ab}) &= \max_{s \in S} \min_{r \in \overline{ab}} \|s - r\| = \max \left\{ \max_{s \in L} \|s - a\|, \max_{s \in R} \|s - b\|, \max_{s \in M} |y_s - y| \right\} \\ &= \max \left\{ h_{\overleftarrow{L}}(a), h_{\overrightarrow{R}}(b), \max_{s \in M} |y_s - y| \right\}. \end{aligned}$$

We conclude the argument through showing that:

$$\max \left\{ h_{\overleftarrow{L}}(a), h_{\overrightarrow{R}}(b), \max_{s \in M} |y_s - y| \right\} = \max \{ h_{\overleftarrow{S}}(a), h_{\overrightarrow{S}}(b) \}.$$

For all points  $s$  not strictly left of  $a$  it must be that  $h_{\overleftarrow{s}}(a) = |y_s - y_a| = |y_s - y|$ . For all points  $s$  not strictly right of  $b$  it must be that  $h_{\overrightarrow{s}}(b) = |y_s - y_b| = |y_s - y|$ . Points in  $M$  are in between  $a$  and  $b$  and thus:  $\max_{s \in M} |y_s - y| = h_{\overleftarrow{M}}(a) = h_{\overrightarrow{M}}(b)$ .

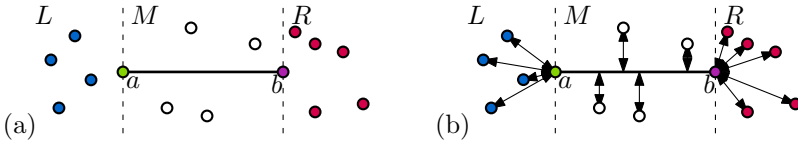
All points  $s \in R$  lie right of  $a$  and right of  $b$ . Therefore:

$$h_{\overleftarrow{s}}(a) = |y_s - y_a| = |y_s - y_b| \leq \|s - b\| = h_{\overrightarrow{s}}(b),$$

which immediately implies that for the set  $R$ :  $h_{\overleftarrow{R}}(a) \leq h_{\overrightarrow{R}}(b)$ . Via a symmetric argument it must be that for the set  $L$ :  $h_{\overleftarrow{L}}(a) \geq h_{\overrightarrow{L}}(b)$  and the lemma follows. ■

**Corollary 6.1** For any point  $p$  and point set  $S \subseteq \mathbb{R}^2$ ,  $\vec{D}_H(S, p) = \max \{ h_{\overleftarrow{S}}(p), h_{\overrightarrow{S}}(p) \}$ .

By Observation 6.1, the function  $h_{\overleftarrow{S}}$  corresponds to the furthest segment Voronoi diagram (FSVD) of  $\overleftarrow{S}$ . For  $d = 2$ , this diagram has size  $O(n)$  and can be computed in  $O(n \log n)$  time [170]. Thus, by preprocessing the FSVD for planar point location queries [180] we obtain a linear space data structure that allows us to evaluate  $h_{\overleftarrow{S}}(q)$  for any query point  $q \in \mathbb{R}^2$  in  $O(\log n)$  time. The edges in the FSVD are line segments, rays, or parabolic arcs [170].



**Figure 6.4** We split a point set  $S$  into sets  $L$ ,  $M$  and  $R$ . For points in  $L$ , their distance to  $ab$  is the distance to  $a$ , for points in  $R$  its the distance to  $b$  and for points in  $M$  is the vertical difference.

We split these edges into  $O(1)$   $x$ -monotone curved segments and store them in a partially persistent point location structure on the FDVD [180]. Analogously, we build a linear space data structure for querying  $h_{\vec{S}}$ , and obtain the following result through Lemma 6.1:

**Theorem 6.4** *Let  $S$  be a set of  $n$  points in  $\mathbb{R}^2$ . In  $O(n \log n)$  time we can build a data structure of linear size so that given a horizontal query segment  $\overline{ab}$ ,  $\vec{D}_H(S, \overline{ab})$  can be computed in  $O(\log n)$  time.*

### 6.3.2 The backward pairs term

In this section we show that the function  $y \mapsto \mathcal{D}_B(y)$ , representing the backward pair distance, has complexity  $O(n \log n)$ . We show how to compute the function with an appropriate search structure in  $O(n \log^2 n)$  time such that it can be evaluated for some query value  $y$  in  $O(\log n)$  time. This leads to an efficient data structure for querying  $P$  for the Fréchet distance to a horizontal query segment  $\overline{ab}$ , proving Theorem 6.1.

Recall that for a fixed  $y$ ,  $\mathcal{D}_B(y)$  is the maximum over all function values  $\delta_{pq}(y) = \min_x \max \{\|(x, y) - p\|, \|(x, y) - q\|\}$  for all backward pairs  $(p, q) \in \mathcal{B}(P)$ . We avoid explicitly computing  $\mathcal{B}(P)$  and we define a new function instead denoted by  $\delta'_{pq}(y)$  that applies to any ordered pair of points  $(p, q) \in P^\leq$ . Recall that  $P^\leq$  is the set of ordered pairs of vertices where for each pair  $(p, q) \in P^\leq$ ,  $p$  precedes or equals  $q$  along  $P$ . Since we assume that for the segment  $\overline{ab}$  the point  $a$  lies left of  $b$ , any pair  $(p, q) \in P^\leq$  is a backward pair if and only if the  $x$ -coordinate of  $p$  is greater than the  $x$ -coordinate of  $q$ . We show that for all backward pairs  $(p, q) \in \mathcal{B}(P)$ , we have  $\delta'_{pq}(y) = \delta_{pq}(y)$ . For any pair  $(p, q) \in P^\leq$  that is not a backward pair, we show that there exists a backward pair  $(p', q') \in \mathcal{B}(P)$  such that  $\delta'_{pq}(y) \leq \delta'_{p'q'}(y) = \delta_{p'q'}(y)$ . Consequently, we can compute the value  $\mathcal{D}_B(y)$  by computing the maximum value of  $\delta'_{pq}(y)$  over all pairs in  $P^\leq$ . We will show how to do this in an efficient manner.

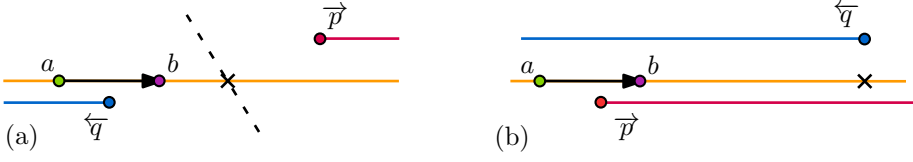
For each pair of points  $(p, q) \in P^\leq$ , we define the *pair distance* between a query  $\overline{ab}$  at height  $y$  and  $(p, q)$  as the Hausdorff distance from a horizontal line of height  $y$  to  $(\vec{p} \cup \vec{q})$ , or more formally (Figure 6.5):

$$\delta'_{pq}(y) = \min_x \max \{h_{\vec{q}}((x, y)), h_{\vec{p}}((x, y))\}.$$

Any pair  $(p, q) \in P^\leq$  is a backward pair if and only if  $x_q \leq x_p$ . Let  $(p, q)$  be a backward pair. For all points  $p$  and fixed  $y$ , the function  $x \mapsto \|(x, y) - p\|$  is convex, and minimized at  $x = x_p$ . The function  $\delta_{pq}(y)$  is defined as  $\delta_{pq}(y) = \min_x \max \{\|(x, y) - p\|, \|(x, y) - q\|\}$ . Since  $x_q \leq x_p$ , it follows from convexity that  $\max \{\|(x, y) - p\|, \|(x, y) - q\|\}$  is minimized for an  $x$  in  $[x_q, x_p]$ . Since for any point  $p$ ,  $\|(x, y) - p\| = \max \{h_{\vec{p}}((x, y)), h_{\overleftarrow{p}}((x, y))\}$ , we obtain the following:

**Lemma 6.2** *Let  $(p, q) \in P^\leq$  be a pair of points with  $x_p \geq x_q$ . Then for all  $y$ ,*

$$\delta_{pq}(y) = \delta'_{pq}(y).$$



**Figure 6.5** The distance  $\delta'_{pq}(y)$  is either realised by (a) the point of intersection between  $y$  and the bisector between  $\overleftarrow{q}$  and  $\overrightarrow{p}$  or, (b) the maximal vertical distance between a line at height  $y$  and either  $p$  or  $q$ .

**Proof** For all points  $p$ , the function  $x \mapsto \|(x, y) - p\|$  is convex, and minimized at  $x = x_p$ . Since  $x_q \leq x_p$ , it follows from the convexity that the function value  $\max \{\|(x, y) - p\|, \|(x, y) - q\|\}$  is minimized for an  $x$ -coordinate in  $[x_q, x_p]$  (and thus  $\delta_{pq}(y)$  is realized by an  $x$ -value in  $[x_q, x_p]$ ).

For all points  $p$ ,  $\|(x, y) - p\| = \max \{h_{\overrightarrow{p}}((x, y)), h_{\overleftarrow{p}}((x, y))\}$ , thus we observe that:

$$\begin{aligned}
 \delta_{pq}(y) &= \min_{x \in [x_q, x_p]} \max \{\|(x, y) - p\|, \|(x, y) - q\|\} \\
 &= \min_{x \in [x_q, x_p]} \max \{h_{\overleftarrow{q}}((x, y)), h_{\overleftarrow{q}}((x, y)), h_{\overrightarrow{p}}((x, y)), h_{\overrightarrow{p}}((x, y))\} \\
 &= \min_{x \in [x_q, x_p]} \max \{h_{\overleftarrow{q}}((x, y)), h_{\overrightarrow{p}}((x, y))\} \\
 &= \min_x \max \{h_{\overleftarrow{q}}((x, y)), h_{\overrightarrow{p}}((x, y))\} \\
 &= \delta'_{pq}(y).
 \end{aligned}$$

■

The consequence of the above lemma is that for each  $(p, q) \in \mathcal{B}(P)$ ,  $\delta_{pq}(y) = \delta'_{pq}(y)$ .

Next we make an observation about pairs of points that are not a backward pair:

**Lemma 6.3** Let  $(p, q) \in P^{\leq}$  be a pair of points with  $x_p < x_q$  then

$$\delta'_{pq}(y) = \max \{\delta_{qq}(y), \delta_{pp}(y)\}.$$

**Proof** By definition,  $\delta'_{pq}(y) = \min_x \max \{h_{\overleftarrow{q}}((x, y)), h_{\overrightarrow{p}}((x, y))\}$ . For any  $y$ , the function  $x \mapsto h_{\overleftarrow{q}}((x, y))$  is minimal and constant for all  $x \leq x_q$ . Similarly, the function  $x \mapsto h_{\overrightarrow{p}}((x, y))$  is minimal and constant for all  $x_p \leq x$ . Since  $(-\infty, x_q] \cap [x_p, \infty) = [x_p, x_q]$ , it follows that:

$$\begin{aligned}
 \delta'_{pq}(y) &= \min_x \max \{h_{\overleftarrow{q}}((x, y)), h_{\overrightarrow{p}}((x, y))\} \\
 &= \min_{x \in [x_p, x_q]} \max \{h_{\overleftarrow{q}}((x, y)), h_{\overrightarrow{p}}((x, y))\} \\
 &= \max \{|y_q - y|, |y_p - y|\} = \max \{\delta_{qq}(y), \delta_{pp}(y)\}.
 \end{aligned}$$

Where the last equality follows from the observation that for any point  $p$  and fixed  $y$ , the function  $x \mapsto \|(x, y) - p\|$  is convex, and minimized at  $x = x_p$ . ■

For all pairs of points  $(p, q) \in P^{\leq}$  either  $(p, q)$  is a backward pair or  $x_p < x_q$ , and thus we obtain the following lemma.

**Lemma 6.4** *For any polygonal curve  $P$  and any  $y$ ,*

$$\mathcal{D}_B(y) = \max \{ \delta_{pq}(y) \mid (p, q) \in \mathcal{B}(P) \} = \max \{ \delta'_{pq}(y) \mid (p, q) \in P^{\leq} \}.$$

**Relating  $\mathcal{D}_B(y)$  to furthest segment Voronoi diagrams.** What remains is to show how we can compute a suitable representation of  $\mathcal{D}_B(y)$  using our new functions  $\delta'_{pq}(y)$ . We devise a divide and conquer algorithm that computes  $\mathcal{D}_B(y)$  by computing it for subsets of vertices of  $P$ . To that end, we refine the definition of  $\mathcal{D}_B(y)$  to make it decomposable (we define  $\mathcal{D}_B(y)$  on pairs of subsets of  $P$ ). Let  $S, T$  be any two subsets of vertices of  $P$ , we define:

$$\mathcal{D}_B^{S \times T}(y) = \max \{ \delta'_{pq}(y) \mid (p, q) \in (S \times T) \cap P^{\leq} \}.$$

In the remainder of this section we proceed as follows: we fix a value of  $y$  and show that computing  $\mathcal{D}_B^{S \times T}(y)$  is equivalent to computing an intersection between two curves that consist of a linear number of pieces, each of constant complexity. We then argue that as  $y$  changes, the intersection point moves along a linear complexity curve that can be computed in  $O(n \log n)$  time. This allows us to query  $\mathcal{D}_B(y) = \mathcal{D}_B^{P \times P}(y)$  in  $O(\log n)$  time, for any query height  $y$ .

**From distance to intersections.** For a fixed value  $y'$ , we show that computing  $\mathcal{D}_B^{S \times T}(y')$  is equivalent to computing an intersection point between two curves:

**Lemma 6.5** *Let  $y' \in \mathbb{R}$  be a fixed height, let  $p$  be a point in  $P$ , and let  $T$  be a subset of the vertices of  $P[p, p_n]$ . The graphs of the functions  $x \mapsto h_{\vec{p}}((x, y'))$  and  $x \mapsto h_{\vec{T}}((x, y'))$  intersect at a single point  $(x^*, y')$ . Moreover,  $\mathcal{D}_B^{\{p\} \times T}(y') = h_{\vec{T}}((x^*, y')) = h_{\vec{p}}((x^*, y'))$ .*

**Proof** Recall that  $P[p, p_n]$  is the subcurve of  $P$  from  $p$  to  $p_n$ . In Observation 6.2 we noted that for all fixed  $y'$ , the function  $x \mapsto h_{\vec{T}}((x, y'))$  is monotonically increasing. Similarly, for any point  $p$ , the function  $x \mapsto h_{\vec{p}}((x, y'))$  is monotonically decreasing.

This implies that the value  $\min_x \max \{ h_{\vec{p}}((x, y')), h_{\vec{T}}((x, y')) \}$  is realized at  $x^*$  (the  $x$ -coordinate of their point of intersection). Recall that  $(x^*, y')$  is a unique point as we assume general position, i.e., no two points have the same  $y$ -coordinate.

Next, we apply the definition of  $\mathcal{D}_B^{\{p\} \times T}(y')$ :

$$\begin{aligned} \mathcal{D}_B^{\{p\} \times T}(y') &= \max_{q \in T} \{ \delta'_{pq}(y') \} = \max_{q \in T} \left\{ \min_x \max \{ h_{\vec{p}}((x, y')), h_{\vec{q}}((x, y')) \} \right\} = \\ &= \min_x \max \left\{ h_{\vec{p}}((x, y')), \max_{q \in T} \{ h_{\vec{q}}((x, y')) \} \right\} = \min_x \max \{ h_{\vec{p}}((x, y')), h_{\vec{T}}((x, y')) \} \Rightarrow \\ \mathcal{D}_B^{\{p\} \times T}(y') &= h_{\vec{T}}((x^*, y')) = h_{\vec{p}}((x^*, y')). \end{aligned}$$

Which concludes the proof. ■

Lemma 6.6 now follows easily from the previous lemma.

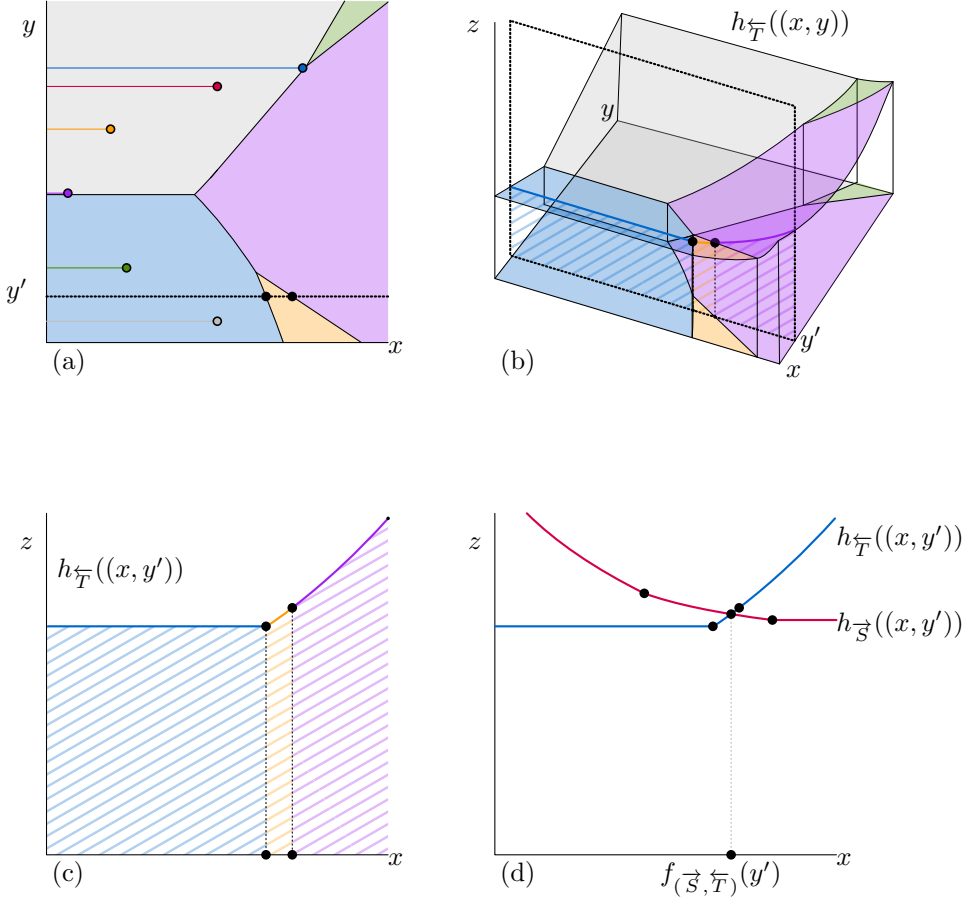
**Lemma 6.6** *Let  $S, T$  be subsets of vertices of  $P$  such that all vertices in  $S$  precede all vertices in  $T$ , and let  $y'$  be fixed. The graphs of the functions  $x \mapsto h_{\vec{S}}((x, y'))$  and  $x \mapsto h_{\vec{T}}((x, y'))$  intersect at a single point  $(x^*, y')$ . Moreover,  $\mathcal{D}_B^{S \times T}(y') = h_{\vec{S}}((x^*, y')) = h_{\vec{T}}((x^*, y'))$ .*

**Proof** If all points in  $S$  precede all points in  $T$ , then all elements in  $S \times T$  are in  $P^{\leq}$  and we note:  $\mathcal{D}_B^{S \times T}(y') = \max_{p \in S} \{ \mathcal{D}_B^{\{p\} \times T}(y') \}$ . The equality then follows from Lemma 6.5. ■

The above lemma immediately implies an approach to compute, for a fixed value  $y'$ , a linear-size representation of the function  $x \mapsto h_{\vec{T}}((x, y'))$  for a set of vertices  $T$ , which we illustrate in Figure 6.6.

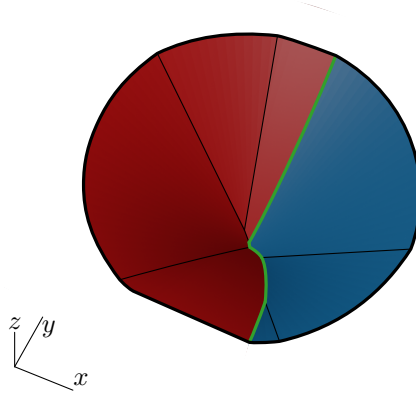
- We compute the furthest segment Voronoi diagram of  $\overleftarrow{T}$  in  $O(|T| \log |T|)$  time.
- We denote by  $\ell_{y'}$  a horizontal line of height  $y'$  and we compute the Voronoi cells of this *FSVD* intersected by  $\ell_{y'}$  in left-to-right order in  $O(|T| \log |T|)$  time.
- Suppose that a segment of  $\ell_{y'}$  intersects only the Voronoi cell belonging to a halfline  $\overleftarrow{q} \in \overleftarrow{T}$ , then on this domain the function  $h_{\vec{T}}((x, y')) = h_{\vec{q}}((x, y'))$ , and thus it has constant complexity. We store this constant-complexity function for every maximal segment of  $\ell_{y'}$  that intersects only one cell of the *FSVD* in  $O(|T|)$  total time through traversing the *FSVD*.
- Note that a representation of  $x \mapsto h_{\vec{S}}((x, y'))$  can be computed with a symmetric procedure in  $O(|S| \log |S|)$  time.

Let for all points in  $S$  precede all points in  $T$ . Given the above representation of their functions  $x \mapsto h_{\vec{T}}((x, y'))$  and  $x \mapsto h_{\vec{S}}((x, y'))$  we can find the  $x$ -coordinate  $x^*$  of intersection between these two functions in logarithmic time through a binary search on both respective functions. This approach however, can only compute the  $x$ -coordinate of the point of intersection between  $x \mapsto h_{\vec{T}}((x, y'))$  and  $x \mapsto h_{\vec{S}}((x, y'))$  for some fixed  $y$ -coordinate  $y'$ . Next we show how to extend our approach so that it works for varying  $y$ -coordinates.



**Figure 6.6** (a) A set  $\overleftarrow{T}$  of rays arising from a set  $T$  of points, with their *FSVD*. (b)  $h_{\overleftarrow{T}}((x, y))$  is the distance to the ray corresponding to the Voronoi cell at  $(x, y)$ . (c) For a fixed  $y'$ ,  $x \mapsto h_{\overleftarrow{T}}((x, y'))$  is monotonically increasing. (d) The value  $x^*$  for which  $h_{\overleftarrow{T}}((x, y')) = h_{\overrightarrow{S}}((x, y'))$  corresponds to  $D_B^{S \times T}(y')$ .

**Varying the  $y$ -coordinate.** Let  $f_{(\overrightarrow{S}, \overleftarrow{T})} : y \mapsto x^*$  be the function that for each  $y$  gives the intersection point  $x^*$  such that  $h_{\overrightarrow{S}}((x^*, y)) = h_{\overleftarrow{T}}((x^*, y))$ . Via the above argument, the intersection point  $(x^*, y')$  lies on a Voronoi edge of the *FSVD* of  $(\overleftarrow{T} \cup \overrightarrow{S})$ . More precisely, it lies on the bichromatic bisector of the *FSVD* of  $\overleftarrow{T}$  and the one of  $\overrightarrow{S}$  (see Figure 6.7). When we vary the  $y$ -coordinate, the  $x$ -coordinate of point of intersection between  $h_{\overrightarrow{S}}$  and  $h_{\overleftarrow{T}}$  traces this bisector. This implies that, given the *FSVD* of  $\overrightarrow{S}$  and the *FSVD* of  $\overleftarrow{T}$ , the graph of  $f_{(\overrightarrow{S}, \overleftarrow{T})}$  can be computed in  $O(|S| + |T|)$  time. Using these properties, we can prove the following.



**Figure 6.7** The  $h_{\overleftarrow{T}}$  (blue) and  $h_{\overrightarrow{S}}$  (red) functions shown plotted on a 3d cone (outside of the cone we encounter computational degeneracies). We highlight  $y \mapsto f_{(\overrightarrow{S}, \overleftarrow{T})}(y)$  (the bichromatic bisector) in green.

**Lemma 6.7** *Let  $S, T$  be subsets of vertices of  $P$  such that all vertices in  $S$  precede all vertices in  $T$ . The function  $\mathcal{D}_B^{S \times T}$  has complexity  $O(n)$  and can be computed in  $O((|S| + |T|) \log(|S| + |T|))$  time. Evaluating  $\mathcal{D}_B^{S \times T}(y)$ , for some query value  $y \in \mathbb{R}$ , takes  $O(\log n)$  time.*

**Proof** For any  $y$ , we consider the value  $x^*$  such that  $h_{\overrightarrow{S}}((x^*, y)) = h_{\overleftarrow{T}}((x^*, y))$  and the associated value  $\mathcal{D}_B^{S \times T}(y) = h_{\overrightarrow{S}}((x^*, y)) = h_{\overleftarrow{T}}((x^*, y))$ , or formally the function:

$$y \mapsto (f_{(\overrightarrow{S}, \overleftarrow{T})}(y), \mathcal{D}_B^{S \times T}(y)).$$

For each choice of  $y$  there exists a unique value  $f_{(\overrightarrow{S}, \overleftarrow{T})}(y)$  and we need to compute:

$$\mathcal{D}_B^{S \times T}(y) = h_{\overrightarrow{S}}((f_{(\overrightarrow{S}, \overleftarrow{T})}(y), y)) = h_{\overleftarrow{T}}((f_{(\overrightarrow{S}, \overleftarrow{T})}(y), y)).$$

Hence the image of  $y \mapsto (f_{(\overrightarrow{S}, \overleftarrow{T})}(y), \mathcal{D}_B^{S \times T}(y))$  is a well-defined curve in  $\mathbb{R}^3$  parametrized only by  $y$ . Now consider the image of  $y \mapsto (f_{(\overrightarrow{S}, \overleftarrow{T})}(y), \mathcal{D}_B^{S \times T}(y))$  projected onto the  $(x, y)$ -plane. Since this corresponds to the bichromatic bisector between the Farthest Segment Voronoi Diagrams of  $\overleftarrow{T}$  and  $\overrightarrow{S}$  this projected curve consists of linearly many, constant-complexity curves and can be computed in  $O((|S| + |T|) \log(|S| + |T|))$  time. Since the three-dimensional function is parametrized only by  $y$ , it follows that the projection of its image onto the  $(y, z)$ -plane also has  $O(|S| + |T|)$  complexity and can be computed in  $O(|S| + |T|)$  time. By storing the breakpoints of this function in a balanced binary search tree we can subsequently evaluate  $\mathcal{D}_B^{S \times T}(y)$ , for any  $y$ , in  $O(\log(|S| + |T|))$  time. ■

**Applying divide and conquer** Finally, we analyse the complexity of the function  $\mathcal{D}_B(y) = \mathcal{D}_B^{P \times P}(y)$ . Consider a partition of  $P$  into subcurves  $S$  and  $T$  with at most  $\lceil n/2 \rceil$  vertices each, and with  $S$  occurring before  $T$  along  $P$ . Our approach relies on the following fact:

**Observation 6.3** *Let  $P$  be partitioned into two subcurves  $S$  and  $T$  with all vertices in  $S$  occurring on  $P$  before the vertices of  $T$ . We have that*

$$\mathcal{D}_B(y) = \mathcal{D}_B^{P \times P}(y) = \max \{ \mathcal{D}_B^{S \times S}(y), \mathcal{D}_B^{S \times T}(y), \mathcal{D}_B^{T \times T}(y) \}.$$

Note that we can omit the term  $\mathcal{D}_B^{T \times S}(y)$  because  $(T \times S) \cap P^\leq = \emptyset$ . We obtain the following lemma.

**Theorem 6.5** *Let  $P$  be a polygonal curve with  $n$  vertices. Function  $\mathcal{D}_B$  has complexity  $O(n \log n)$  and can be computed in  $O(n \log^2 n)$  time. Evaluating  $\mathcal{D}_B(y)$ , for a given  $y \in \mathbb{R}$ , takes  $O(\log n)$  time.*

**Proof** By Observation 6.3,  $\mathcal{D}_B(y) = \mathcal{D}_B^{P \times P}(y) = \max \{ \mathcal{D}_B^{S \times S}(y), \mathcal{D}_B^{S \times T}(y), \mathcal{D}_B^{T \times T}(y) \}$ . By Lemma 6.7, the complexity of  $\mathcal{D}_B^{S \times T}$  is  $O(|S| + |T|)$ . Hence, there are  $O(|S| + |T|)$  backward pairs from  $S \times T$  which may contribute to  $\mathcal{D}_B^{P \times P}$  (that is, there are at most  $O(|S| + |T|)$  backward pairs  $(p, q) \in S \times T$  such that there exists a value  $y'$  such that  $\mathcal{D}_B^{P \times P}(y')$  is realised by the backward pair distance to  $(p, q)$ ).

Let  $C(n)$  denote the number of backward pairs contributing to  $\mathcal{D}_B^{P \times P}$ . It follows that  $C(n) = 2C(\lceil n/2 \rceil) + O(n)$ , which solves to  $O(n \log n)$ . Since the complexity of  $\mathcal{D}_B = \mathcal{D}_B^{P \times P}$  is linear in the number of contributing backward pairs [63], the function  $\mathcal{D}_B$  has complexity  $O(n \log n)$ .

To compute  $\mathcal{D}_B$  we apply the same divide and conquer strategy. We recursively partition  $P$  into roughly equal size subcurves  $S$  and  $T$ . At each step, we compute the (graph of the) function  $\mathcal{D}_B^{S \times T}$ , and merge it with the recursively computed functions  $\mathcal{D}_B^{S \times S}$  and  $\mathcal{D}_B^{T \times T}$ . By Lemma 6.7, computing  $\mathcal{D}_B^{S \times T}$  takes  $O(n \log n)$  time. Computing the upper envelope of  $\mathcal{D}_B^{S \times T}$ ,  $\mathcal{D}_B^{S \times S}$ , and  $\mathcal{D}_B^{T \times T}$ , takes time linear in the complexity of the functions involved. The function  $\mathcal{D}_B^{S \times T}$  has complexity  $O(n)$ . However,  $\mathcal{D}_B^{S \times S}$ ,  $\mathcal{D}_B^{T \times T}$ , and the output  $\mathcal{D}_B^{P \times P}$ , have complexity  $O(n \log n)$ . Hence, we spend  $O(n \log n)$  time to compute  $\mathcal{D}_B^{P \times P}$ . The total running time obeys the recurrence  $R(n) = 2R(n/2) + O(n \log n)$ , that resolves to  $O(n \log^2 n)$  time.

We can easily store (the breakpoints of)  $\mathcal{D}_B^{P \times P}$  in a balanced binary search tree so that we can evaluate  $\mathcal{D}_B(y)$  for some query value  $y$  in  $O(\log(n \log n)) = O(\log n)$  time. ■

Equation 6.1 together with Theorem 6.4 and Theorem 6.5 thus imply that we can store  $P$  in an  $O(n \log n)$  size data structure so that we can compute  $\mathbf{D}_{\mathbb{F}}(P, \overline{ab})$  for some horizontal query segment  $\overline{ab}$  in  $O(\log n)$  time. That is, we established Theorem 6.1:

**Theorem 6.1** *Let  $P$  be a polygonal curve in  $\mathbb{R}^2$  with  $n$  vertices. There is an  $O(n \log n)$  size data structure that can be built in  $O(n \log^2 n)$  time such that given a horizontal query segment  $\overline{ab}$  it can report  $\mathbf{D}_{\mathbb{F}}(P, \overline{ab})$  in  $O(\log n)$  time.*

## 6.4 Horizontal queries: querying for subcurves

In this section we extend our data structure to support Fréchet distance queries to subcurves of  $P$ , establishing Theorem 6.2. A query now consists of two points  $s$  and  $t$  on  $P$  and the horizontal query segment  $\overline{ab}$ , and we wish to efficiently report the Fréchet distance  $\mathbf{D}_{\mathbb{F}}(P[s, t], \overline{ab})$  between the subcurve  $P[s, t]$ , from  $s$  to  $t$ , and  $\overline{ab}$ . Crucially, we do not demand that  $s$  and  $t$  are vertices of  $P$ . We assume that given  $s$  and  $t$  we can determine the edges of  $P$  containing  $s$  and  $t$  respectively, in constant time. Note that this is the case, for instance, when  $s$  is given as a pointer to its containing edge together with a location. If  $s$  and  $t$  are given only as points in the plane, and  $P$  is not self-intersecting, we can find these edges in  $O(\log n)$  time using a linear-size data structure for vertical ray-shooting [180] on  $P$ . If  $P$  does contain self-intersections this requires more space and preprocessing time [3].

We show that we can support such queries in  $O(\log^3 n)$  time using  $O(n \log^2 n)$  space. The two main ideas for this section are as follows:

- we can explicitly store all intermediate data structures constructed in the above divide and conquer algorithm and,
- we can actually achieve the result of Lemma 6.7 (evaluating  $\mathcal{D}_B^{S \times T}(y')$  for some query value  $y'$ , and some sets  $S$  and  $T$ , in  $O(\log n)$  time) by *separately* storing a data structure on  $S$  and a data structure on  $T$ .

By Equation 6.1,  $\mathbf{D}_{\mathbb{F}}(P[s, t], \overline{ab})$  can again be decomposed into four terms, the first two of which can be trivially computed in constant time. Our final data structure is a two-level data structure that has total size  $O(n \log^2 n)$ . The first level allows us to find  $O(\log n)$  nodes whose associated subcurves contain all vertices of the query subcurve  $P[s, t]$ . For each such pair of nodes  $\mu, \nu$  we use the (extended) Lemma 6.7 data structures associated with these nodes to compute the contribution  $\mathcal{D}_B^{P_\mu \times P_\nu}(y)$  of backward pairs with in one vertex in subcurve  $P_\nu$  and one vertex in  $P_\mu$ . We subsequently handle the points  $s$  and  $t$  separately and thus spend  $O(\log^3 n)$  time to compute the backward pair distance. This dominates the  $O(\log^2 n)$  time required to query the Hausdorff distance term for each subcurve.

Just as in the previous section, we build two separate data structures for the remaining two terms: the Hausdorff distance and the backward pair distance terms.

### 6.4.1 Hausdorff distance for subcurves

We build a data structure on  $P$  such that given points  $s, t$  on  $P$  and  $\overline{ab}$  we can report  $\overrightarrow{\mathbf{D}}_H(P[s, t], \overline{ab})$  efficiently. In particular, we use a two-level data structure of size  $O(n \log n)$  that supports queries in  $O(\log^2 n)$  time, after  $O(n \log n)$  preprocessing time, based on two observations:

1. The Hausdorff distance is decomposable in its first argument. That is, for all points  $m \in P[s, t]$ :

$$\vec{D}_H(P[s, t], \overline{ab}) = \max \left\{ \vec{D}_H(P[s, m], \overline{ab}), \vec{D}_H(P[m, t], \overline{ab}) \right\}.$$

2. The Hausdorff distance  $\vec{D}_H(P[s, t], \overline{ab})$  is realized by  $s$ ,  $t$ , or a vertex  $p$  of  $P[s, t]$ . That is, there is a vertex  $p \in P$  such that:

$$\vec{D}_H(P[s, t], \overline{ab}) = \max \{ \vec{D}_H(p, \overline{ab}), \vec{D}_H(s, \overline{ab}), \vec{D}_H(t, \overline{ab}) \}.$$

The data structure is a balanced binary search tree (essentially, a 1D-range tree) in which the leaves store the vertices of  $P$ , in the order along  $P$ . Each internal node  $\nu$  represents a *canonical subcurve*  $P_\nu$  and stores the vertices of  $P_\nu$  in the data structure of Theorem 6.4. Since these associated data structures use linear space, the total space used is  $O(n \log n)$ . Building the associated data structures from scratch would take  $O(n \log^2 n)$  time. However, the following lemmas immediately imply that this time can be reduced to  $O(n \log n)$ :

**Lemma 6.8** *The FSDs of  $\vec{S}$  and  $\overleftarrow{S}$  consist of  $y$ -monotone cells.*

**Proof** We assume for the sake of contradiction that there exists some horizontal line  $\ell_y$  at height  $y$  that intersects a cell in  $\vec{S}$  twice (the case for  $\overleftarrow{S}$  is symmetric). Specifically we assume that there is a ray  $\vec{p}_i \in \vec{S}$  such that the furthest Voronoi region of  $\vec{p}_i$ , intersected by  $\ell_y$ , has at least two maximal disjoint intervals  $A$  and  $B$  where  $A$  is left of  $B$  (these intervals contain the boundary of the associated Voronoi region). Consider the rightmost point  $x$  in  $A$ . Since  $x$  coincides with the right boundary of  $A$ , there is a ray  $\vec{p}_j \in \vec{S}$  such that  $d((x, y), \vec{p}_i) = d((x, y), \vec{p}_j)$ , and for an arbitrary small  $\varepsilon > 0$ ,  $d((x + \varepsilon, y), \vec{p}_i) < d((x + \varepsilon, y), \vec{p}_j)$ . We make a distinction based on whether the distance from  $(x, y)$  to  $\vec{p}_i$  is realized by the distance to the point  $p_i$  or by the vertical distance to the line supporting  $\vec{p}_i$  (similarly for the distance from  $(x, y)$  to  $\vec{p}_j$ ). Refer to Figure 6.8. The first two cases show that the interval  $B$  is empty and the last two show that the interval  $A$  does not end at  $x$ .

**Case 1:**  $d((x, y), \vec{p}_i) = d((x, y), p_i)$  and  $d((x, y), \vec{p}_j) = d((x, y), p_j)$ . In this case,  $x$  lies on the bisector between  $p_i$  and  $p_j$  and the interval  $A$  lies left of this bisector. Hence  $p_i$  must lie right of  $p_j$ . It follows that for all  $x' > x$ ,  $d((x', y), \vec{p}_i) < d((x', y), \vec{p}_j)$ . Indeed if for such  $x'$ ,  $d((x', y), \vec{p}_j) = d((x', y), p_j)$  then it must be that  $d((x', y), \vec{p}_i) = d((x', y), p_i)$ . Since  $x'$  lies right of the bisector between  $p_i$  and  $p_j$  it follows that  $d((x', y), \vec{p}_i) = d((x', y), p_i) < d((x', y), \vec{p}_j) = d((x', y), p_j)$ . For all  $x' > x$  where  $d((x', y), \vec{p}_j) \neq d((x', y), p_j)$ , the value  $d((x', y), \vec{p}_j)$  remains constant whilst the value  $d((x', y), \vec{p}_i)$  decreases or stays constant. This contradicts the assumption that the farthest Voronoi cell of  $\vec{p}_i$  intersects  $\ell_y$  right of  $x$ .

**Case 2:**  $d((x, y), \vec{p_i}) = d((x, y), p_i)$  and  $d((x, y), \vec{p_j}) \neq d((x, y), p_j)$ . In this case, for all  $x' > x$ ,  $d((x', y), \vec{p_j}) = d((x, y), \vec{p_j})$  and  $d((x', y), \vec{p_i}) < d((x, y), \vec{p_i})$ . This contradicts the assumption that the farthest Voronoi cell of  $\vec{p_i}$  intersects  $\ell_y$  right of  $x$ .

**Case 3:**  $d((x, y), \vec{p_i}) \neq d((x, y), p_i)$  and  $d((x, y), \vec{p_j}) = d((x, y), p_j)$ . In this case, for all  $x' > x$ ,  $d((x', y), \vec{p_j}) < d((x, y), \vec{p_j})$  and  $d((x', y), \vec{p_i}) = d((x, y), \vec{p_i})$  which contradicts the assumption that for an arbitrary small  $\varepsilon > 0$ ,  $d((x + \varepsilon, y), \vec{p_i}) < d((x + \varepsilon, y), \vec{p_j})$ .

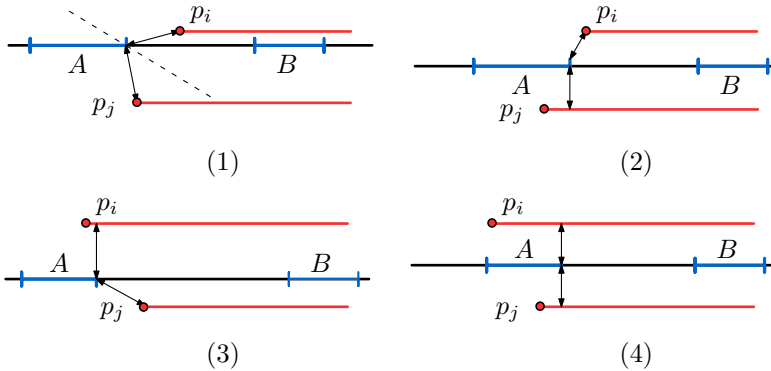
**Case 4:**  $d((x, y), \vec{p_i}) \neq d((x, y), p_i)$  and  $d((x, y), \vec{p_j}) \neq d((x, y), p_j)$ . In this case, for all  $x' > x$ ,  $d((x', y), \vec{p_j}) = d((x, y), \vec{p_j})$  and  $d((x', y), \vec{p_i}) = d((x, y), \vec{p_i})$  which contradicts the assumption that for an arbitrary small  $\varepsilon > 0$ ,  $d((x + \varepsilon, y), \vec{p_i}) < d((x + \varepsilon, y), \vec{p_j})$ . ■

**Lemma 6.9** *Two instances of the data structure of Theorem 6.4, consisting of a furthest segment Voronoi diagram preprocessed for point location, can be merged in linear time.*

**Proof** Two furthest segment Voronoi diagrams can be merged in linear time [170]. Furthermore, if the regions are  $y$ -monotone then the associated point location data structure can also be constructed in linear time [78]. ■

The above lemma almost immediately implies the following theorem:

**Theorem 6.6** *Let  $P$  be a polygonal curve in  $\mathbb{R}^2$  with  $n$  vertices. In  $O(n \log n)$  time we can construct a data structure of size  $O(n \log n)$  so that given a horizontal query segment  $\overline{ab}$ , and two points  $s, t$  on  $P$ ,  $\vec{D}_H(P[s, t], \overline{ab})$  can be computed in  $O(\log^2 n)$  time.*



**Figure 6.8** The four cases considered in the proof of Lemma 6.9 in order. The black line is some fixed horizontal line  $\ell_y$  at height  $y$  that supposedly intersects the Voronoi region of  $\vec{p_i}$  twice.

**Proof** The data structure is a range tree on the vertices of  $P$  where in every node we store the data structure of Theorem 6.4. By Lemma 6.9 this  $O(n \log n)$  size data structure can be constructed in  $O(n \log n)$  total time. Let  $s, t$  be two points on  $P$ , and let  $s'$  and  $t'$  be the first vertex succeeding  $s$  and preceding  $t$ , respectively. The terms  $\vec{D}_H(P[s, s'], \overline{ab})$  and  $\vec{D}_H(P[t', t], \overline{ab})$  can be computed in constant time as this is the Hausdorff distance between two constant-complexity objects.

There are  $O(\log n)$  internal nodes in our range tree whose canonical subcurves together form  $P[s', t']$  (see e.g. [61, Chapter 5]). For each such node  $\nu$ , corresponding to a subcurve  $P_\nu$ , we query its associated data structures in time logarithmic in the number of vertices in  $P_\nu$  to compute  $\vec{D}_H(P_\nu, \overline{ab})$ . This takes  $O(\log^2 n)$  total time. We report the maximum distance found and conclude the theorem. ■

We wish to briefly note that since we are given pointers to  $s$  and  $t$  we can make the query time sensitive to the complexity  $|P[s, t]|$  of the query subcurve. We do thus by augmenting our balanced decomposition of  $P$ . We store for every internal node  $v$  that stores the canonical subcurve  $P[p_i, p_j]$  the interval  $[i, j]$ . Note that given a pointer to  $s'$  and  $t'$ , this allows us to determine if  $s'$  or  $t'$  is stored in the internal node in  $O(1)$  time by comparing the index of  $s'$  and  $t'$  to this interval. Additionally, we add pointers between neighboring internal leaves of equal height (these are sometimes called *level links* [159]). It is well-known that given a pointer (*finger*) to  $s'$  and  $t'$  that we can identify the internal nodes whose canonical subcurves make up  $P[s', t']$  in  $O(\log |P[s, t]|)$  time through traversing this binary tree. Refer to Chapter 5.3.3 from the textbook by Mehlhorn [159]. The remainder of our algorithm is quadratic in the number of nodes whose canonical subcurves make up  $P[s', t']$  and we conclude:

**Theorem 6.7** *Let  $P$  be a polygonal curve in  $\mathbb{R}^2$  with  $n$  vertices. In  $O(n \log n)$  time we can construct a data structure of size  $O(n \log n)$  so that given a horizontal query segment  $\overline{ab}$ , and two points  $s, t$  on  $P$ ,  $\vec{D}_H(P[s, t], \overline{ab})$  can be computed in  $O(\log^2 |P[s, t]|)$  time.*

## 6.4.2 Backward pair distance for subcurves

In this section we describe how to store  $P$  so that given points  $s$  and  $t$  on  $P$  and the horizontal query segment  $\overline{ab}$  we can compute the backward pair distance  $\mathcal{D}_B^{P[s, t] \times P[s, t]}(y_a)$  efficiently. The main idea to support subcurve queries is to store all intermediate results of the divide and conquer algorithm from Section 6.3.2. Hence, our main data structure is a 1D-range tree whose leaves store the vertices of  $P$ , ordered along  $P$ . Each internal node  $\nu$  corresponds to some subcurve  $P_\nu$  of  $P$ , and will store the function  $\mathcal{D}_B^{P_\nu \times P_\nu}(y)$  (Theorem 6.5) as well as the functions  $h_{\overleftarrow{P}_\nu}(x, y)$  and  $h_{\overrightarrow{P}_\nu}(x, y)$  represented by two new data structures that we will denote by  $\Delta_{\overleftarrow{P}_\nu}(y)$  and  $\Delta_{\overrightarrow{P}_\nu}(y)$ .

We first sketch our query approach: in the following subsections we show how to construct the specific data structure and queries. We are given  $y'$ , and  $s, t \in P$ . We then identify the vertex  $s'$  succeeding  $s$  and the vertex  $t'$  preceding  $t$ , and define  $P_0 = P[s, s']$  and  $P_{k+1} = P[t', t]$ . There are  $k = O(\log n)$  internal nodes whose canonical subcurves  $P_1, P_2, \dots$  together form  $P[s', t']$ . We prove that  $\mathcal{D}_B^{P[s, t] \times P[s', t']}(y')$  is the maximum over  $O(k^2) = O(\log^2 n)$  terms in two categories:

$$\mathcal{D}_B^{P_i \times P_i}(y'), \text{ for all } i, \text{ and } \mathcal{D}_B^{P_i \times P_j}(y') \text{ for } i < j.$$

The  $O(\log n)$  values of the first category can be computed in  $O(\log^2 n)$  total time, using the data structures of Theorem 6.5 stored in each node. The second category contains  $O(\log^2 n)$  values, and we show how to compute each value in  $O(\log n)$  time, using the new data structures, for a total query time of  $O(\log^3 n)$ . This query time then dominates the time it takes to compute the maximum of all these terms.

**Using furthest segment Voronoi diagrams to compute  $\mathcal{D}_B^{S \times T}(y')$ .** Let  $S, T$  be two contiguous subcurves of  $P$ , with  $S$  occurring strictly before  $T$  on  $P$ . We first study how to compute  $\mathcal{D}_B^{S \times T}(y')$  efficiently for any given  $y'$ . For ease of exposition, we first show how to construct a linear-size data structure on  $T$  such that given a query point  $p$  that precedes  $T$  along  $P$  and a value  $y'$ , we can compute  $\mathcal{D}_B^{\{p\} \times T}(y')$  in  $O(\log |T|)$  time. By Lemma 6.5 this amounts to computing the point of intersection between the functions  $x \mapsto h_{\vec{p}}((x, y'))$  and  $x \mapsto h_{\overleftarrow{T}}((x, y'))$ .

Our global approach is illustrated in Figure 6.6. Suppose for the ease of exposition, that  $y'$  is fixed. Given the query point  $p$ , we can compute the convex function  $x \mapsto h_{\vec{p}}((x, y'))$  in constant time. For the set  $\overleftarrow{T}$ , we note that Observation 6.1 shows that the (graph of the) function  $(x, y) \mapsto h_{\overleftarrow{T}}((x, y))$  is an upper envelope whose maximization diagram is the furthest segment Voronoi diagram of the set of halflines  $\overleftarrow{T}$ . For any fixed  $y'$  and  $\overleftarrow{T}$ , we can construct a data structure  $\Delta_{\overleftarrow{T}}(y')$  that stores the Voronoi cell edges that are intersected by a horizontal line of height  $y'$  in their left-to-right order. Specifically, we store a red-black tree where each node stores a consecutive pair of edges (in this way, a node in the tree corresponds to a unique Voronoi cell in the diagram). Since a furthest segment Voronoi diagram has a linear number of edges,  $\Delta_{\overleftarrow{T}}(y')$  has  $O(|T|)$  size.

**Lemma 6.10** *Let  $p \in P$  be a query point, let  $y'$  be some given value, and let  $T$  be a sequence of points of  $P$  that succeed  $p$ . Given  $\Delta_{\overleftarrow{T}}(y')$  we can compute the point  $(x^*, y')$  where  $h_{\vec{p}}((x^*, y')) = h_{\overleftarrow{T}}((x^*, y'))$  in  $O(\log |T|)$  time.*

**Proof** Consider the two edges stored at the root of  $\Delta_{\overleftarrow{T}}(y')$ . These two edges partially bound a Voronoi cell corresponding to a halfline  $\overleftarrow{r} \in \overleftarrow{T}$ . Given  $p$  we can, in constant time, compute the point of intersection  $(x', y')$  between  $x \mapsto h_{\vec{p}}((x, y'))$  and  $x \mapsto h_{\overleftarrow{r}}((x, y'))$ .

Using the two edges stored at the root we can compute the Voronoi cell of  $\overleftarrow{T}$  that is bounded by the two edges constant time. This cell intersects  $\ell_y$  in some  $x$ -interval. If the value  $x'$  lies in this interval, then we have found the unique point of intersection between  $h_{\overrightarrow{P}}$  and  $h_{\overleftarrow{T}}$  (in other words,  $(x', y') = (x^*, y')$ ). If  $x'$  lies left of this  $x$ -interval, we can disregard all nodes in the right subtree of the root. This is because for all  $x \geq x'$ ,  $h_{\overleftarrow{T}}(x, y') \geq h_{\overleftarrow{T}}(x, y') > h_{\overrightarrow{P}}(x, y')$ . A symmetrical property holds when  $x'$  lies right of this  $x$ -interval. Hence at we go down the tree we can discard at least half of the remaining candidate halflines in  $\overleftarrow{T}$ . It follows that we can compute the halflines in  $\overleftarrow{T}$  that forms the intersection with  $x \mapsto h_{\overrightarrow{P}}(x, y')$  in logarithmic time. ■

Now, we extend our approach from products of single points and sets, to products between sets. Recall that for any set of horizontal halflines  $\overrightarrow{S}$ , the function  $x \mapsto h_{\overrightarrow{S}}(x, y')$  is piecewise monotone, where each piece is a constant-complexity segment coinciding with  $x \mapsto h_{\overrightarrow{s}}(x, y')$  for some  $\overrightarrow{s} \in \overrightarrow{S}$ .

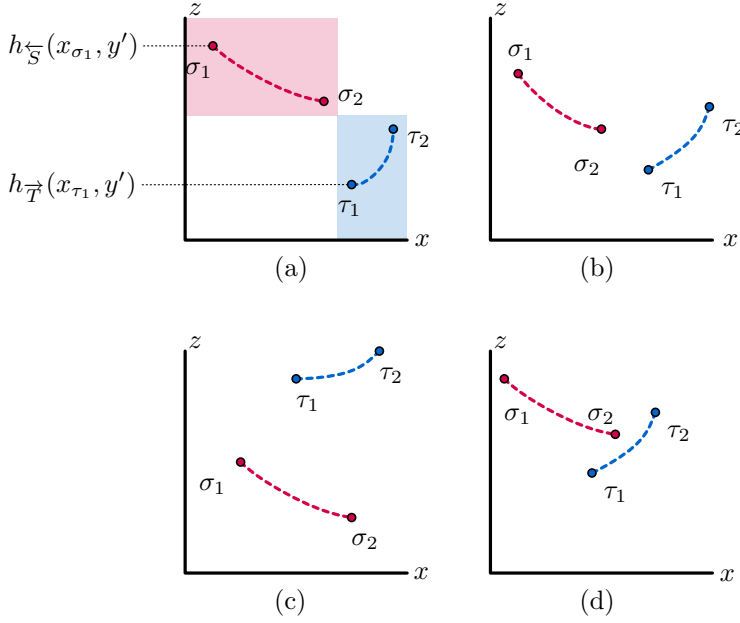
**Lemma 6.11** *Let  $S, T \subset P$  such that all points in  $S$  precede all points in  $T$  and let  $y'$  be some given value. Let  $\sigma_1, \sigma_2$  be two points on  $x \mapsto h_{\overrightarrow{S}}(x, y')$  that bound a curve segment with  $\sigma_1$  left of  $\sigma_2$ . Moreover, let  $\tau_1, \tau_2$  be defined analogously as points on  $h_{\overleftarrow{T}}$ . If the segments bounded by  $(\sigma_1, \sigma_2)$  and  $(\tau_1, \tau_2)$  do not intersect then at least one of four following cases applies and we can identify each in  $O(1)$  time. The point of intersection between  $h_{\overrightarrow{S}}(x, y')$  and  $h_{\overleftarrow{T}}(x, y')$  lies: (1) left of  $\sigma_1$ , (2) right of  $\sigma_2$ , (3) left of  $\tau_1$ , or (4) right of  $\tau_2$ .*

**Proof** The proof is a case distinction, illustrated by Figure 6.9, that relies on the property that  $x \mapsto h_{\overrightarrow{S}}(x, y')$  and  $x \mapsto h_{\overleftarrow{T}}(x, y')$  are monotonically decreasing and increasing functions, respectively.

**Case (a): there is no vertical or horizontal line that intersects both segments.**

In this case, there is a separation of the  $(x, z)$ -plane into four quadrants, such that one segment lies in a top quadrant and the other segment in the opposite bottom quadrant. Let the segment of  $h_{\overrightarrow{S}}$  lie in the top left quadrant, then all points on  $h_{\overrightarrow{S}}$  left of this segment, are higher than  $\sigma_1$ . All points left of  $h_{\overleftarrow{T}}$  must be lower than  $\tau_1$ . Hence all segments of  $h_{\overrightarrow{S}}$  left  $\sigma_1$  cannot form the point of intersection and can be discarded. Given  $(\sigma_1, \sigma_2, \tau_1, \tau_2)$  we can identify this case in constant time. The three remaining (sub-)cases are symmetrical.

**Case (b): At least one horizontal line intersects both segments.** If the segments between  $(\sigma_1, \sigma_2)$  and  $(\tau_1, \tau_2)$  do not intersect, all horizontal lines intersect these two segments in the same order. Via the same argument as above we note that if the segment of  $h_{\overrightarrow{S}}$  is intersected first, then all segments on  $h_{\overrightarrow{S}}$  left of  $\sigma_1$  can be discarded. Similarly then all segments of  $h_{\overleftarrow{T}}$  right of  $\tau_2$  can be discarded. The argument for when the order is reversed is symmetrical and we can compute the intersections with a horizontal line in  $O(1)$  time.



**Figure 6.9** The 3 cases in the proof of Lemma 6.11. In (d) both case (b) and (c) apply.

**Case (c): At least one vertical line intersects both segments.** In this case, each vertical line that intersects both segments must intersect them in the same order. We note that if the segment between  $(\sigma_1, \sigma_2)$  is intersected first, then all segments on  $h_{\vec{S}}$  right of  $\sigma_2$  can be discarded, along with all segments right of  $\tau_2$ . The argument for when the order is reversed is symmetrical and we can compute the intersections with a vertical line in  $O(1)$  time. ■

**Lemma 6.12** Let  $S, T \subset P$  where points in  $S$  precede points in  $T$ . Let  $y'$  be some given value. Given  $\Delta_{\vec{S}}(y)$  and  $\Delta_{\vec{T}}(y')$ , we can compute  $\mathcal{D}_B^{S \times T}(y)$  in  $O(\log |S| + \log |T|)$  time.

**Proof** The algorithmic procedure that computes  $\mathcal{D}_B^{S \times T}(y)$  traverses the trees  $\Delta_{\vec{S}}(y)$  and  $\Delta_{\vec{T}}(y')$  from root to leaf until the point of intersection between  $h_{\vec{S}}(x, y')$  and  $h_{\vec{T}}(x, y')$  is found. Consider the Voronoi edges stored at the root of  $\Delta_{\vec{S}}(y')$ . We say that these edges are intersected by a horizontal line of height  $y'$  in the  $x$ -coordinates  $x_1$  and  $x_2$  respectively. The edges of the Voronoi cell point to the cell corresponding to the halfline  $\vec{s} \in \vec{S}$ . Per definition of the Voronoi diagram, for all  $x'$  with  $x_1 < x' < x_2$ ,  $h_{\vec{S}}(x', y') = h_{\vec{s}}(x', y')$ . Given the pointer to  $\vec{s} \in \vec{S}$  we compute the points  $\sigma_1 = h_{\vec{S}}(x_1, y') = h_{\vec{s}}(x_1, y')$  and  $\sigma_2 = h_{\vec{S}}(x_2, y') = h_{\vec{s}}(x_2, y')$  in constant time and the segment of  $h_{\vec{S}}(x, y')$  connecting them. Similarly, we can compute a point  $\tau_1$  and point  $\tau_2$  for the edges stored at the root of  $\Delta_{\vec{T}}(y')$ , and the segment of  $h_{\vec{T}}(x, y')$  connecting  $(\tau_1, \tau_2)$ .

Then, we can apply Lemma 6.11 to the pair  $(\sigma_1, \sigma_2)$ ,  $(\tau_1, \tau_2)$ , to discard at least all segments left/right of  $(\sigma_1, \sigma_2)$  or all segments left/right of  $(\tau_1, \tau_2)$ . Repeating this recursively on the remaining subtrees of  $\Delta_{\vec{S}}(y')$  and  $\Delta_{\overleftarrow{T}}(y')$ , we can find the two segments defining the point of intersection in  $O(\log |S| + \log |T|)$  time. ■

All that remains is to show that we can apply Lemma 6.12 not only to a given  $y'$ , but to any  $y$ . We note that for any  $y$ ,  $\Delta_{\overleftarrow{T}}(y)$  and  $\Delta_{\vec{S}}(y)$  are balanced binary trees. We sweep the plane with a horizontal line at height  $y$  while maintaining  $\Delta_{\overleftarrow{T}}(y)$  as a partially persistent red black tree [180]. We do the same for  $\Delta_{\vec{S}}(y)$  and conclude:

**Lemma 6.13** *Let  $S$  and  $T$  be two sets of vertices of  $P$ . We can store  $S$  in a data structure of size  $O(|S|)$ , and  $T$  in a data structure of size  $O(|T|)$  such that, if all vertices in  $S$  precede all vertices in  $T$ , we can compute  $\mathcal{D}_B^{S \times T}(y')$  for any fixed  $y' \in \mathbb{R}$  in  $O(\log(|S| + |T|))$  time. Building these data structures takes  $O(|S| \log |S|)$  and  $O(|T| \log |T|)$  time, respectively.*

**Proof** Consider a horizontal sweepline at height  $y'$ , and consider the continuously changing data structure  $\Delta_{\overleftarrow{T}}(y')$ . For any set  $\overleftarrow{T}$ , the FSVD has linear complexity [170]. Hence, for any  $y'$ ,  $\Delta_{\overleftarrow{T}}(y')$  contains at most a  $O(|T|)$  edges. Moreover, there are only  $O(|T|)$   $y$ -coordinates at which the combinatorial structure of  $\Delta_{\overleftarrow{T}}(y)$  changes. At each such an event we make a constant number of updates to  $\Delta_{\overleftarrow{T}}$ . Since  $\Delta_{\overleftarrow{T}}$  is a (partially persistent) red black tree, these changes take  $O(\log |T|)$  time, and  $O(1)$  space [180]. We use the same preprocessing for  $\vec{S}$ . Finally, we observe that we can obtain  $\Delta_{\overleftarrow{T}}(y')$  and  $\Delta_{\vec{S}}(y')$  for a given  $y' \in \mathbb{R}$  in  $O(\log |T| + \log |S|)$  time. ■

**Divide and conquer for subcurves.** What remains is to show that we can use the data structure from Lemma 6.13 to compute the backward pair distance  $\mathcal{D}_B^{P[s,t] \times P[s,t]}(y)$  efficiently. We denote by  $\Delta_{\overleftarrow{T}}(y)$  the partially persistent red-black tree that is obtained by sweeping a horizontal line through the furthest-segment Voronoi diagram of  $\overleftarrow{T}$ . We now show how for any two vertices  $s, t \in P$  and height  $y'$ , we can compute  $\mathcal{D}_B^{P[s,t] \times P[s,t]}(y')$  efficiently through repeated application of Lemma 6.13 and the following observation. For completeness, we present a proof of this observation.

**Observation 6.4** *Let  $S$  and  $T$  be two sets of vertices of  $P$  such that all points in  $S$  precede all points in  $T$  and  $S = S' \cup S''$  and  $T = T' \cup T''$ . Then for all  $y \in \mathbb{R}$ :*

$$\mathcal{D}_B^{S \times T}(y) = \max \left\{ \mathcal{D}_B^{S' \times T}(y), \mathcal{D}_B^{S'' \times T}(y) \right\} \text{ and } \mathcal{D}_B^{S \times T}(y) = \max \left\{ \mathcal{D}_B^{S \times T'}(y), \mathcal{D}_B^{S \times T''}(y) \right\}.$$

**Proof** Recall that  $\mathcal{D}_B^{S \times T}(y) = \max \{ \delta'_{pq}(y) \mid (p, q) \in (S \times T) \cap P^{\leq} \}$ . Using that  $S = S' \cup S''$  we then get  $S \times T = (S' \times T) \cup (S'' \times T)$ , and thus also  $(S \times T) \cap P^{\leq} = ((S' \times T) \cap P^{\leq}) \cup ((S'' \times T) \cap P^{\leq})$ . Since computing a maximum is decomposable we therefore get  $\mathcal{D}_B^{S \times T}(y) = \max \left\{ \mathcal{D}_B^{S' \times T}(y), \mathcal{D}_B^{S'' \times T}(y) \right\}$ . Analogously,  $\mathcal{D}_B^{S \times T}(y) = \max \left\{ \mathcal{D}_B^{S \times T'}(y), \mathcal{D}_B^{S \times T''}(y) \right\}$ . ■

**Theorem 6.8** *Let  $P$  be a polygonal curve in  $\mathbb{R}^2$  with  $n$  vertices. We can build an  $O(n \log^2 n)$  size data structure in  $O(n \log^2 n)$  time, such that given any query  $(s, t, y)$ , for  $s, t \in P$  (not necessarily vertices) we can compute  $\mathcal{D}_B^{P[s,t] \times P[s,t]}(y)$  in  $O(\log^3 n)$  time.*

**Proof** Our main data structure is a range tree whose leaves store the vertices of  $P$ , ordered along  $P$ . Each internal node  $\nu$  corresponds to some subcurve  $P_\nu = P[p, q]$  for two vertices  $p, q \in P$ . We assume that  $P_\nu$  consists of  $m$  vertices. Each node will store the representation of function  $\mathcal{D}_B^{P_\nu \times P_\nu}(y)$  as an upper envelope. By Theorem 6.5, this envelope has complexity  $O(m \log m)$ . The algorithm to compute this envelope is a divide and conquer algorithm whose recursion tree matches our range tree. Hence, when we compute  $\mathcal{D}_B^{P_\nu \times P_\nu}(y)$  for the root of our tree in  $O(n \log^2 n)$  time we actually also construct all the envelopes associated with the other nodes. In addition, each node will store  $\Delta_{\overleftarrow{P}_\nu}(y)$  and  $\Delta_{\overrightarrow{P}_\nu}(y)$  which require  $O(m)$  space and can be constructed by a divide and conquer approach in  $O(m \log m)$  time. Hence, the resulting data structure requires  $O(n \log^2 n)$  space and can be constructed in  $O(n \log^2 n)$  total time.

If  $s$  and  $t$  lie on the same segment of  $P$  then for any  $y$ , we can compute  $\mathcal{D}_B^{P[s,t] \times P[s,t]}(y)$  in constant time. Thus, let  $s, t$  be two points on  $P$ , and  $s'$  and  $t'$  be the first vertex succeeding  $s$  and preceding  $t$ , respectively. There are  $O(\log n)$  subtrees in our range search tree, whose canonical subcurves  $P[p, q]$  together form  $P[s', t']$ . We can identify the nodes bounding these subtrees in  $O(\log n)$  time. We denote these nodes by  $P_1, \dots, P_k$ , where the index matches the order along  $P$ . Furthermore, we define  $P_0 = P[s, s']$  and  $P_{k+1} = P[t', t]$  and observe:

$$\mathcal{D}_B^{P[s,t] \times P[s,t]}(y) = \max_{0 \leq i \leq j \leq k+1} \mathcal{D}_B^{P_i \times P_j}(y) \quad (6.2)$$

Indeed, by Observation 6.4,  $\mathcal{D}_B^{P[s,t] \times P[s,t]}(y)$  is decomposable. By repeated application of this observation we have that  $\mathcal{D}_B^{P[s,t] \times P[s,t]}(y) = \max_{i,j} \mathcal{D}_B^{P_i \times P_j}(y)$ . Moreover, we only need to consider pairs with  $i \leq j$ , since if  $j < i$ , we have  $P_i \times P_j \not\subseteq P[s,t]$ . Thus, we can compute  $\mathcal{D}_B^{P[s,t] \times P[s,t]}(y')$  by only computing the values  $\mathcal{D}_B^{P_i \times P_j}(y')$  with  $i \leq j$ .

By construction, for each of the  $O(\log^2 n)$  pairs  $S, T$  in our decomposition (Eq. 6.2) it holds that all points in  $S$  precede all points in  $T$ . Thus we can compute  $\mathcal{D}_B^{S \times T}(y')$  in  $O(\log(|S| + |T|)) = O(\log n)$  time by Lemma 6.13. Computing this value for each pair takes  $O(\log^3 n)$  total time. For each subcurve  $P_i$ , we compute  $\mathcal{D}_B^{P_i \times P_i}(y)$  in  $O(\log n)$  time using the data structure for  $\mathcal{D}_B^{P_\nu \times P_\nu}(y)$  of Theorem 6.5 stored in the node  $\nu$  corresponding to  $P_i$  (or in  $O(1)$  time if  $i \in \{0, k+1\}$ ). The lemma follows by taking the maximum of these  $O(\log^2 n)$  values. ■

As with the Hausdorff term we can make the query time sensitive to the complexity of  $P[s, t]$ , and obtain a query time of  $O(\log^3 |P[s, t]|)$ . Theorem 6.2 now follows:

**Theorem 6.2** *Let  $P$  be a polygonal curve in  $\mathbb{R}^2$  with  $n$  vertices. There is an  $O(n \log^2 n)$  size data structure that can be built in  $O(n \log^2 n)$  time such that given a horizontal query segment  $\overline{ab}$  and two query points  $s$  and  $t$  on  $P$  the data structure can report  $\mathbf{D}_F(P[s, t], \overline{ab})$  in  $O(\log^3 |P[s, t]|)$  time.*

## 6.5 Arbitrary orientation queries

In this section we extend our results to arbitrarily oriented query segments, proving Theorem 6.3. We denote by  $\alpha, \beta$  the reals such that the query segment  $\overline{ab}$  coincides with the line  $y = \alpha x - \beta$  (note that vertical query segments can be handled by a rotated version of our data structure for horizontal queries). To avoid degeneracies we explicitly assume that  $\overline{ab}$  is not a vertical segment and we assume  $a$  is left of  $b$ ; the case where  $\overline{ab}$  is vertical can be handled with a separate data structure that rotates the plane by 90 degrees and the case where  $b$  is left of  $a$  is symmetric. Throughout this section we consider some query  $\overline{ab}$  with slope  $\alpha$  and we refer to points ‘left’, ‘right’, ‘above’ and ‘below’ some geometric object. These directions are defined on an intuitive level where ‘top’ and ‘bottom’ follow directions perpendicular to a line with slope  $\alpha$ .

It follows immediately from Equation 6.1 that we can write  $\mathbf{D}_{\mathbb{F}}(P, \overline{ab})$  as the maximum of four terms:  $\|p_1 - a\|$ ,  $\|p_n - b\|$ ,  $\vec{\mathbf{D}}_H(P, \overline{ab})$ , and the *backward pair distance*  $\mathcal{D}_B(\alpha, \beta)$  with respect to  $\alpha$ . The backward pair distance is now defined as:

$$\begin{aligned} \mathcal{D}_B(\alpha, \beta) &= \max \{ \delta_{pq}(\alpha, \beta) \mid (p, q) \in \mathcal{B}(P) \}, & \text{where} \\ \delta_{pq}(\alpha, \beta) &= \min_x \max \{ \| (x, \alpha x + \beta) - p \|, \| (x, \alpha x + \beta) - q \| \}. \end{aligned}$$

In this section we will follow a structure similar to previous sections. First in Section 6.5.1 we present an  $O(n \log n)$  size data structure that supports querying the Hausdorff distance  $\vec{\mathbf{D}}_H(P, \overline{ab})$  in  $O(\log^2 n)$  time. The key insight for our approach is that we can use furthest *point* Voronoi diagrams instead of furthest *segment* Voronoi diagrams. In Section 6.5.2 we present a data structure that efficiently supports querying  $\mathcal{D}_B(\alpha, \beta)$ . In Section 6.5.3 we extend our results to support queries against subcurves of  $P$  as well. This combines our insights from the horizontal queries with our results from Sections 6.5.1 and 6.5.2.

### 6.5.1 The Hausdorff distance term

For any point  $p$  and slope  $\alpha$  we will denote by  $\overleftarrow{p}^\alpha$  the ray with apex  $p$  and slope  $\alpha$  that points in the leftward direction. For any point set  $T$ , we define  $\overleftarrow{T}^\alpha = \{ \overleftarrow{p}^\alpha \mid p \in T \}$  as the set of ‘leftward’ pointing rays whose apex lies at points in  $T$ . Furthermore, we define  $h_{\overleftarrow{p}^\alpha}(x, y)$  to be the directed Hausdorff distance from  $(x, y)$  to the ray  $\overleftarrow{p}^\alpha$ , and denote:

$$h_{\overleftarrow{T}^\alpha}(x, y) := \max \{ h_{\overleftarrow{p}^\alpha}(x, y) \mid p \in T \}.$$

In previous sections, we used furthest segment Voronoi diagrams to store the function  $h_{T^{\leftarrow\alpha}}(x, y)$  as a persistent red-black tree. This approach relied on the fact that the slope  $\alpha$  was fixed and thus the set of segments was fixed also. Now we show that we can represent  $h_{T^{\leftarrow\alpha}}(x, y)$  using farthest point Voronoi diagrams instead.

Let  $CH(T)$  be the convex hull of  $T$ . For fixed  $\alpha$ , we denote by  $\ell_{(x,y)}$  the line that is perpendicular to a line with slope  $\alpha$  that goes through  $(x, y)$ . We show that for an arbitrary point  $(x, y)$  the Hausdorff distance to  $T^{\leftarrow\alpha}$  is realised by either a point in  $CH(T)$  that lies left of  $\ell_{(x,y)}$  or the distance to the topmost or bottommost halfline in  $T^{\leftarrow\alpha}$ :

**Lemma 6.14** *Let  $T^{\leftarrow\alpha}$  be a collection of halflines containing  $p^{\leftarrow\alpha}$  as a topmost and  $q^{\leftarrow\alpha}$  as a bottommost halfline, respectively (always with respect to  $\alpha$ ). We have that*

$$h_{T^{\leftarrow\alpha}}(x, y) = \max\{h_{p^{\leftarrow\alpha}}(x, y), h_{q^{\leftarrow\alpha}}(x, y), \max\{\|s - (x, y)\| \mid s \in CH(T) \cap \ell_{(x,y)}^{\leftarrow\alpha}\}\}.$$

**Proof** Let  $t \in T$  be the point realizing  $h_{T^{\leftarrow\alpha}}(x, y) = \max\{h_{s^{\leftarrow\alpha}}(x, y) \mid s \in T\}$ , for some query point  $(x, y)$ . Without loss of generality, we can assume  $\alpha = 0$ , and hence the line  $\ell_{(x,y)}$  is vertical. We distinguish two cases depending on the position of  $t$ , (Figure 6.10).

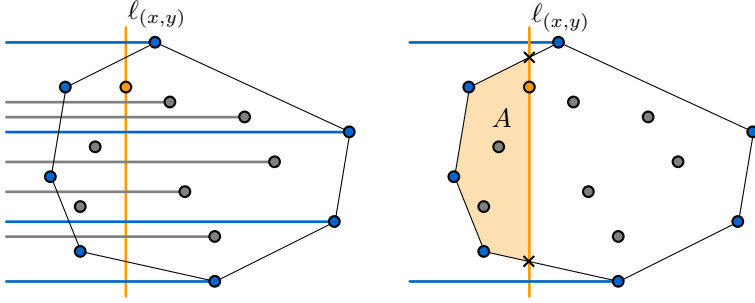
**Case (a):  $t$  lies right of  $\ell_{(x,y)}$ .** In this case,  $h_{T^{\leftarrow\alpha}}(x, y)$  must be given by the vertical distance  $|y_t - y|$ , therefore we must have that  $|y_t - y| = \max\{|y_p - y|, |y_q - y|\}$ . It follows that  $t^{\leftarrow\alpha}$  must be the topmost or bottommost halfline out of all halflines in  $T^{\leftarrow\alpha}$ .

**Case (b):  $t$  lies left of  $\ell_{(x,y)}$ .** Observe that for all points  $u \in T$  left of  $\ell_{(x,y)}$  it must be that:

$$h_{u^{\leftarrow\alpha}}(x, y) = \|u - (x, y)\|.$$

Consider all points  $T' \subset T$  left of  $\ell_{(x,y)}$ . Per definition, the point set  $T'$  is contained in the area  $A$  bound by  $CH(T)$  and the line  $\ell_{(x,y)}$  and since this area is the intersection of two convex areas, the area  $A$  itself is convex. Since  $(x, y)$  is contained in the convex area  $A$ , it follows that the farthest point to  $(x, y)$  is a vertex of the boundary of  $A$ . Thus, the farthest point is either a point in  $CH(T)$  or one of the two vertices of  $A$  corresponding to the point of intersection between  $CH(T)$  and  $\ell_{(x,y)}$ . In the first case,  $t \in CH(T)$ . In the second case the distance between  $(x, y)$  and that vertex is their vertical distance which can be at most the vertical distance between  $(x, y)$  and the topmost/bottommost vertex of  $T$ . ■

Our data structure for the Hausdorff distance term will store  $CH(T)$  in a 1D-range tree whose internal nodes store FPDs. We show this allows us to evaluate  $h_{T^{\leftarrow\alpha}}(x, y)$  and  $h_{T^{\rightarrow\alpha}}(x, y)$  for a query point  $(x, y)$  and slope  $\alpha$ .



**Figure 6.10** (a) For all points  $u$  right of  $\ell_{(x,y)}$  the distance  $h_{\leftarrow \alpha}(x, y)$  is the vertical difference between  $u$  and  $y$  hence the maximal distance comes from either the highest or lowest point  $u$ . (b) The area  $A$  is the intersection between the area bounded by  $CH(T)$  and all points left of  $\ell_{(x,y)}$ .

**Lemma 6.15** Let  $T$  be a set of  $n$  points in  $\mathbb{R}^2$ . In  $O(n \log n)$  time we can construct a data structure of size  $O(n \log n)$  so that given a query point  $(x, y)$  and query slope  $\alpha$  we can compute  $h_{\leftarrow \alpha}(x, y)$  in  $O(\log^2 n)$  time.

**Proof** Consider a clockwise traversal of the convex hull of  $T$  that visits every vertex twice. Let  $t_1, \dots, t_{2k}$  denote the vertices in this order (so  $t_{i+k} = t_i$ ). We store these vertices  $t_1, \dots, t_{2k}$  in the leaves of a range tree. Each internal node  $\nu$  corresponds to some contiguous subsequence  $T_\nu = t_i, \dots, t_j$  of these vertices, and stores the furthest point Voronoi diagram (FPVD) of  $T_\nu$  (removing duplicate vertices). Since the FPVD has linear size and the points are in convex position, it can be computed in linear time [7]. Thus our data structure has size  $O(n \log n)$  and can be computed in  $O(n \log n)$  time.

To answer a query, we first find the bottom- and topmost points of  $CH(T)$  (and thus of  $T$ ) with respect to slope  $\alpha$ . Let  $q$  and  $p$  be these points, respectively. We now find a contiguous subsequence  $t_i, \dots, t_j$  of the vertices of  $CH(T)$  in the halfplane left of  $\ell_{(x,y)}$ . Note that since  $t_1, \dots, t_{2k}$  traverses  $CH(T)$  twice such a contiguous sequence exists. We then query the data structure to obtain  $O(\log n)$  internal nodes  $\nu$  whose associated sets  $T_\nu$  together represent  $t_i, \dots, t_j$ , and query their furthest point Voronoi diagrams to find the point  $s$  in  $t_i, \dots, t_j$  furthest from  $(x, y)$ . We report the maximum of  $h_{\leftarrow \alpha}(x, y)$ ,  $h_{\leftarrow \alpha}(x, y)$ , and  $\|s - (x, y)\|$ . By Lemma 6.14 this is  $h_{\leftarrow \alpha}(x, y)$ .

Finding  $p, q, t_i$ , and  $t_j$  takes  $O(\log n)$  time. This is dominated by the  $O(\log^2 n)$  time to query all FPVDs. Thus, we can compute  $h_{\leftarrow \alpha}(x, y)$  in  $O(\log^2 n)$  time. Symmetrically, we can query  $h_{\rightarrow \alpha}(x, y)$  in  $O(\log^2 n)$  time. ■

The above lemma provides  $O(\log^2 n)$  query time in  $O(n \log n)$  space. We show through using similar ideas to those of range minimum queries [11, 19] that we can expand this approach to achieve  $O(\log n)$  query time using  $O(n^2)$  space:

**Lemma 6.16** *Let  $T$  be a set of  $n$  points in  $\mathbb{R}^2$ . In  $O(n^2)$  time we can construct a data structure of size  $O(n^2)$  so that given a query point  $(x, y)$  and query slope  $\alpha$  we can compute  $h_{\overleftarrow{T}}(x, y)$  in  $O(\log n)$  time.*

**Proof** For every vertex  $t_i$  on  $CH(T)$ , and every  $\ell \in 1, \dots, \log n$  we store the FPVD of  $t_i, \dots, t_{i+2^\ell}$ . For a fixed vertex  $t_i$  we hence store a  $\sum_{\ell=1}^{\log n} O(2^\ell) = O(2^{\log n}) = O(n)$  space structure and thus we use  $O(n^2)$  space in total. Building the FPVDs takes linear time, so the total construction time is  $O(n^2)$  as well.

To evaluate  $h_{\overleftarrow{T}}(x, y)$  for some query point  $(x, y)$  and query slope  $\alpha$  we again have to find the furthest point among some interval  $t_i, \dots, t_j$  along  $CH(T)$ . We compute the largest value  $\ell$  such that  $2^\ell \leq j - i$ . This allows us to decompose the “query range”  $t_i, \dots, t_j$  into two overlapping intervals  $t_i, \dots, t_{i+2^\ell}$  and  $t_{j-2^\ell}, \dots, t_j$  for which we have pre-stored the FPVD. We can thus query both these FPVDs, in  $O(\log n)$  time, and report the furthest point found. Computing  $\ell$  and finding the points  $t_{i+2^\ell}$  and  $t_{j-2^\ell}$  takes  $O(\log n)$  time as well, by a binary search on  $t_i, \dots, t_j$ . Hence the total query time is  $O(\log n)$ . ■

The above two lemmas, combined with Corollary 6.1, show that we can compute  $\overrightarrow{D}_H(P, \overline{ab})$  by querying either of these data structures and we conclude:

**Theorem 6.9** *Let  $P$  be a polygonal curve in  $\mathbb{R}^2$  with  $n$  vertices.*

- *In  $O(n \log n)$  time we can construct a data structure of size  $O(n \log n)$  so that given a query segment  $\overline{ab}$ ,  $\overrightarrow{D}_H(P, \overline{ab})$  can be computed in  $O(\log^2 n)$  time.*
- *In  $O(n^2)$  time we can construct a data structure of size  $O(n^2)$  so that given a query segment  $\overline{ab}$ ,  $\overrightarrow{D}_H(P, \overline{ab})$  can be computed in  $O(\log n)$  time.*

## 6.5.2 The backward pair distance term

Let  $(p, q) \in P^\leq$  be an ordered pair. Then there is an interval of slopes  $\alpha$  for which  $(p, q)$  is backward with respect to the orientation  $\alpha$ . We restrict the function  $\delta_{pq}(\alpha, \beta)$  to this interval of  $\alpha$  values. Hence, each such function  $\delta_{pq}(\alpha, \beta)$  that represents the backward pair distance to the pair  $(p, q)$  (on a restricted domain) is a partially defined, constant algebraic degree, constant complexity, bivariate function. The backward pair distance  $\mathcal{D}_B$  is the upper envelope of  $O(n^2)$  such functions. It follows by Sharir [185] that this envelope has complexity  $O(n^{4+\varepsilon})$ , for some arbitrarily small constant  $\varepsilon > 0$ , and can be computed in  $O(n^{4+\varepsilon})$  time. Via that same representation, evaluating  $\mathcal{D}_B(\alpha, \beta)$  for some given  $\alpha, \beta$  takes  $O(\log n)$  time and we conclude:

**Theorem 6.10** *Let  $P$  be an  $n$ -vertex polygonal curve in  $\mathbb{R}^2$  and  $\varepsilon > 0$  be an arbitrarily small constant. There exists an  $O(n^{4+\varepsilon})$  size data structure that can be built in  $O(n^{4+\varepsilon})$  time such that given a query segment  $\overline{ab}$ , it can report  $\mathcal{D}_B(\overline{ab})$  in  $O(\log n)$  time.*

This theorem, together with Theorem 6.9, then gives an  $O(n^{4+\varepsilon})$  size data structure that supports  $O(\log n)$  time Fréchet distance queries and we conclude:

**Theorem 6.11** *Let  $P$  be a polygonal curve in  $\mathbb{R}^2$  with  $n$  vertices and let  $\varepsilon > 0$  be an arbitrarily small constant. There is an  $O(n^{4+\varepsilon})$  size data structure that can be built in  $O(n^{4+\varepsilon})$  time such that given an arbitrary query segment  $\overline{ab}$  it can report  $\mathbf{D}_{\mathbb{F}}(P[s, t], \overline{ab})$  in  $O(\log n)$  time.*

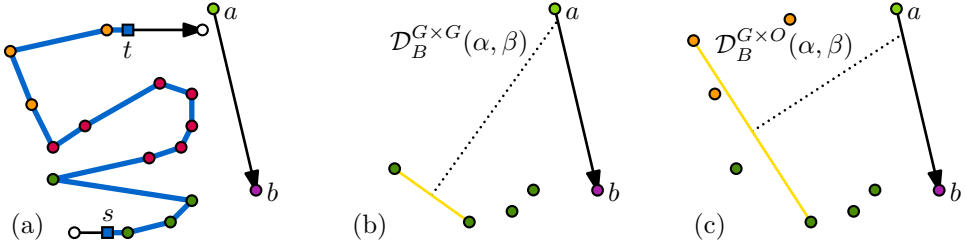
### 6.5.3 Subcurve queries

Next, we describe how to support querying against subcurves  $P[s, t]$  of  $P$  in  $O(\log^4 n)$  time. We use the same approach as in the horizontal query segment case: we store the vertices of  $P$  into the leaves of a range tree where each internal node  $\nu$  corresponds to some canonical subcurve  $P_\nu$ , so that any subcurve  $P[s, t]$  can be represented by  $O(\log n)$  nodes in the range tree.

**The Hausdorff distance term.** Since computing the directed Hausdorff distance is decomposable, using this approach with the data structure of Theorem 6.9 immediately gives us a data structure that allows us to compute  $\vec{\mathbf{D}}_H(P[s, t], \overline{ab})$  in  $O(\log^2 n)$  time. Since the space usage satisfies the recurrence  $S(n) = 2S(n/2) + O(n^2)$ , this uses  $O(n^2)$  space in total.

**The backward pair distance term.** To compute the backward pair distance term, we store the data structure of Theorem 6.10 at every node of the tree. Using these data structures, we can compute the backward pair distance  $\mathcal{D}_B^{S \times S}(\alpha, \beta)$  for every subtrajectory  $S$  stored at a node of the range tree. What remains to compute is the distance  $\mathcal{D}_B^{S \times T}$  for all subtrajectories  $S$  and  $T$  that are contained in  $P[s, t]$  and stored at two different nodes in the tree. We again store additional data structures at every node of the tree that allow us to efficiently compute the  $O(\log^2 n)$  values  $\mathcal{D}_B^{S \times T}$  for every such pair  $S, T \subset P[s, t]$  (refer to Figure 6.11 for an example of the approach).

Let  $S$  and  $T$  be (the vertices of) two such canonical subcurves, with all vertices of  $S$  occurring before  $T$  along  $P$ . Analogous to Section 6.4 we will argue that for some given  $\alpha$  and  $\beta$  the functions  $x \mapsto h_{T^\leftarrow \alpha}(x, \alpha x + \beta)$  and  $x \mapsto h_{S^\rightarrow \alpha}(x, \alpha x + \beta)$  are monotonically increasing and decreasing, respectively, and that the intersection point of (the graphs of) these functions corresponds to the contribution of the backward pairs in  $S \times T$ . Our goal is to build data structures storing  $S$  and  $T$  that, given a query  $\alpha, \beta$ , allow us to efficiently compute the intersection point of these functions. As we will argue next, we can use the furthest point Voronoi diagram data structure of Lemma 6.16 to support such queries in  $O(\log^2 n)$  time.



**Figure 6.11** (a) A polygonal curve  $P$  and a subcurve  $P[s, t]$  in blue and a query  $\overline{ab}$  with slope  $\alpha$  and offset  $\beta$ . The subcurve contains three canonical subsets ( $G$ ) with green vertices, ( $R$ ) with red vertices and ( $O$ ) with orange vertices. (b) The maximal backward pair distance  $\mathcal{D}_B^{G \times G}(\alpha, \beta)$  can be computed by querying the data structure associated with the node in the range tree that stores  $G$ . (c) The maximal backward pair distance  $\mathcal{D}_B^{G \times O}(\alpha, \beta)$  can be computed with a separate data structure.

We generalize some of our earlier geometric observations to arbitrary orientations. Let  $p, q$  be vertices of  $P$ , and let  $S$  and  $T$  be subsets of vertices of  $P$ . We define

$$\delta'_{pq}(\alpha, \beta) = \min_x \max \left\{ h_{\leftarrow \alpha}((x, \alpha x + \beta)), h_{\rightarrow \alpha}((x, \alpha x + \beta)) \right\} \quad \text{and}$$

$$\mathcal{D}_B^{S \times T}(\alpha, \beta) = \max \left\{ \delta'_{pq}(\alpha, \beta) \mid (p, q) \in (S \times T) \cap P^{\leq} \right\}.$$

and prove the following by essentially rotating the plane so that the query segment becomes horizontal and applying the appropriate lemmas from earlier sections:

**Lemma 6.17** *Let  $P$  be partitioned into two subcurves  $S$  and  $T$  with all vertices in  $S$  occurring on  $P$  before the vertices of  $T$ . We have that*

$$\mathcal{D}_B(\alpha, \beta) = \mathcal{D}_B^{P \times P}(\alpha, \beta) = \max \left\{ \mathcal{D}_B^{S \times S}(\alpha, \beta), \mathcal{D}_B^{T \times T}(\alpha, \beta), \mathcal{D}_B^{S \times T}(\alpha, \beta) \right\}.$$

**Proof** Consider the rotation that transforms the line  $y = \alpha x + \beta$  into a horizontal line, keeping  $a$  to the left of  $b$ . Applying this rotation to  $P$  and  $\overline{ab}$  produces an instance of the horizontal query problem. Since the same rotation is applied to all vertices of  $P$ , the set of backward pairs and the relative distances from the query segment to them remain unchanged. Therefore, we can apply the results for the horizontal case, in particular Lemma 6.4, to obtain that  $\mathcal{D}_B(\alpha, \beta) = \mathcal{D}_B^{P \times P}(\alpha, \beta)$ .

For the second equality, we make two observations. The first observation is that  $S$  and  $T$  are a partition of  $P$  and thus:

$$P \times P = (S \times S) \cup (T \times T) \cup (S \times T) \cup (T \times S).$$

The second observation is that all vertices of  $S$  occur before  $T$  and thus:

$$(T \times S) \cap P^{\leq} = \emptyset.$$

These observations plus basic set theory show the second equality.  $\blacksquare$

As the values  $\mathcal{D}_B^{S \times S}(\alpha, \beta)$  and  $\mathcal{D}_B^{T \times T}(\alpha, \beta)$  can be computed by querying the data structure of Theorem 6.10, we focus our analysis on computing  $\mathcal{D}_B^{S \times T}(\alpha, \beta)$ :

**Lemma 6.18** *Let  $S$  and  $T$  be subsets of vertices of  $P$ , with  $S$  occurring before  $T$  along  $P$  and let  $\alpha, \beta$  denote some query parameters. The function  $x \mapsto h_{T^\alpha}^\leftarrow(x, \alpha x + \beta)$  is monotonically increasing, whereas  $x \mapsto h_{S^\alpha}^\rightarrow(x, \alpha x + \beta)$  is monotonically decreasing. These functions intersect at a point  $(x^*, \alpha x^* + \beta)$ , for which  $\mathcal{D}_B^{S \times T}(\alpha, \beta) = h_{T^\alpha}^\leftarrow(x^*, \alpha x^* + \beta) = h_{S^\alpha}^\rightarrow(x^*, \alpha x^* + \beta)$ .*

**Proof** Consider the rotation that transforms the line  $y = \alpha x + \beta$  into a horizontal line, keeping  $a$  to the left of  $b$ . Note again that this preserves distances. It follows then from Observation 6.2 that  $x \mapsto h_{T^\alpha}^\leftarrow(x, \alpha x + \beta)$  and  $x \mapsto h_{S^\alpha}^\rightarrow(x, \alpha x + \beta)$  are monotonically increasing and decreasing, respectively (since after the rotation,  $a$  is left of  $b$ ). Furthermore, by Lemma 6.6 they intersect in a single point  $(x^*, y) = (x^*, \alpha x^* + \beta)$ , at which  $\mathcal{D}_B^{S \times T}(\alpha, \beta) = h_{S^\alpha}^\rightarrow = (x^*, y) = h_{T^\alpha}^\leftarrow(x^*, y)$ .  $\blacksquare$

**Querying  $\mathcal{D}_B^{S \times T}(\alpha, \beta)$ .** Consider the predicate:

$$Q(x) = h_{T^\alpha}^\leftarrow(x, \alpha x + \beta) < h_{S^\alpha}^\rightarrow(x, \alpha x + \beta).$$

It follows from Lemma 6.18 that there is a single value  $x^*$  so that  $Q(x) = \text{FALSE}$  for all  $x < x^*$  and  $Q(x) = \text{TRUE}$  for all  $x \geq x^*$ . Moreover,  $x^*$  realizes  $\mathcal{D}_B^{S \times T}(\alpha, \beta)$ . By storing  $S$  and  $T$ , each in a separate copy of the data structure of Lemma 6.16, we can evaluate  $Q(x)$ , for any value  $x$ , in  $O(\log n)$  time. We then use parametric search [158] to find  $x^*$  in  $O(\log^2 n)$  time. Note that this approach is an  $O(\log n)$  factor slower compared to the approach we used for horizontal queries (Lemma 6.13).

**Lemma 6.19** *Let  $S, T$  be subsets of vertices of  $P$  such that all vertices in  $S$  precede all vertices in  $T$ , stored in the data structure of Lemma 6.16. For any query  $\alpha, \beta$  we can compute  $\mathcal{D}_B^{S \times T}(\alpha, \beta)$  in  $O(\log^2 n)$  time.*

**Proof** We treat  $x^*$  as a variable, and evaluate  $Q$  on the (unknown) value  $x^*$ . While doing so, we maintain an interval that is known to contain  $x^*$ . Initially this interval is  $\mathbb{R}$  itself. When the algorithm to evaluate  $Q(x^*)$  reaches a comparison involving  $x^*$ , we obtain a constant degree polynomial in  $x^*$ . Indeed, all comparisons in the query algorithm of Lemma 6.16 test if the query point lies left or right of some line, or compare the Euclidean distance between two pairs of points.

We compute the (constantly many) roots of this polynomial and evaluate  $Q$  again at each root. This shrinks the interval known to contain  $x^*$ , and allows the evaluation of  $Q(x^*)$  to proceed. When the evaluation of  $Q(x^*)$  finishes, the interval known to contain  $x^*$  has shrunk to a single point,  $x^*$ , or  $x^*$  can be computed from it by solving one more equation in constant time.

Evaluating  $Q$  takes  $O(\log n)$  time, and thus encounters at most  $O(\log n)$  comparisons. For each such comparison we again evaluate  $Q$  at a constant number of roots, taking  $O(\log n)$  time each. Hence the total time required to compute  $x^*$  is  $O(\log^2 n)$ . Given  $x^*$  we can obtain  $\mathcal{D}_B^{S \times T}(\alpha, \beta) = h_{T^\alpha}^\alpha(x^*, \alpha x + \beta)$  in  $O(\log n)$  time. ■

Given the above lemmas, our final data structure works as follows: the core data structure is a range tree on the traversal of the vertices in  $T$ . For every node  $\nu$  of the range tree on  $P$  we store:

- (i) the data structure of Theorem 6.10 built on its canonical subcurve  $P_\nu$  and
- (ii) the data structure of Lemma 6.16 built on the vertices of  $P_\nu$ .

The total space usage of the data structure follows the recurrence  $S(n) = 2S(n/2) + O(n^{4+\varepsilon})$ , which solves to  $O(n^{4+\varepsilon})$ . To query the data structure with some subcurve  $P[s, t]$  from some vertex  $s$  to a vertex  $t$  we again find the  $O(\log n)$  nodes whose canonical subcurves together define  $P[s, t]$ . For each of the  $O(\log n)$  nodes we query the data structure of Theorem 6.10 in  $O(\log^2 n)$  total time. For every pair of nodes we run the algorithm from Lemma 6.19 for each pair. The total running time is then  $O(\log^4 n)$ . As before, the procedure can easily be extended to the case where  $s$  and  $t$  lie on the interior of an edge. We conclude:

**Theorem 6.12** *Let  $P$  be a polygonal curve in  $\mathbb{R}^2$  with  $n$  vertices, and let  $\varepsilon > 0$ . There is an  $O(n^{4+\varepsilon})$  size data structure that can be built in  $O(n^{4+\varepsilon})$  time such that given an arbitrary query segment  $\overline{ab}$  and two query points  $s$  and  $t$  on  $P$  it can report  $\mathcal{D}_B^{P[s,t] \times P[s,t]}(\alpha, \beta)$  in  $O(\log^4 n)$  time.*

For any points  $s, t$  on  $P$ , using Theorem 6.9 and Theorem 6.12 we can compute all four terms

$$\|s - a\|, \quad \|t - b\|, \quad \vec{\mathbf{D}}_H(P[s, t], \overline{ab}), \quad \text{and } \mathcal{D}_B^{P[s,t] \times P[s,t]}(\alpha, \beta)$$

in  $O(\log^4 n)$  total time. It follows that we can efficiently answer Fréchet distance queries against subcurves. Since all computations follow from balanced decompositions of  $P[s, t]$  we can immediately apply the same analysis as presented in

make the runtime sensitive to  $|P[s, t]|$ . This allows us to conclude Theorem 6.3:

**Theorem 6.3** *Let  $P$  be a polygonal curve in  $\mathbb{R}^2$  with  $n$  vertices, and let  $\varepsilon > 0$  be an arbitrarily small constant. There is an  $O(n^{4+\varepsilon})$  size data structure that can be built in  $O(n^{4+\varepsilon})$  time such that given an arbitrary query segment  $\overline{ab}$  and two query points  $s$  and  $t$  on  $P$  it can report  $\mathbf{D}_F(P[s, t], \overline{ab})$  in  $O(\log^4 |P[s, t]|)$  time.*

In the following section we show that by we can extend this result to introduce a tradeoff between the required query time and the space requirement.

## 6.6 Space time tradeoff

We can obtain a space vs query time trade off for queries. Let  $k \in [1..n]$  be a parameter. In all our approaches, we constructed a recursive tree on  $P$  where at every node we split  $P$  into two roughly equal-size subtrajectories. We trim the recursion tree on  $P$  at nodes  $\nu$  of size  $O(k)$  and built a separate data structure on each leaf of this trimmed tree. Specifically, we show the following:

**Theorem 6.13** *Let  $P$  be a polygonal curve in  $\mathbb{R}^2$  with  $n$  vertices,  $\varepsilon > 0$  be an arbitrarily small constant, and  $k \in [1..n]$  be some fixed parameter. In  $O(nk^{3+\varepsilon} + n^2)$  time we can construct a data structure of size  $O(nk^{3+\varepsilon} + n^2)$  so that given a query segment  $\overline{ab}$ ,  $\mathcal{D}_B(\overline{ab})$  can be computed in  $O((n/k) \log^2 n)$  time.*

**Proof** Let  $T$  denote the tree resulting from trimming the recursion tree of  $P$  such that all leaves contain  $O(k)$  vertices (the tree  $T$  consists of the top  $\log(n/k)$  levels of the full recursion tree). Let  $L(T)$  denote the set of leaves of  $T$ , each of which thus corresponds to a subcurve of length  $O(k)$ . Denote by  $\ell(\nu)$  and  $r(\nu)$  the left and right child of  $\nu$ , respectively. By repeated application of the second equality in Lemma 6.17 we have that

$$\mathcal{D}_B^{P \times P}(\alpha, \beta) = \max \left\{ \max_{\nu \in T} \mathcal{D}_B^{P_{\ell(\nu)} \times P_{r(\nu)}}(\alpha, \beta), \max_{\nu \in L(T)} \mathcal{D}_B^{P_\nu \times P_\nu}(\alpha, \beta) \right\}.$$

At every leaf of our trimmed tree  $T$  we store the data structure of Theorem 6.10, and at every internal node the data structure of Lemma 6.16. Since there are  $O(n/k)$  leaves in  $T$  and each leaf contains  $k$  vertices, the space required by all Theorem 6.10 data structures is  $O((n/k)k^{4+\varepsilon}) = O(nk^{3+\varepsilon})$ . The total size for all Lemma 6.16 data structures follows the recurrence  $S(n) = 2S(n/2) + O(n^2)$  which solves to  $O(n^2)$ . Hence, the total space used is  $O(nk^{3+\varepsilon} + n^2)$ . The preprocessing time is  $O(nk^{3+\varepsilon} + n^2)$ .

To answer a query  $(\alpha, \beta)$  we now query the Theorem 6.10 data structures at the leaves of  $T$  in  $O(\log k)$  time each. For every internal node  $\nu$  we use Lemma 6.19 to compute the contribution of  $\mathcal{D}_B^{P_{\ell(\nu)} \times P_{r(\nu)}}(\alpha, \beta)$  in  $O(\log^2 n)$  time. Since  $(n/k)$  is at least 1, we can write the total query time as:

$$O((n/k) \log k + (n/k) \log^2 n) = O((n/k) \log^2 n),$$

which concludes the proof. ■

The value  $k$  serves as the parameter for the tradeoff between the space used by our solution and the query time. For example, choosing  $k = n^{1/3}$  yields an  $O(n^{2+\varepsilon})$  size data structure supporting  $O(n^{2/3} \log^2 n)$  time queries. We can extend this idea to support subcurve queries in  $O((n/k) \log^2 n + \log^4 n)$  time as well, giving us the following result:

**Theorem 6.14** *Let  $P$  be a polygonal curve in  $\mathbb{R}^2$  with  $n$  vertices,  $\varepsilon > 0$  be an arbitrarily small constant, and  $k \in [1..n]$  be some fixed parameter. In  $O(nk^{3+\varepsilon} + n^2)$  time we can construct a data structure of size  $O(nk^{3+\varepsilon} + n^2)$  so that given a query segment  $\overline{ab}$  and two points  $s$  and  $t$  on  $P$ ,  $\mathcal{D}_B^{P[s,t] \times P[s,t]}(\overline{ab})$  can be computed in  $O((n/k) \log^2 n + \log^4 n)$  time.*

**Proof** We can also apply the time space trade off in case of subcurve queries. We use essentially the structure as above with the appropriate associated data structures with a slight variation: we now denote by  $T$  the recursion tree on  $P$ , but instead of trimming the tree such that all leaves contain  $O(k)$  vertices we keep the tree as is. On every internal node  $\nu$ , we build the Lemma 6.16 data structure on the vertices  $P_\nu$  associated with that node. This is a quadratic size data structure hence the total size used by this construction is given by the recursion:

$$S(n) = S(n/2) + O(n^2),$$

which solves to  $O(n^2)$  space. In addition, we select all internal nodes that store  $\leq 2k$  vertices and for every such node we build the Theorem 6.10 data structure on all vertices associated with that node. We augment our tree with neighbor pointers between nodes of equal height (*level links* [159]) and every internal node that stores more than  $2k$  vertices receives a pointer to the leftmost descendent of size  $O(k)$ . Since there are  $O(n/k)$  such internal nodes the total space used by our construction is  $O(nk^{3+\varepsilon} + n^2)$ .

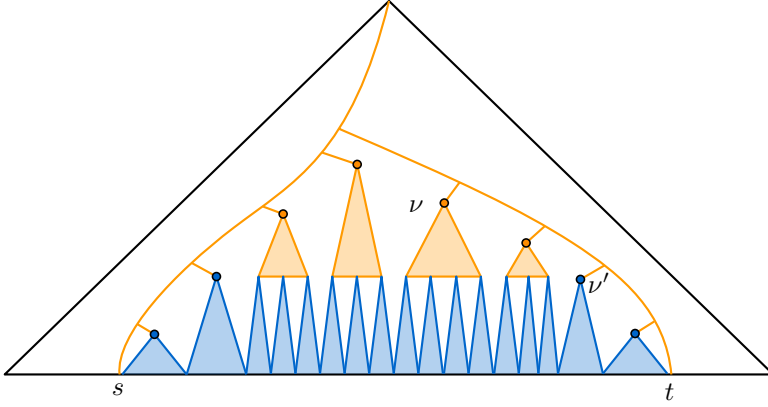
Given a pointer to the segments of  $P$  that contain  $s$  and  $t$ , we can again find a set  $Q$  of  $O(\log n)$  nodes  $\nu$  whose canonical subcurves together represent the query subcurve  $P[s, t]$ . We then have

$$\mathcal{D}_B^{P[s,t] \times P[s,t]}(\alpha, \beta) = \max \left\{ \max_{\nu \in Q} \mathcal{D}_B^{P_\nu \times P_\nu}(\alpha, \beta), \max_{\mu, \nu \in Q, \mu \neq \nu} \mathcal{D}_B^{P_\mu \times P_\nu}(\alpha, \beta) \right\}. \quad (6.3)$$

Observe that for each of distinct nodes  $\mu, \nu \in Q$  we can compute  $\mathcal{D}_B^{P_\mu \times P_\nu}(\alpha, \beta)$  in  $O(\log^2 n)$  time using the Lemma 6.16 data structures stored at nodes  $\mu$  and  $\nu$  (see the analysis in Lemma 6.19). It follows that we can compute the second category of terms in Equation 6.3 in  $O(\log^4 n)$  total time by querying the associated data structure for each of the  $O(\log^2 n)$  pairs of nodes  $\mu$  and  $\nu$ .

To compute  $\mathcal{D}_B^{P_\nu \times P_\nu}(\alpha, \beta)$  for some  $\nu \in Q$  there are two cases.

- If  $P_\nu$  contains at most  $2k$  vertices, we can compute the term directly by querying the Theorem 6.10 data structure of node  $\nu$  in  $O(\log k)$  time.
- If  $P_\nu$  contains more than  $2k$  vertices we traverse the pointer from  $\nu$  to the leftmost descendent of size  $O(k)$ . By traversing the neighbor pointers in our tree, we obtain the at most  $O(n/k)$  descendants of  $\nu$  whose vertices partition  $P_\nu$  and that each have a Theorem 6.10 data structure (see the node  $\nu$  in Figure 6.12).



**Figure 6.12** A query with a subcurve  $P[s, t]$  selects  $O(\log n)$  nodes whose associated vertices partition  $P[s', t']$ . Whenever the subcurve  $P_{\nu'}$  associated with a node  $\nu'$  is small enough (the blue nodes), we can directly query their Theorem 6.10 data structures. When the subtree is too large (refer to the node  $\nu$ ) we visit its top part (in orange) computing the their contribution to  $\mathcal{D}_B^{P[s, t] \times P[s, t]}(\alpha, \beta)$  using the Lemma 6.16 data structures until we reach the (blue) subtrees of size  $O(k)$ . The total number of nodes visited is  $O(n/k)$ .

Given these  $O(n/k)$  nodes, where every node stores  $O(k)$  vertices, we query their associated Theorem 6.10 data structure in  $O((n/k) \log n)$  total time. For every node  $\mu$  that stores  $O(k)$  vertices, this gives us the value  $\mathcal{D}_B^{P_\mu \times P_\mu}(\alpha, \beta)$ . What remains, is to compute the maximum of  $\mathcal{D}_B^{P_\mu \times P_{\mu'}}(\alpha, \beta)$  for every pair  $\mu, \mu'$  contained in  $\nu$  (where  $P_\mu$  and  $P_{\mu'}$  contain more than  $2k$  vertices). We obtain this value, by querying the associated Lemma 6.16 data structures of all descendants of  $\nu$  that store more than  $2k$  vertices (this data structure automatically compute the maximum between any two pairs between their left and right children). Since there are at most  $O(n/k)$  descendants of  $\nu$  which store more than  $2k$  vertices this takes  $O((n/k) \log^2 n)$  total time. The maximum of these terms is the value  $\mathcal{D}_B^{P_\nu \times P_\nu}(\alpha, \beta)$ .

It follows that the query time is at most  $O(\log^4 n + (n/k) \log^2 n)$  and this concludes the proof. ■

We show some applications of our result before we present our concluding remarks.

## 6.7 Applications

In this section we discuss how to apply our result to improve the results of prior publications on curve simplification and Fréchet distance under translation.

**Curve simplification.** Our tools can be used for local curve simplification under the Fréchet distance. In curve simplification, the input is a polygonal curve  $P = (p_1, p_2, \dots, p_n)$  with  $n$  vertices and some parameter  $\delta$  and the goal is to compute a polygonal curve  $S$ , whose vertices are vertices of  $P$ , that is within distance  $\delta$  of  $P$  for some distance metric and has a minimum number of vertices. The simplification is a *local simplification* if for every edge  $e_{ij}$  in  $S$  from  $p_i$  to  $p_j$ , the distance between  $e_{ij}$  and  $P[i, j]$  is at most  $\delta$ . Curve simplification is well-studied within computational geometry. Recent examples that use the Fréchet distance include algorithms for global curve simplification by van de Kerkhof et al. [194], a polynomial-time algorithm for computing an optimal global simplification for the Fréchet distance by van Kreveld et al. [198], and an  $\Omega(n^3)$  lower bound on computing a local simplification in higher dimensions using  $L_p$  ( $p \neq 2$ ) norms by Bringmann and Chaudhury [33].

The state-of-the-art approach to compute a local  $\delta$ -simplification using Fréchet distance and the  $L_2$  norm is the Imai-Iri line simplification algorithm [119] (see also Godau [101]). This algorithm, originally designed to compute a simplification with respect to the Hausdorff distance, considers all  $O(n^2)$  edges between vertices of  $P$  and computes for every edge  $e_{ij}$  from  $p_i$  to  $p_j$  ( $i < j$ ), the (Fréchet/Hausdorff) distance between  $e_{ij}$  and  $P[p_i, p_j]$  in total  $O(n^3)$  time. They assign the corresponding Fréchet distance as a weight to  $e_{ij}$ . This results in  $O(n^2)$  weighted edges. Computing a minimum link path from  $p_1$  to  $p_n$  in this graph takes  $O(n^2)$  time. Thus, a local  $\delta$ -simplification can be computed in  $O(n^3)$  time and  $O(n^2)$  space.

We can improve this state-of-the-art algorithm by applying Theorem 6.14. Specifically, if we choose our parameter  $k = n^{1/2}$ , we can construct the corresponding data structure in  $O(n^{2.5+\varepsilon})$  time and space (where  $\varepsilon$  is an arbitrarily small positive constant). For each edge  $e_{ij}$ , we can compute the Fréchet distance between  $e_{ij}$  and the subcurve  $P[p_i, p_j]$  in  $O(\sqrt{n} \log^2 n)$  time, and we conclude:

**Theorem 6.15** *Let  $P$  be a polygonal curve in  $\mathbb{R}^2$  with  $n$  vertices,  $\varepsilon > 0$  be an arbitrarily small constant and let  $\delta > 0$ . We can compute a local  $\delta$ -simplification of  $P$  with respect to the Fréchet distance using  $O(n^{5/2+\varepsilon})$  time and space.*

**Fréchet distance queries under translation.** There are many geometric pattern matching applications where one would like to find a transformation (e.g., translation or rotation) that minimizes the Fréchet distance between two input curves. One typical example is handwriting recognition. Our results can be used to compute a data structure for Fréchet distance queries under translation. Given two polygonal curves  $P$  and  $Q$  in the plane, the goal is to find an optimal translation of  $Q$  such that the Fréchet distance between the translated version of  $P$  and  $Q$  is minimized.

This problem can be solved in  $O((nm)^3(n+m)^2 \log(n+m))$  time [10] where  $n$  and  $m$  are the number of vertices of  $P$  and  $Q$ , respectively. Recently, Gudmundsson et al. [106] (full version [107]) studied the query version of this problem, where the goal is to preprocess  $P$ , such that given a query curve  $Q$  and two points  $s$  and  $t$  on  $P$ , one can find the translation of  $Q$  that minimizes the Fréchet distance between  $P[s, t]$  and  $Q$  efficiently. They study this query version in a restricted setting, where  $Q$  is a horizontal segment. Their data structure uses  $O(n^2 \log^2 n)$  space and allows for  $O(\log^{32} n)$  time queries. By applying our data structure, we obtain the following result:

**Theorem 6.16** *Let  $P$  be a polygonal curve in  $\mathbb{R}^2$  with  $n$  vertices. There is an  $O(n \log^2 n)$  size data structure that can be built in  $O(n \log^2 n)$  time such that given any two points  $s, t \in P$  and a horizontal query segment  $\overline{ab}$ , one can report the translation of  $\overline{ab}$  that minimizes its Fréchet distance to  $P[s, t]$  in  $O(\log^{12} n)$  time.*

**Proof** The approach by Gudmundsson et al. [106] essentially uses four levels of parametric search [158] to turn a Fréchet distance data structure into a data structure that can find the translation of  $\overline{ab}$  that minimises the distance to  $P[s, t]$ . We describe the result by Gudmundsson et al. and show how to use our result for their analysis.

Specifically, suppose you have access to a data structure that for a horizontal query segment  $\overline{ab}$  can decide if the Fréchet distance between  $P[s', t']$  and  $\overline{ab}$  is at most  $R$  in  $Q(n)$  time. In our assumption,  $s'$  and  $t'$  are required to be vertices of  $P$ . The authors use the aforementioned data structure for four levels of parametric search. The first two levels are used to, given two points  $s, t$  on  $P$  that are not necessarily vertices, compute the Fréchet distance between  $\overline{ab}$  and the subcurve  $P[s, t]$  (as opposed to the Fréchet distance between  $\overline{ab}$  and  $P$ ). The third level decides (given a fixed  $x$ -coordinate for the point  $a$  of  $\overline{ab}$ ) the vertical translation of  $\overline{ab}$  that minimises the Fréchet distance to  $P[s, t]$ . The fourth level decides the arbitrary translation of  $\overline{ab}$  that minimises the Fréchet distance to  $P[s, t]$ , by deciding, for a given  $x$ -coordinates, whether the endpoint  $a$  of  $\overline{ab}$  lies left or right of this  $x$ -coordinate.

Each parametric search squares the running time of the decision algorithm that it has access to, and their final solution runs in  $O(Q(n)^{16})$  time. By replacing the data structure of de Berg et al. [63] by Theorem 6.2 we note that we can skip the first two levels of parametric search (since our data structure already supports arbitrary subcurves  $P[s, t]$ ), getting the total query time down to  $O(Q(n)^4)$ . With our data structure, we can decide if the Fréchet distance between  $P[s, t]$  and  $\overline{ab}$  is at most  $R$  in  $O(\log^3 n)$  time. Hence our total running time is  $O((\log^3 n)^4) = O(\log^{12} n)$ . ■

Since Theorem 6.2 is essentially used as a black-box solution, replacing it with Theorem 6.3 yields a data structure supporting arbitrarily oriented query segments.

**Theorem 6.17** *Let  $P$  be a polygonal curve in  $\mathbb{R}^2$  with  $n$  vertices, let  $k \in [1..n]$ , and let  $\varepsilon > 0$  be an arbitrarily small constant. There is an  $O(nk^{3+\varepsilon} + n^2)$  size data structure that can be built in  $O(nk^{3+\varepsilon} + n^2)$  time such that given a query segment  $\overline{ab}$  and two points  $s$  and  $t$  on  $P$ , it can report the translation of  $\overline{ab}$  that minimizes its Fréchet distance to  $P[s, t]$  in  $O((n/k)^4 \log^8 n + \log^{16} n)$  time.*

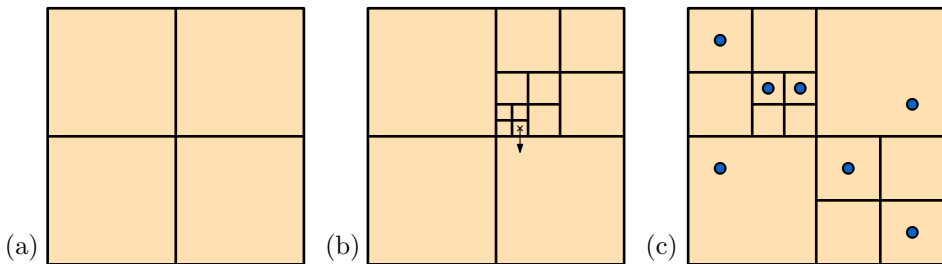
## 6.8 Concluding remarks

We presented data structures for efficiently computing the Fréchet distance of (part of) a curve to a query segment. Our results improve over previous work for horizontal segments and are the first for arbitrarily oriented segments. However, we are left with the challenge of reducing the space used for arbitrary orientations. Before we can efficiently achieve this, there are two main issues. The first issue is that even for a small interval of query orientations (e.g., one of the  $O(n^2)$  angular intervals defined by lines through a pair of points) it is difficult to limit the number of relevant backward pairs to  $o(n^2)$ . This is because even a small rotation of a query segment may make a backward pair suddenly relevant. The second issue is how to combine the backward pair distance values contributed by various subcurves. For (low algebraic degree) univariate functions, the upper envelope has near linear complexity. However, when the function is bivariate then the complexity is worst-case near quadratic. The combination of these issues makes it hard to improve over the somewhat straightforward  $O(n^{4+\epsilon})$  space bound that is presented in this chapter. There is some evidence to believe that the actual upper envelope of this bivariate function does not have a complexity that is quadratic in the number of involved functions or, at least, that this upper envelope can be smartly decomposed or maintained. Indeed, the set of backward pairs that generate the functions of this envelope seems to be well-behaved with respect to the query orientation. Consider the set of all pairs of vertices of the trajectory. Each pair becomes a backward pair as soon as the segment between the two vertices becomes horizontal with respect to the orientation of the query segment. Thus, let us sort all pairs of vertices in the trajectory by slope (possibly, one of the variables of our bivariate function). Then the pairs become backward pairs (and stop becoming backward pairs) in some predetermined order. Moreover, every pair is a backward pair for exactly  $180^\circ$ . It follows that if we were to plot which pairs were backwards with respect to the query slope, then we can extract an interval tree of unit-size intervals. Perhaps it is possible to leverage this property to further decompose the bivariate envelope into an efficient search structure.

## Chapter Seven

# Dynamic smooth compressed quadtrees

The quadtree is a hierarchical spacial subdivision data structure. The subdivision is created with the following scheme: starting with a single square, iteratively pick a square and subdivide it into four equal-size smaller squares, until a desired criterion is met (Figure 7.1). Quadtrees can be used as a data structure to store a set of geometric objects. Throughout this chapter, we simultaneously reference by  $C \in T$  nodes in the quadtree  $T$  and their corresponding cube. Leaf cubes are cubes that correspond to a leaf in the quadtree. A quadtree is *smooth* when every leaf cube  $C$  is comparable in size to every leaf cube  $C'$  that is geometrically adjacent to  $C$ . We show how to dynamically maintain a smooth quadtree with worst case constant update time.



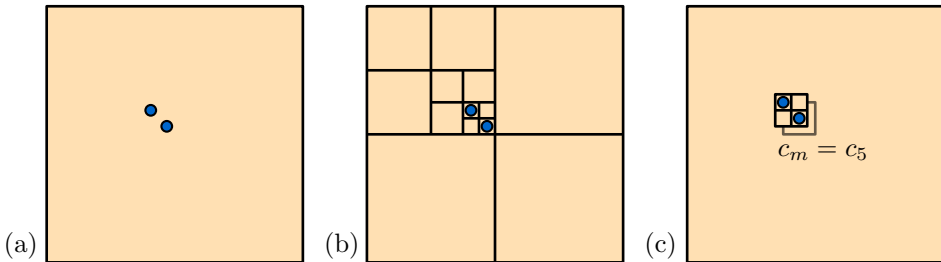
**Figure 7.1** (a) A single split partitions the root square into four squares. (b) After a number of iterative splits, a leaf marked with a cross is adjacent to a leaf with much larger size. (c) Given a point set  $P$  and a square that contains  $P$ , we can construct the minimal quadtree such that every point in  $P$  is in a unique leaf.

## 7.1 Introduction

Quadrees and their higher-dimensional equivalents have been long studied in computational geometry [22, 25, 40, 44, 80, 135, 151, 193]. Quadrees are popular among practitioners because of their ease of implementation and good performance in many practical applications [12, 21, 24, 64, 90, 152]. Given a planar point set  $P$  contained in a square, we can define the minimal quadtree that contains  $P$  to be the quadtree we obtain by recursively subdividing the root square until no leaf contains more than one (or a constant number of) point(s), refer to Figure 7.1(c). Given such a minimal quadtree  $T$ , we say  $T$  stores  $P$ . In a way, a quadtree captures the local information of every point as it can be converted to a Delaunay triangulation or Euclidean minimum spanning tree in linear time [147]. We do not review the rich literature here and instead refer to the great books by Samet [179] and Har-Peled [114].

**Smooth and dynamic quadrees.** A quadtree is *smooth* (also called *balanced* by some authors, which is not to be confused with the notion of balance in trees related to the relative weights of subtrees) if each leaf is comparable in size to adjacent leaves. It has been long recognized that smooth quadrees are useful in many applications [24], and smooth quadrees can be computed in linear time (and have linear complexity) from their non-smooth counterparts [61, Theorem 14.4]. A quadtree is *dynamic* if it supports making changes to the structure in sublinear time. Common structure changes considered are the *split* operation, which selects a leaf square and subdivides it into four equal-sized squares and the *merge* operation which is its inverse.

Recently, quadrees have been applied in kinetic and uncertain settings that call for dynamic behaviour of the decomposition [64, 129, 148]. Bennett and Yap [22] show how to maintain a smooth *and* dynamic quadtree subject to amortized constant-time *split* and *merge* operations on the quadtree leaves.



**Figure 7.2** (a) Two points in a root square. (b) The quadtree would need five splits to store these points. (c) Alternatively, we can compress the path from the root  $C_1$  to a square  $C_5$ . Computing the exact location of  $C_5$  naively requires four divisions which might be computationally expensive.

In this chapter, we distinguish between the quadtree *an sich*, a combinatorial subdivision of space containing  $n$  cubes, and the quadtree as a data structure for storing a point set  $P$  of  $n$  points (or some other set of geometric objects). A quadtree that stores a set of geometric objects has two well-known complications in the real RAM (Figure 7.2): First, such a minimal quadtree may have superlinear (even unbounded) complexity. Indeed, consider a real-valued point set where two points lie arbitrarily close within a unit square. It may be that the square must be split a near-infinite number of times before the two points each lie within a distinct cube. Such superlinear quadtrees can still be stored in linear space, using *compression* of each root-to-leaf path [61] (when compressing a quadtree, any path  $(C_1, C_2, \dots, C_m)$  in the tree, where all geometric objects contained in  $C_1$  are also contained in  $C_m$  can get replaced by a pointer from  $C_1$  to  $C_m$ ). Secondly, when constructing such a compressed quadtree it may not be possible to keep different compressed components properly aligned [114, 151] using reasonable computation time. Indeed, consider again our scenario of the two arbitrarily close points. If the starting square was unit size, then the size of each cube that stores a point may be  $2^{-L}$  for some value of  $L$  that is doubly exponential in  $n$ . Computing this exact value on a real RAM takes superlinear time. Hence compressed quadtrees in the real RAM compute an approximation of the disjoint squares which are not aligned with any other squares in the quadtree. These complications imply that we cannot simply apply results known for standard (*uncompressed*) quadtrees from the word RAM.

For any square  $C$  in the quadtree  $T$  its *neighbours* are the squares in  $T$  which have a facet that (partially) coincides with a facet of  $C$ . We say a quadtree is  $2^j$ -smooth if for every square  $C$  of  $T$  its neighbouring leaves have size at most  $2^j|C|$ . A quadtree is smooth if it is  $2^j$ -smooth for some constant  $j$ . The results in this chapter are twofold. First we show how to dynamically maintain a *standalone* smooth quadtree, subject to splits and merges of quadtree leaves. Second we show how to dynamically maintain a smooth, compressed quadtree that stores a point set  $P$ , subject to the high-level operations of inserting points into  $P$  and removing points from  $P$ . In Table 7.1 in Section 7.2 we provide a complete list of operations and how they relate.

**Contribution and organisation.** In the first half of this chapter, we study standalone smooth dynamic quadtrees. We improve a recent result by Bennett and Yap [22] as we can dynamically maintain a smooth quadtree in worst-case (as opposed to amortized) constant time per split and merge operation. We recall the higher-dimensional generalisation of quadtrees and we show that for  $d$ -dimensional quadtrees we can dynamically maintain their smooth variants with a worst-case update time that only depends on the dimension  $d$ . We study the number of combinatorial changes required to maintain a smooth quadtree. Therefore, we follow [22] and assume that every update we are given a pointer (*finger*) to a leaf subject to a split operation, or four leaves that were the result of a prior split operation that now need to be merged. We show how to perform both operations in worst-case constant time, whilst maintaining a smooth quadtree. The challenge here is to avoid cascading chains of updates required to maintain smoothness.

Our key idea is to introduce several layers of smoothness: we maintain a  $d$ -dimensional quadtree  $T$  (which consists of hypercubes, that we henceforth reference as cubes) and we require  $T$  to be 2-smooth. The split and merge updates are performed on  $T$  only. To achieve this, we add a second layer of cubes to the core quadtree that make sure that  $T$  is 2-smooth. These cubes themselves only need to be 4-smooth, and are made 4-smooth by a third layer of cubes that need to be 8-smooth and so forth.

Section 7.2 presents the necessary preliminaries needed to define and obtain our results. Here we also formally define the quadtree and its higher-dimensional variants. In Section 7.3, we show that when defining layers in this way, we need only  $(d + 1)$  layers for a quadtree over  $\mathbb{R}^d$ : the last layer will always already be  $2^{d+1}$ -smooth. In Section 7.4, we use the observations from Section 7.3 to dynamically maintain the core tree and its additional layers in constant time. The results in this first half are applicable to both the word RAM and real RAM. In Section 7.5 we show how to enhance the smooth quadtree so that it supports point location queries whilst we maintain worst-case constant update time.

The second half of the chapter, is focused on lifting our result to quadtrees that store a set of points. If the point set is in the word RAM, and every coordinate can be specified with constant word size, there can be no compression or alignment issues and the results of the first half immediately apply. If the point set is a real-valued point set in the real RAM, the quadtree can have superlinear complexity without compression (Section 7.6). Inserting a compressed component can be computationally infeasible if it has to be aligned with the root cube of the quadtree (Section 7.7). The challenge in these sections is to redefine compressed quadtrees in a consistent way across different layers of smoothness, and to re-align possibly misaligned components on the fly when such components merge. We tackle these challenges by restricting the point sets  $P$  that can be stored in a compressed quadtree.

**Implications.** In many applications of quadtrees, it is useful to be able to walk in constant time from any leaf of the tree to any of its neighbouring leaves using pointers (see a *threaded quadtree* in [135]). If the quadtree is not smooth then per definition, a single leaf may have many smaller neighbours. If we require that every leaf has a pointer to its neighbouring leaves, this prohibits constant-time updates of these pointers in a dynamic tree. In some applications of quadtrees, it is only required to maintain pointers from every leaf to every *larger* neighbouring leaf. However, even in this case it has been shown that *dynamically* maintaining such pointers in worst-case constant time is not possible unless the quadtree is smooth [22].

A large number of papers in the literature explicitly or implicitly rely on the ability to efficiently navigate a quadtree as if it were a threaded quadtree. Our results readily imply improved bounds from amortized to worst-case [64, 129, 148, 152, 172]. Several dynamic applications are in graphics-related fields and are trivially made parallel, which enhances the need for worst-case bounds.

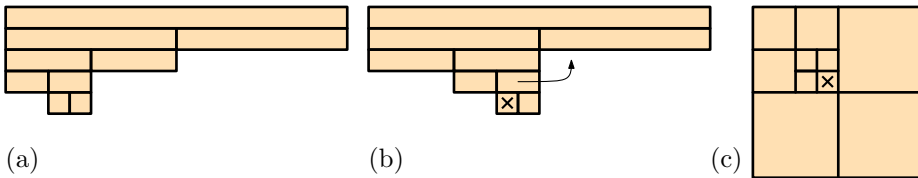
## 7.2 Preliminaries

In this section we review the necessary definitions for stating and proving our results. Consider the  $d$ -dimensional real space  $\mathbb{R}^d$ . For brevity, we refer to hypercubes in  $\mathbb{R}^d$  as cubes. Given a cube  $R \subset \mathbb{R}^d$  the *size* of  $R$ , denoted  $|R|$ , is the length of a 1-dimensional facet (i.e., an edge) of  $R$ . Given an axis-aligned cube  $R \subset \mathbb{R}^d$ , a *quadtree*  $T$  on  $R$  is a hierarchical decomposition of  $R$  into smaller axis-aligned hypercubes called *quadtree cubes*. We refer to  $T$  as both a decomposition of  $R$  and a tree, and to  $R$  as the root cube and the root of the tree  $T$ . Every tree node  $v$  of  $T$  has an associated cube  $C_v$ , and  $v$  is either a leaf or it has  $2^d$  equal-sized children whose cubes partition  $C_v$ . We follow [151] in using *quadtree* in any dimension rather than dimension-specific terms, such as *octree* for a three-dimensional quadtree. Henceforth, unless explicitly stated otherwise, when we refer to a quadtree cube  $C \in T$  we refer to both the cube  $C \subset \mathbb{R}^d$  and the quadtree node of the tree corresponding to  $C$ . Two cubes in  $T$  are neighbours whenever they are interior-disjoint and share (part of) a  $(d-1)$ -dimensional facet. They are siblings whenever they have the same parent cube in  $T$ , and sibling neighbours when both conditions hold (Figure 7.3(c)).

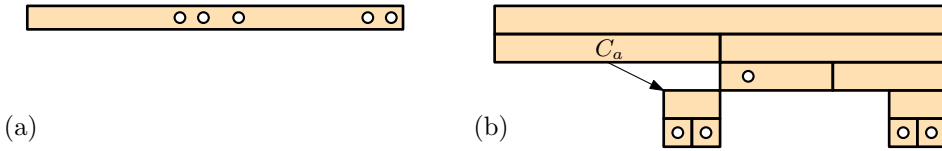
**Observation 7.1** *Let  $C$  be a quadtree cube. Then  $C$  has at most  $2d$  neighbours of size  $|C|$  and for each of the  $d$  dimensions,  $C$  has exactly one sibling neighbour that shares a  $(d-1)$ -dimensional facet in that dimension.*

For an integer  $j$ , we call a cube  $C$   $2^j$ -smooth if the size of each leaf neighbouring  $C$  is at most  $2^j|C|$ . If a cube is  $2^j$ -smooth, it is also  $2^k$ -smooth for  $k > j$ . If every cube in a quadtree is  $2^j$ -smooth, the quadtree is called  $2^j$ -smooth. We note that if all the quadtree leaves are  $2^j$ -smooth, then all the intermediate cubes are  $2^j$ -smooth as well (that is, the quadtree is  $2^j$ -smooth). Observe that if a *single* (leaf) cube is  $2^j$ -smooth, its ancestors do not necessarily have to be such (Figure 7.3).

Given a set  $P$  of points in  $\mathbb{R}^d$  and a cube  $R$  containing all points in  $P$ , an *uncompressed quadtree* that stores  $P$  is a quadtree  $T$  in  $\mathbb{R}^d$  on the root cube  $R$  that can be obtained by starting from  $R$  and iteratively subdividing every cube that contains at least two points in  $P$  into  $2^d$  child cubes.



**Figure 7.3** (a) We draw quadtrees in  $\mathbb{R}^1$  where intervals are rectangles and the tree depth is shown by the  $y$ -axis. (b) The marked cube is 2-smooth whilst its parent cube is not. (c) In  $\mathbb{R}^2$ , a leaf cube has two sibling neighbours but three siblings.



**Figure 7.4** (a) An interval containing a set  $P$  of five points. (b) In the 4-compressed quadtree that stores  $P$ , the cube  $C_a$  gets a compressed link.

**Defining  $\alpha$ -compressed quadtrees.** In the introduction, we described a scenario of a point set of two points  $P$ , such that the quadtree that stores  $P$  has a superlinear number of cubes. Let  $\alpha$  be a large fixed constant. An  $\alpha$ -compressed quadtree is a quadtree with in addition *compressed nodes*. A compressed node  $C_a$  does not have  $2^d$  children, but only one child  $C$ . Specifically, given a quadtree  $T$  the compressed quadtree replaces every maximal path  $(C_a, \dots, C)$  where (1) all cubes in the path contain at most 1 child that is not a leaf and (2)  $|C| \leq |C_a|/\alpha$ , with a compressed link ( $C_a$  has a compressed link to  $C$ ). We say that  $C_a$  is the *parent* of the compressed link (Figure 7.4). Given a point set  $P$  of  $n$  in  $\mathbb{R}^d$  where  $d$  is constant, its compressed quadtree  $T(P)$  has a linear number of cubes.

Compressed nodes induce a partition of a compressed quadtree  $T$  into a collection of uncompressed quadtrees interconnected by compressed links. We call the members of such a collection the *uncompressed components* of  $T$  and we denote the set of all uncompressed components by  $\mathcal{A}(T)$ . We refer to the largest cube in any uncompressed component as its root.

| Operation  | Running time               |
|--|----------------------------|
| <i>I. Quadtree operations (uncompressed quadtree)</i>  |                            |
| Splitting/ merging cube(s)   | $O((2d)^{d+2})$            |
| <i>II. Quadtree operations (<math>\alpha</math>-compressed quadtree)</i>                           |                            |
| Splitting / merging cube(s)  | $O((2d)^{d+2} \cdot 36^d)$ |
| Upgrowing of an uncompressed component   | $O((2d)^{d+3} \cdot 54^d)$ |
| Inserting / deleting an uncompressed component   | $O(d \cdot 36^d + 54^d)$   |
| <i>III. Operations on an <math>O(1)</math>-dim. point set <math>P</math>, stored in a quadtree</i> |                            |
| Insert a point into $P$  | $O(\log(n) + 1)$           |
| Insert a point into $P$ , given a finger   | $O(1)$                     |
| Delete a point from $P$  | $O(1)$                     |

**Table 7.1** A list of the operations considered in this chapter grouped into three categories with their corresponding running times in the provided implementation.

### 7.2.1 Quadtree operations and queries

Next, we specify what operations we support for our quadtree. It is insightful to distinguish three levels of operations:

- I Operations on an uncompressed quadtree, which are the split and merge operation.
- II Operations on a compressed quadtree (II), which internally require operations on an uncompressed quadtree (I).
- III Operations on a point set stored in a quadtree (III), which internally perform operations on a compressed quadtree (II and I).

Table 7.1 gives an overview of quadtree operations and the associated running times of the operations. We now give the formal definitions:

**Definition** (*split, merge*) Given a leaf cube  $C$  of a quadtree  $T$ , the *split* operation for  $C$  inserts the  $2^d$  equal-sized children of  $C$  into  $T$ . Given a set  $2^d$  leaves of a quadtree  $T$  whose parent is the same cube, the *merge* operation these  $2^d$  cubes.

**Definition** (*component insertion, component deletion*) Given a leaf cube  $C$  of a compressed quadtree  $T$ , a *component insertion* into  $C$  inserts a leaf  $A$  and a compressed link to  $C$ . Component deletion is its inverse, and can only be performed if the deleted component  $A$  is a single leaf.

**Definition** (*upgrowing, downgrowing*) Let  $A$  be an uncompressed component of a compressed quadtree. *Upgrowing* of  $A$  adds the parent  $R'$  of the root cube  $R$  of  $A$  ( $R'$  becomes the root of  $A$ ). *Downgrowing* of  $A$  removes the root  $R$  of  $A$ , and all the children of  $R$  except one. The remaining child becomes the root of  $A$ .

The downgrowing operation requires that the root  $R$  that gets removed contains at most one child cube that is not a leaf (that child cube is the only cube that can remain after the downgrowing operation). Whenever the tree  $T$  is a compressed quadtree, the merge operation requires that the  $2^d$  children contain at most one compressed link. The parent of these children becomes the new parent of the link.

**Definition** (*point location*) Given a point  $q \in \mathbb{R}^d$ , return the quadtree leaf of  $T$  that contains  $q$ .

An insertion of a point  $p$  into the set  $P$  stored in an  $\alpha$ -compressed quadtree  $T$  is performed in two phases: first, the unique leaf cube of  $T$  that contains  $p$  should be found. We do this in logarithmic time using our point location structure. Second, the quadtree should be updated by either performing a constant number of splits or by changing the leaf into a compressed node that stores the new leaf that contains  $p$ . We refer in Table 7.1 to the second phase separately as *inserting a point given a finger*. Deletion of a point is its inverse, where we identify the quadtree leaf that contains the point in constant time through a pointer. Given the pointer, we check the children of the parent of the leaf in  $O(2^d)$  time to see if they are all empty. If so, the parent can be merged and we recurse. We perform this check until no merges must be done.

### 7.3 Static smooth non-compressed quadrees

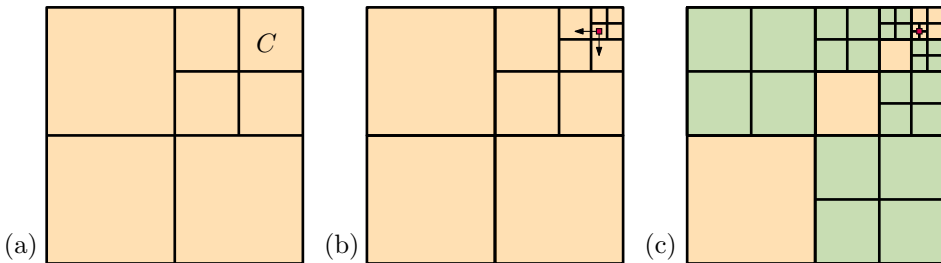
We first view the quadtree as a standalone static planar decomposition. In this section we are given a non-smooth, uncompressed quadtree  $T$  with  $n$  cubes. It is known [61, Theorem 14.4], that if  $T$  is an uncompressed quadtree containing  $n$  cubes over a constant dimension then  $T$  can be made 2-smooth by adding  $O(n)$  cubes.

However, if one wants to dynamically maintain a quadtree then naively adding (and possibly later removing) these  $O(n)$  cubes is costly. Specifically, Bennet and Yap [22] note that there is a specific order of split and merge operations, such that if at all times one wants to maintain a 2-smooth quadtree then there is a split that triggers  $O(n)$  additional split operations in order to keep the quadtree 2-smooth. Such an effect they call a *cascade* and we present an example of such a sequence of splits in Figure 7.5.

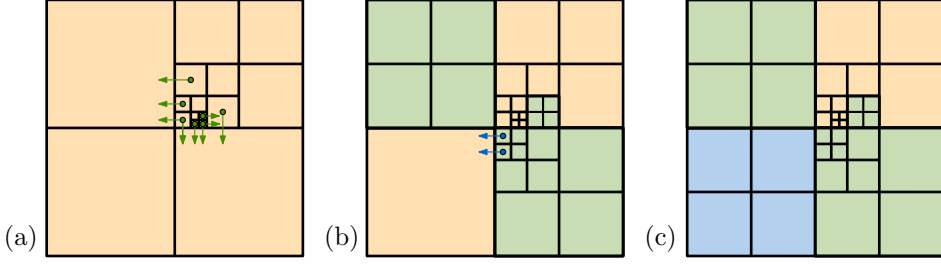
In this section we aim to avoid such costly behavior. We study a quadtree  $T$  over  $\mathbb{R}^d$  of  $n$  cubes. We refer to  $T$  as the *core quadtree* and we desire  $T$  to be 2-smooth. To ensure  $T$  is 2-smooth, we create what we call an *extended tree*  $f(T)$  by adding cubes to  $T$ . The quadtree  $T$  is a subset of the extended quadtree  $f(T)$ . Specifically given  $T$ , we define sets of cubes  $f_j(T)$  and quadrees  $f^j(T)$  as follows (Figure 7.6):

- $f_1(T) = T$ ,
- $f_{j+1}(T) =$  the minimal set of cubes (and their siblings) such that cubes in  $f^j(T)$  are 2-smooth,
- $f^j(T) = \bigcup_{i=1}^j f_i(T)$ , and
- $f(T) = f^{d+1}(T)$  where  $d$  is the quadtree dimension.

In this section, we prove the following properties of our extended tree  $f(T)$ : that it is unique, that if  $T$  contains  $n$  cubes, the tree  $f(T)$  contains at most  $O((2d)^{d+1}n)$  cubes, that  $f(T)$  is  $2^{d+1}$ -smooth and that it can be dynamically maintained in  $O((2d)^{d+2})$  time per split and merge in the core tree  $T$ .



**Figure 7.5** (a) A quadtree, where we iteratively split the top right cube. At all times, this quadtree is 2-smooth. (b) After  $O(n)$  iterations, we split a sibling cube instead. Its children are not 2-smooth. (c) Cubes that we add to maintain 2-smoothness are shown in green. If we split the neighbours of the marked cube to make them 2-smooth, the new cubes themselves are not 2-smooth. This cascades  $\Theta(n)$  times.



**Figure 7.6** Cubes in  $f_1(T)$  are yellow,  $f_2(T)$  is green and  $f_3(T)$  is blue. (a) A quadtree  $T = f_1(T)$  where cubes point to neighbouring leaves that make them not 2-smooth. (b) Some cubes in  $f_2(T)$  point to neighbouring leaves that make them not 4-smooth. (c) We denote  $f(T) = f^3(T) = f_1(T) \cup f_2(T) \cup f_3(T)$  and  $f(T)$  is 8-smooth.

**Lemma 7.1** For any set  $f^j(T)$ , the set  $f_{j+1}(T)$  is unique.

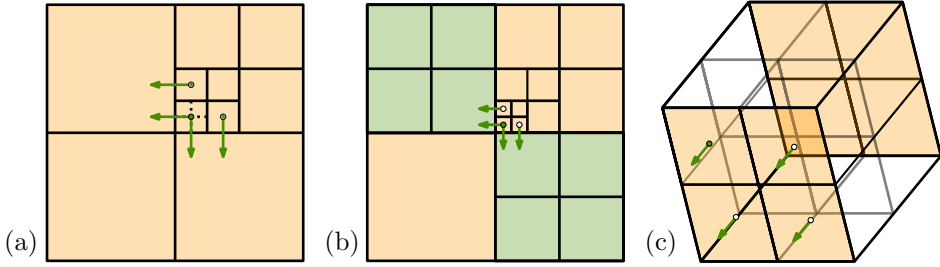
**Proof** A cube  $C$  is  $2^j$ -smooth if all its neighbouring leaf cubes are at most a factor  $2^j$  larger than  $C$ . This means that if we want ensure a cube  $C$  is  $2^j$ -smooth then we need to check for each of its neighbouring cubes if they are too large and if so, add a minimal number of cubes accordingly (thus, this set is unique). If for each cube  $C$ , the set of cubes that ensure it is  $2^j$ -smooth is unique, then the union (the set  $f_{j+1}(T)$ ) is unique. ■

**Lemma 7.2** If  $T$  is a set of  $n$  cubes, then for all  $j$ ,  $f_j(T)$  contains at most  $2^d(2d)^j n$  cubes.

**Proof** We prove this by induction. Per assumption  $f_1(T) = T$  has  $n$  cubes.

We continue with the induction step, for some integer  $j$ . Let  $f_j(T)$  be a set of at most  $2^d(2d)^j n$  cubes. We prove that  $f_{j+1}(T)$  is a set of at most  $2^d(2d)^{j+1} n$  cubes. Recall that the set  $f_{j+1}(T)$  is the minimal set of cubes that ensures all cubes in  $f^j(T) = \bigcup_{i=1}^j f_i(T)$  are  $2^j$ -smooth. Per definition, all cubes in  $f_i(T)$  with  $i < j$ , are  $2^i$ -smooth and thus already  $2^j$ -smooth. Therefore our analysis focuses on non-smooth cubes in  $f_j(T)$ .

We now count the number of splits needed in  $f^j(T)$  to ensure all cubes in  $f_j(T)$  are  $2^j$ -smooth. We start by setting the counter to 0 and we consider the largest cube  $C \in f_j(T)$  that is not  $2^j$ -smooth (if there are multiple such cubes, choose one). The cube  $C$  can have at most  $d$  leaf neighbours which are larger than  $C$ . We claim that for each such neighbour  $C'$ ,  $|C'| \leq 2^{j+1}|C|$ . Indeed, suppose for the sake of contradiction that  $|C'| > 2^{j+1}|C|$ . The parent  $C_a$  of  $C$  also neighbours  $C'$  and per assumption:  $|C'| > 2^{j+1}|C| = 2^j|C_a|$ . However, then  $C_a$  is not  $2^j$ -smooth which contradicts the assumption that  $C$  was the largest cube that is not  $2^j$ -smooth in  $f_j(T)$ . Hence, we need to split each of the at most  $d$  larger neighbours of  $C$  at most once to let  $C$  be  $2^j$ -smooth. We add  $d$  to our counter, and apply our argument recursively. Since adding cubes to  $f_{j+1}(T)$  cannot make cubes in  $f_j(T)$  less smooth, we split at most  $d \cdot |f_j(T)| = d \cdot 2^d(2d)^j n$  leaves (creating  $d \cdot 2^d \cdot 2^d(2d)^j n$  cubes which is too many).



**Figure 7.7** (a) The proof considers the largest cube in  $f_1(T)$  that is not 2-smooth. Choose a cube  $C$  marked by a green dot. (b) When we split the neighbours of  $C$  to create cubes in  $f_2(T)$ , the next largest cube that is not 2-smooth is a child cube of  $C$ . (c) In  $\mathbb{R}^3$ , if there is a cube  $C$  in  $f_1(T)$  that is not 2-smooth because of a neighbour  $C'$ , this cube  $C$  has four siblings which neighbour  $C'$ .

However, this argument is overcounting the number of leaves that need to be split. Indeed, consider a leaf  $C'$  in  $f^i(T)$  that needs to be split because of a cube  $C$ . The leaf  $C'$  neighbours  $2^{d-1}$  siblings of  $C$  that via this argument all call for  $C$  to be split (Figure 7.7). Hence, the prior analysis overcounted  $2^{d-1}$ -splits that needed to occur and the total number leaves in  $f^i(T)$  that need to be split is:

$$[\text{Number of leaves split in } f^j(T)] = \frac{d \cdot 2^d (2d)^j n}{2^{d-1}} = 2d \cdot (2d)^j n = (2d)^{j+1} n.$$

Each split creates  $2^d$  cubes. Thus,  $|f_{j+1}(T)| \leq 2^d (2d)^{j+1} n$ . ■

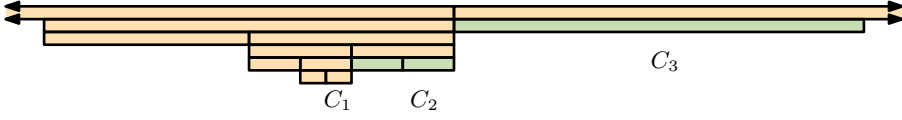
**Corollary 7.1** *If  $T$  is a quadtree over  $\mathbb{R}^d$  with  $n$  cubes then the set  $f(T) = f^{d+1}(T)$  contains at most  $O((2d)^{d+2} n)$  cubes.*

**Lemma 7.3** *Let  $T$  be some quadtree over  $\mathbb{R}^d$  and  $C$  be a cube in  $f_j(T)$ . For every neighbour  $N \in f(T)$  of  $C$  with  $2^j |C| \leq |N|$ , it must be that  $N \in f^{j+1}(T)$ .*

**Proof** Suppose for the sake of contradiction that a cube  $C$  has a neighbour  $N \in f(T)$  with  $N \geq 2^j |C|$  and  $N \notin f^{j+1}(T)$ . Then  $C$  is not  $2^j$ -smooth in  $f^{j+1}(T)$  which contradicts its definition. ■

### 7.3.1 Proving that $f(T)$ is always smooth over $\mathbb{R}^1$ and $\mathbb{R}^2$

With these lemmas in place, we are almost ready to prove the main results of this section: if  $T$  is a quadtree over  $\mathbb{R}^d$  then cubes in  $f_{d+1}(T)$  are already  $2^{d+1}$ -smooth. For ease of exposition, we first present our argument for  $\mathbb{R}^1$  and  $\mathbb{R}^2$  separately. We later generalise the argument for  $\mathbb{R}^2$  to  $\mathbb{R}^d$ . We denote for any cube  $C \in f(T)$ , by  $\text{PARENT}(C)$  its parent cube and by  $\text{GPARENT}(C)$  the parent of its parent.



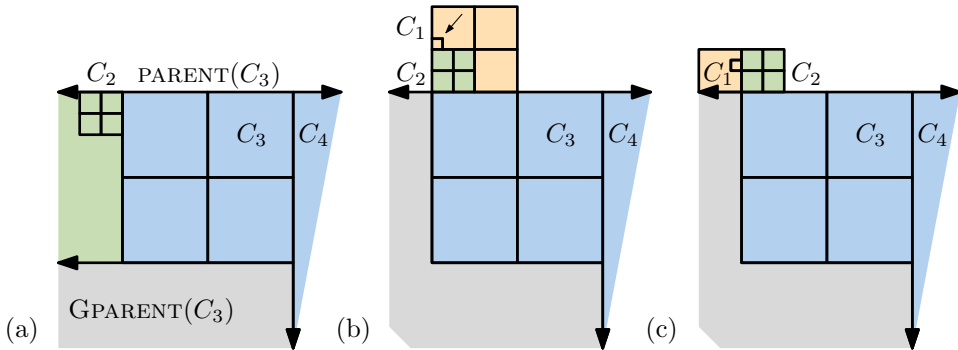
**Figure 7.8** The construction of the proof of Theorem 7.1. Green cubes are in  $f_2(T)$ . Two cubes  $C_2$  and  $C_3$  share their right facet, with  $C_3$  eight times as large as  $C_2$ . Observe that the cube  $C_1$  has to be contained in  $\text{GPARENT}(C_2)$ .

**Theorem 7.1** Let  $T$  be an uncompressed quadtree over  $\mathbb{R}^1$ . In  $f(T) = f^2(T)$  there cannot be two neighbouring leaf cubes  $C_2, C_3 \in f^2(T)$ , such that  $|C_2| < \frac{1}{4}|C_3|$ .

**Proof** Assume for the sake of contradiction that we have two neighbouring cubes  $C_2$  and  $C_3$  in  $f^2(T)$  with  $|C_3| \geq 8|C_2|$ . It follows from the definition of  $f^2(T)$ , that  $C_2 \in f_2(T)$ . The cube  $C_3$  may be in either  $f_1(T)$  or  $f_2(T)$ . We assume that  $C_2$  is left of  $C_3$ , the other argument is symmetric. We illustrate the setting in Figure 7.8.

Consider  $\text{PARENT}(C_2)$ . This cube must have its left facet strictly left of  $C_2$  (else  $\text{PARENT}(C_2)$  would partially overlap with  $C_3$ ). Via the same reasoning,  $\text{GPARENT}(C_2)$  must have its left facet strictly left of  $\text{PARENT}(C_2)$ . We denote by  $C_1$  the cube in  $f_1(T)$  such that  $C_2$  exists to ensure that  $C_1$  is 2-smooth and we reason about its location in the plane to conclude the argument.

The cube  $C_1$  must be a descendent of  $\text{GPARENT}(C_2)$  since  $\text{GPARENT}(C_2)$  shares the right facet with  $C_2$  and  $C_1$  is left of  $C_2$ . However, this implies that in order to create  $C_1$  in  $f_1(T)$ , the cube  $\text{GPARENT}(C_2)$  must be split in  $f_1(T)$ . Thus,  $\text{PARENT}(C_2) \in f_1(T)$ . However, per definition  $|\text{PARENT}(C_2)| = 2|C_2| = \frac{1}{4}|C_3|$ . This implies that  $\text{PARENT}(C_2) \in f_1(T)$  is not 2-smooth in  $f^2(T)$  which contradicts its definition. ■



**Figure 7.9** The construction in the proof of Theorem 7.2. Blue cubes are in  $f_3(T)$ , green cubes in  $f_2(T)$  and yellow cubes in  $f_1(T)$ .

Theorem 7.1 proves that if  $T$  is a quadtree over  $\mathbb{R}^1$  then all cubes in the tree  $f^2(T)$  are 4-smooth. If  $T$  is a quadtree over  $\mathbb{R}^2$  then the argument is more involved. We consider two cubes  $C_3, C_4 \in f^3(T)$  with  $|C_3| < \frac{1}{8}|C_4|$  ( $C_3 \in f_3(T)$ ). We reason about the “chain” of cubes  $C_1, C_2, C_3$  that causes  $C_3$  to exist and argue that  $C_2$  or  $C_1$  cannot be placed in the plane without creating a contradiction.

**Theorem 7.2** *Let  $T$  be an uncompressed quadtree over  $\mathbb{R}^2$ . In  $f(T) = f^3(T)$  there cannot be two neighbouring leaf cubes  $C_3, C_4 \in f^3(T)$ , such that  $|C_3| < \frac{1}{8}|C_4|$ .*

**Proof** The proof resembles the proof of Theorem 7.1. We suppose for the sake of contradiction that there are two neighbouring leaves  $C_3, C_4$  in  $f^3(T)$  with  $|C_4| \geq 16|C_3|$ . Note that it must be that  $C_3 \in f_3(T)$ . We assume that the right facet of  $C_3$  is contained in the left facet of  $C_4$ ; all other cases are symmetrical. The proof is illustrated by Figure 7.9.

Per definition, the cubes  $\text{PARENT}(C_3)$  and  $\text{GPARENT}(C_3)$  are both smaller than  $C_4$ . The left facet of  $\text{PARENT}(C_3)$  must be strictly left of  $C_3$ , and the left facet of  $\text{GPARENT}(C_3)$  must be strictly left of  $\text{PARENT}(C_3)$  to avoid overlapping  $C_4$ . Moreover the cube  $\text{PARENT}(C_3)$  must be contained in  $f_3(T)$  since it is at best 8-smooth (it borders  $C_4$ ). We denote by  $C_2$  the cube in  $f_2(T)$  that causes  $C_3$  to exist.

**Placing  $C_2$ .** We argue about the location of  $C_2$ . We first claim that  $C_2$  cannot be contained in  $\text{GPARENT}(C_3)$ . Indeed, suppose for the sake of contradiction that this is the case (Figure 7.9 (a)). Then  $\text{GPARENT}(C_3)$  must have been split in  $f^2(T)$  to create  $C_2$ , which contradicts our earlier observation that  $\text{PARENT}(C_3) \in f_3(T)$ .

We claim next that the bottom facet of  $C_2$  must be aligned with the top facet of  $\text{PARENT}(C_4)$  or vice versa. Indeed, suppose for the sake of contradiction that this is not the case: then the ancestor  $C_a$  of  $C_2$  of size  $2|C_3|$  borders  $C_4$  or a sibling of  $C_4$ . Per definition:

$$|C_a| = 2|C_3| = \frac{2}{16}|C_4| = \frac{1}{8}|C_4|.$$

However, this implies that  $C_a$  is not 8-smooth which contradicts  $C_a, C_2 \in f^2(T)$ . We assume the bottom facet of  $C_2$  is the aligned facet (the other continuation of this argument is symmetrical). We denote by  $C_1$  the cube in  $f_1(T)$  that causes  $C_2$  to exist.

**Placing  $C_1$ .** We now argue about the location of  $C_1$ . The cube  $C_1$  cannot lie below  $C_2$ , or else it overlaps with  $C_3$ . We first claim that  $C_1$  cannot lie in  $\text{GPARENT}(C_2)$  (Figure 7.9 (b)). Indeed, if  $C_1$  is in  $\text{GPARENT}(C_2)$ , then  $\text{GPARENT}(C_2)$  must be split in  $f^1(T)$  to create  $C_1$ . However, this implies that  $\text{PARENT}(C_2) \in f_1(T)$ . Per definition,  $\text{PARENT}(C_2)$  borders either  $C_3$  or a sibling of  $C_3$  and:

$$|\text{PARENT}(C_2)| = 2|C_2| = 2 \cdot \frac{1}{4}|C_3| = \frac{1}{2}|C_3|$$

Via Lemma 7.3, this implies that  $C_3 \in f^2(T)$  which is a contradiction.

Thus,  $C_1$  must lie left or right of  $C_2$  (Figure 7.9 (c)). We claim next that the right facet of  $C_2$  must be aligned with the left facet of  $\text{PARENT}(C_3)$  or vice versa. Indeed, suppose for the sake of contradiction that this is not the case. Then the ancestor  $C_a$  of  $C_1$  of size  $2|C_2|$  borders  $C_3$  or a sibling of  $C_3$ . Per definition:

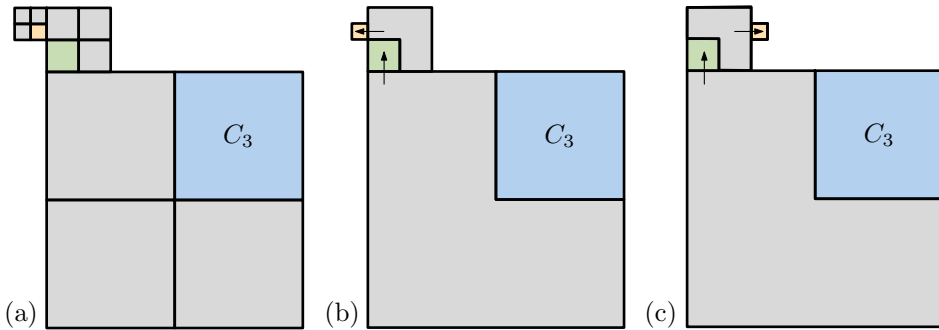
$$|C_a| = 2|C_2| = \frac{2}{4}|C_3| = \frac{1}{2}|C_3|.$$

However,  $C_a \in f^1(T)$  implies by Lemma 7.3 that  $C_3 \in f^2(T)$  which is a contradiction.

From these two claims, it follows that the ancestor  $C_a$  of  $C_1$  of size  $2|C_2|$  borders  $C_4$  or  $\text{GPARENT}(C_3)$ . Per definition,  $C_a$  must be 2-smooth. So either  $C_4$  or  $\text{GPARENT}(C_3)$  must be split. The first case is a contradiction with the assumption that  $C_4$  is a leaf. The second implies that  $\text{PARENT}(C_3) \in f^2(T)$  and since this cube per definition neighbours  $C_4$ ,  $C_4$  should again be split. ■

### 7.3.2 Proving that $f(T)$ is always smooth over $\mathbb{R}^d$

What remains to show is that if  $T$  is a quadtree over  $\mathbb{R}^d$ , then all cubes in the tree  $f^{d+1}(T)$  are  $2^{d+1}$ -smooth. We generalise the proof of Theorem 7.2. Specifically, we prove that there cannot be two cubes  $C_{d+1}, C_{d+2} \in f^{d+1}(T)$  with  $|C_{d+1}| < \frac{1}{2^{d+1}}|C_{d+2}|$ . Note that if this were to be the case, then per definition  $C_d \in f_{d+1}(T)$ . When assuming that such a pair  $C_{d+1}, C_{d+2}$  does exist, we create what we call a *chain* of cubes  $C_1, C_2, C_3, \dots, C_{d+1}$  where  $C_j$  is a cube in  $f_j(T)$  that requires  $C_{j+1}$  to exist in  $f_{j+1}(T)$  (Figure 7.10). More precisely: for every  $j$ :  $2^j|C_j| = |C_{j+1}|$  and  $C_j$  neighbours  $\text{PARENT}(C_{j+1})$ . We intuitively define that  $C_j$  *expands* the chain in a cardinal direction  $\pm \vec{e}$  if  $C_{j-1}$  lies in that direction from  $\text{PARENT}(C_j)$  (refer to Figure 7.10(b)).



**Figure 7.10** (a) A chain  $C_1, C_2, C_3$  with  $C_2$  in green and  $C_1$  in yellow. (b) The cube  $C_3$  expands the chain in the positive  $y$  direction. The cube  $C_2$  expands the chain in the negative  $x$  direction. (c) Here,  $C_2$  expands in the positive  $x$  direction.

Given this construction, we reason about where each cube  $C_j$  can be placed in  $\mathbb{R}^d$ . Recall the proof of Theorem 7.2. Whenever we placed  $C_2$  or  $C_1$ , we made two claims about its placement. The two following lemmas generalise these two claims. They are illustrated by Figure 7.11 (a) and (b) respectively.

**Lemma 7.4 (The first claim)** *Any cube  $C_{j-1}$  cannot be contained in  $\text{GPARENT}(C_j)$ .*

**Proof** Suppose for the sake of contradiction that  $C_{j-1}$  is contained in  $\text{GPARENT}(C_j)$ . Then to create  $C_{j-1}$ ,  $\text{GPARENT}(C_j)$  must be split in  $f^{j-1}(T)$  and thus  $\text{PARENT}(C_j) \in f^{j-1}(T)$ . However, per construction  $\text{PARENT}(C_j)$  neighbours either  $C_{j+1}$  or one of its siblings and:

$$|\text{PARENT}(C_j)| = 2|C_j| = \frac{1}{2^{j-1}}|C_{j+1}|.$$

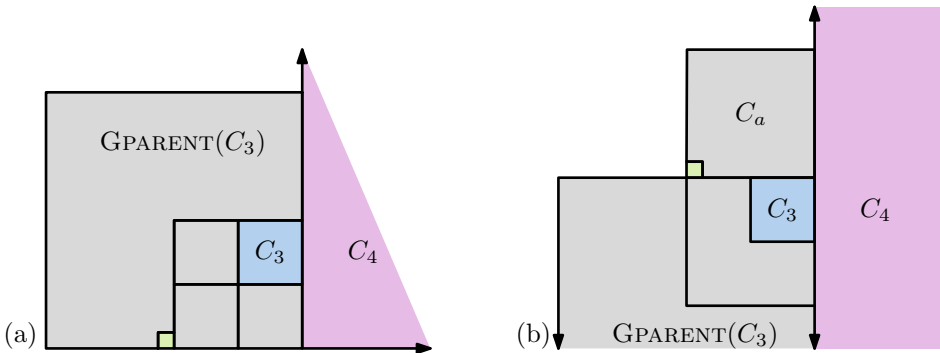
Lemma 7.3, this implies that  $C_{j+1} \in f^j(T)$ , contradiction. ■

**Lemma 7.5 (The second claim)** *For any  $j$ , if  $C_j$  expands in the positive  $y$ -direction, the bottom facet of  $C_{j-1}$  must be aligned with the top facet of  $C_{j+1}$ . A symmetrical property holds for any other cardinal direction.*

**Proof** Suppose for the sake of contradiction that this is not the case. We denote by  $C_a$  the ancestor of  $C_{j-1}$  of size  $2|C_j|$ . Since  $C_a \leq |C_{j+1}|$ ,  $C_a$  either borders  $C_{j+1}$  or a sibling of  $C_{j+1}$ . However  $C_a \in f^{j-1}(T)$  and per construction:

$$|C_a| = 2|C_j| = \frac{2}{2^j}|C_{j+1}| = \frac{1}{2^{j-1}}|C_{j+1}|$$

By Lemma 7.3, this implies that  $C_{j+1} \in f^j(T)$ , contradiction. ■



**Figure 7.11** (a) The scenario from the proof of Lemma 7.4 where  $C_2$  is in green and contained in  $\text{GPARENT}(C_3)$ . (b) The scenario from the proof of Lemma 7.5. The cube  $C_2$  is shown in green and is not contained in  $\text{GPARENT}(C_3)$ .  $C_a$  must neighbour  $C_4$ .

Finally, we are ready to prove the main theorem of this section:

**Theorem 7.3** *Let  $T$  be an uncompressed quadtree over  $\mathbb{R}^d$ . In  $f(T) = f^{d+1}(T)$  there cannot be two neighbouring leaf cubes  $C_d, C_{d+1} \in f(T)$ , such that  $|C_d| < \frac{1}{2^{d+2}}|C_{d+1}|$ .*

**Proof** Consider for the sake of contradiction the chain  $C_{d+1}, C_d, \dots, C_1$ . Let  $i, j$  with  $i \neq j$  be the two highest two indices, where  $C_j$  and  $C_{i+1}$  expand in a cardinal direction that coincides with the same axis. Since there are at most  $d$  dimensions and  $(d+1)$  expansions, the pigeonhole principle ensures that there exists at least one pair  $i, j$ , for which these conditions hold.

Observe that  $i < j - 2$ , since if  $i = j - 1$ ,  $C_i$  must be contained in  $\text{GPARENT}(C_i)$  which contradicts Lemma 7.4. We denote by  $C_a$  the ancestor of  $C_i$  of size  $2|C_{i+1}|$ . By Lemma 7.4 and 7.5,  $C_a$  either borders  $\text{GPARENT}(C_j)$ ,  $C_{j+1}$  or a sibling of  $C_{j+1}$ .

Suppose  $C_a$  borders  $C_{j+1}$  or one of its siblings, then:

$$|C_{j+1}| = 2^j |C_j| = 2^j \cdot 2^{j-1} |C_{j-1}| = 2^j \cdot 2^{j-1} \cdot \dots \cdot 2^{i+1} |C_{i+1}| > 2^i \cdot |C_a|.$$

The last inequality follows from the fact that  $i < j - 2$ . Now  $C_a \in f^i(T)$  combined with Lemma 7.3, implies that  $C_{j+1} \in f^{i+1}(T)$ , contradiction.

Suppose otherwise that  $C_a$  borders  $\text{GPARENT}(C_j)$ , then:

$$|\text{GPARENT}(C_j)| = 4|C_j| = 4 \cdot 2^{j-1} |C_{j-1}| = 4 \cdot 2^{j-1} 2^{j-2} |C_j| \geq 4 \cdot 2^{j-1} 2^i |C_i| \geq 2^{i+1} |C_a|$$

Now  $C_a \in f^i(T)$  combined with Lemma 7.3, implies that  $\text{GPARENT}(C_j) \in f^{i+1}(T)$ . Per construction of the chain of cubes  $\text{GPARENT}(C_j)$  borders either  $C_{j+1}$  or one of its siblings. Moreover, we have that:

$$|C_{j+1}| = 2^{j+1} |C_j| = 2^{j-1} |\text{GPARENT}(C_j)| \geq 2^{i+1} |\text{GPARENT}(C_j)|.$$

Lemma 7.3 then implies that  $C_{j+1} \in f^{i+2}(T)$  which is a contradiction. ■

**Corollary 7.2** *Let  $T$  be an uncompressed quadtree over  $\mathbb{R}^d$ , then  $f(T)$  is  $2^{d+1}$ -smooth.*

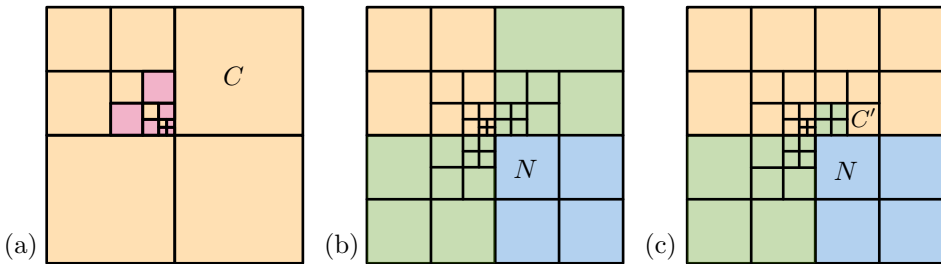
The above corollary implies that whenever  $T$  is an uncompressed quadtree over constant dimension then the extended quadtree  $f(T)$  is smooth. We continue by showing how to dynamically maintain  $f(T)$ .

## 7.4 Dynamic smooth non-compressed quadtrees

In the previous section we have shown that, given a static uncompressed quadtree  $T$  with  $n$  cubes over  $\mathbb{R}^d$ , there exists an extended quadtree  $f(T)$  which is  $2^{d+1}$ -smooth and contains at most  $2^d(2d)^{d+1}n$  cubes. In this section we show how to dynamically maintain  $f(T)$  subject to splitting and merging leaves in  $T$ .

**Naively splitting cells to maintain smoothness does not suffice.** The naive approach to try to maintain  $f(T)$  (subject to splitting cubes) is the following: whenever the split operation is invoked in  $f_1(T)$ , we check if our four new cubes are 2-smooth. If they are not, we split their too large neighbours to create cubes in  $f_2(T)$ . We check if the new cubes are 4-smooth, and recurse if needed. However, this naive strategy does not maintain the tree  $f(T)$ : it does not safeguard that for each  $j$ , the set  $f_{j+1}(T)$  is the minimal set that makes sure that cubes in  $f^j(T)$  are  $2^j$ -smooth. Indeed, consider the example in Figure 7.12 (a) where we start with a tree  $T = f_1(T)$ . Figure 7.12 (b) illustrates the associated tree  $f(T)$ . Suppose we now split the cube denoted by  $C$ , and then its bottom left child. It so happens, that these new cubes are already 2-smooth, due to a cube  $N \in f_3(C)$  (which has to be 8-smooth). Thus, the naive splitting algorithm terminates: as all new cells have the correct degree of smoothness. However, by definition of  $f(T)$ , the cube  $N$  should be a member of  $f_2(T)$  not  $f_3(T)$  (and should thus be required to be 4-smooth instead of 8 smooth). This illustrates that naively splitting cubes to maintain the smoothness of the newly added cubes is not enough: it is required to relocate cubes from  $f_j(T)$  to  $f_i(T)$ .

**Algorithmic sketch of maintaining  $f(T)$ .** We dynamically maintain  $f(T)$  as follows. Each cube  $C \in f(T)$  shares a heap of integers with its siblings which we denote by  $\text{BRANDS}(C)$ . The minimum integer in this heap indicates the level of the quadtree that  $C$  is in. We maintain the following invariant:



**Figure 7.12** (a) A set  $f_1(T)$  where red cubes are not 2-smooth. (b) We show  $f_2(T)$  in green and  $f_3(T)$  in blue. (c) Suppose we split  $C \in f_1(T)$ , and then its bottom left child. Then we create a cube  $C'$  adjacent to  $N \in f_3(T)$ .

- Invariant 7.1** (a) Each cube  $C \in f_j(T)$  if and only if  $j = \min \text{BRANDS}(C)$ .  
 (b) Each cube  $C \in f_j(T)$  has a brand pointer to each neighbour of size  $2^j|C|$ .  
 (c) Each cube  $C \in f(T)$  has a leaf pointer to each neighbouring leaf larger than  $C$ .

Given this addition to our quadtree we use three high-level operations. The SPLIT and MERGE operation can be invoked on any cube in  $T$  (where we define MERGE as a special case of the following REMOVEBRAND operation). Either of these two operations will trigger a chain of operations that are invoked on cubes in  $f_j(T)$  for all  $j \leq d + 1$ . Whenever these operations are invoked on a cube  $C \in f_j(T)$ , they maintain  $2^j$ -smoothness, maintain the brand and leaf pointers from and to  $C$ , and they insure that all cubes neighbouring  $C$  of size  $2^j|C|$  are in  $f_{j+1}(T)$  (thus, maintaining Invariant 7.1 for all cubes in the local area around  $C$ ).

A split on a cube  $C$  in  $T = f_1(T)$  does two things: First, we split cubes in  $f_1(T)$  to create cubes in  $f_2(T)$  to ensure that  $C$  is 2-smooth and second, we add the integer 1 to the set  $\text{BRANDS}(C')$  of neighbouring cubes  $C'$  to notify that these cubes are in  $f_2(T)$ . We additionally add the required pointers to maintain Invariant 7.1. Then, the newly split cubes and the cubes that were relocated to  $f_2(T)$  are checked to see if they are 4-smooth. This may invoke more splits and relocations of cubes and so forth.

The REMOVEBRAND operation is intuitively the inverse of the SPLIT and ADDBRAND operations. It removes an integer from the set  $\text{BRANDS}(C)$  of a cube  $C$ . If this changes the minimal value in  $\text{BRANDS}(C)$ , then the cube  $C$  must be relocated from  $f_i(T)$  to  $f_j(T)$ . If the set  $\text{BRANDS}(C)$  is empty, there is no need for the cube and its siblings to exist and we merge them. Both operations require pointer adjustments and possibly further invocations to maintain  $f(T)$  and Invariant 7.1.

- **SPLIT( $C, j$ ):** select a leaf  $C \in f(T)$  and split it, adding  $2^d$  cubes to  $f_j(T)$ , with for each added cell  $C_r$ ,  $\text{BRANDS}(C_r) = \{j\}$ . Then, do the following:
  - First, set the leaf and brand pointers outgoing from  $C$ . There are at most  $d$  leaves neighbouring  $C$  that are greater or equal to  $|C|$ . Therefore, explicitly adding these pointers from the children of  $C$  takes  $O(d \cdot 2^d)$  time. Note that there can be no  $C' \in f_i(T)$  with a brand pointer pointing to a child of  $C$  or else that child would not have been  $2^i$ -smooth.
  - Next, adjust incoming leaf pointers: since the quadtree is  $O(2^d)$ -smooth, there can be at most  $O(2^d)$  leaves with a leaf pointer to  $C$ . Each of these leaves has at most  $d$  ancestors smaller than  $C$ , shared with  $2^{d-1}$  siblings that all neighbour  $C$ . It follows that there are at most  $O(d \cdot 2^d)$  cubes in  $f(T)$  with a leaf pointer to  $C$ . Redirect every pointer in constant time each. The above two steps maintain Invariant 7.1(b) and (c).
  - For smoothness and Invariant 7.1(a), check for each cardinal direction if there is a leaf  $C'$  of size  $2^j|C|$  that shares a facet with  $C$  in that direction.
    - \* If so, invoke SPLIT( $C', j + 1$ ) and add a brand pointer to the new cells.
    - \* Otherwise, denote by  $C''$  the cube of size  $2^{j-1}|C|$  that shares a facet with  $C$  in that cardinal direction and invoke ADDBRAND( $C'', j + 1$ ).

- **ADDBRAND**( $C, j$ ): select a cube  $C$  and add the integer  $j$  to **BRANDS**( $C$ ). If the minimum of **BRANDS**( $C$ ) has changed from  $i$  to  $j$  then do the following:
  - For neighbouring cubes  $C'$  of size  $2^i|C|$  invoke **REMOVEBRAND**( $C', i$ ).
  - Remove outgoing brand pointers from  $C$ . And in  $O(d \cdot 2^d)$  time, add new brand pointers to all cubes in  $f(T)$  of size  $2^j|C|$  in  $O(d \cdot 2^d)$  time.
  - For neighbouring leaves  $C'$  of size  $2^{j+1}|C|$ , invoke **SPLIT**( $C', j + 1$ ).
  - For neighbouring cubes  $C'$  of size  $2^j|C|$ , invoke **ADDBRAND**( $C', j + 1$ ).
- **REMOVEBRAND**( $C, j$ ): this is the inverse of both **SPLIT** and **ADDBRAND**. Merging a cube  $C \in T$  with its siblings, is defined as **REMOVEBRAND**( $C, 1$ ). This operation removes a specific copy of the integer  $j$  from the heap associated with  $C$  and its siblings. If  $j = 1$ , this integer is the minimum of the heap. Else we make sure we access the copy through its associated brand pointer. Check if the heap is now empty. If so:
  - Remove  $C$  and its siblings in  $O(2^d)$  time.
  - Redirect the leaf pointer address of the  $2^d$  children of  $C$  to  $C$  in  $O(2^d)$  time.
  - For neighbours  $C'$  of  $C$  of size  $2^{j-1}|C|$ , invoke **REMOVEBRAND**( $C', j + 1$ ).

Else, check if removing this integer changes the minimum of the heap. Suppose the minimum changes to some integer  $i > j$ :

- Remove all brand pointers from  $C$  and spend  $O(d \cdot 2^d)$  time to add new brand pointers to neighbouring cubes of size  $2^i|C|$ .
- For neighbours  $C'$  of size  $2^i|C|$ , invoke **ADDBRAND**( $C', i + 1$ ).
- For neighbours  $C'$  of **PARENT**( $C$ ) of size  $2^j|C|$ , invoke **REMOVEBRAND**( $C', j + 1$ ).

**Theorem 7.4** *The above procedure dynamically maintains for every uncompressed quadtree  $T$  over  $\mathbb{R}^d$ , the extended tree  $f(T)$  in  $O((2d)^{d+2})$  time per split and merge in  $T$ .*

**Proof** Every operation invoked with an index  $j$  spends  $O(d \cdot 2^d)$  time, and invokes at most  $2d$  calls to an operation with an index greater than  $j$ . It immediately follows from the fact that  $f(T) = f^{d+1}(T)$ , that a single split or merge operation in  $T$  invokes at most  $(2d)^d$  additional operations, and that the total runtime is at most:

$$O(2^d(2d)^{d+1}) = O((2d)^{d+2}).$$

Each operation explicitly maintains Invariant 7.1. We claim that we also maintain  $f(T)$  via induction on  $j$ . The tree  $T = f_1(T)$  is correctly maintained per definition. Given that  $f_j(T)$  is correctly maintained, every cube  $C \in f_j(T)$  has a brand pointer to every neighbouring cube to every cube  $C'$  of size  $2^j|C|$  and through it the cube  $C$  ensures that the integer  $(j + 1)$  is in the set **BRANDS**( $C'$ ). Through that same mechanic, a cube  $C'$  has the integer  $(j + 1)$  in the set **BRANDS**( $C'$ ) only if there is such a cube  $C$  neighbouring it. It follows that we maintain  $f_{j+1}(T)$  and, by induction,  $f(T)$ . ■

## 7.5 Point location in a quadtree

In the previous sections we considered the quadtree as an abstract structure subject to the *split* and *merge* operations and we showed how to maintain a smooth quadtree in  $O((2d)^{d+2})$  time per split and merge. This update time is constant if the quadtree dimension  $d$  is constant.

Next, we show how to augment any quadtree  $T$  such that it is possible to do planar point location in  $T$  in time logarithmic in the number of leaves of  $T$  (if  $T$  is a quadtree of constant dimension). In point location, we are given a query point  $q$  and need to return the quadtree leaf of  $T$  that contains  $q$ . We want to be able to maintain a point location data structure on  $T$  which can be maintained in constant time if the dimension  $d$  is constant. At this point it is important to make a distinction between word RAM and real RAM operations. In a word RAM, it is possible to do point location in a quadtree at all times using a hash table, even in time sublogarithmic in the number of leaves of  $T$  (see *transdichotomous* operations in [186]). In the real RAM however, we are restricted to comparison-based operations. We show how to dynamically maintain a search tree on any constant-dimensional quadtree  $T$  in a real RAM with constant update time per split or merge in  $T$ .

To this end, we first define a linear order on the leaves of any quadtree  $T$ . We do this by associating to each cube  $C$  in the quadtree a real number  $n_C$ , such that the numbers associated to the leaves of the quadtree form a total order. Specifically, let  $T$  be a  $d$ -dimensional quadtree. We assign to the root of the tree the number 0. We then recursively assign a number as follows: let  $C'$  be the  $i$ -th child of a cube  $C$  (counting from 0, with some fixed order children of a cube) at depth  $\delta$  in the tree, its number is:

$$n_{C'} := n_C + \frac{i}{(2^d)^\delta}.$$

Figure 7.13 illustrates our number assignment. Observe that two quadtree cubes can have the same associated number if and only if one cube is contained in the other. Hence the set of associated numbers induces a linear order on the leaves of any quadtree. In addition in the real RAM (if every cube at depth  $\delta$  stores the number  $(2^d)^\delta$ ) every cube can compute the number for every child in constant time per child.

**Lemma 7.6** *Let  $T$  be a quadtree in some constant dimension,  $C \in T$  be a quadtree leaf,  $n_C$  its associated number and  $p$  a point in the plane. In  $O(2^d)$  time we can verify if  $p \in C$  or if  $p$  is contained in a quadtree leaf with a number greater or equal to  $n_C$ .*

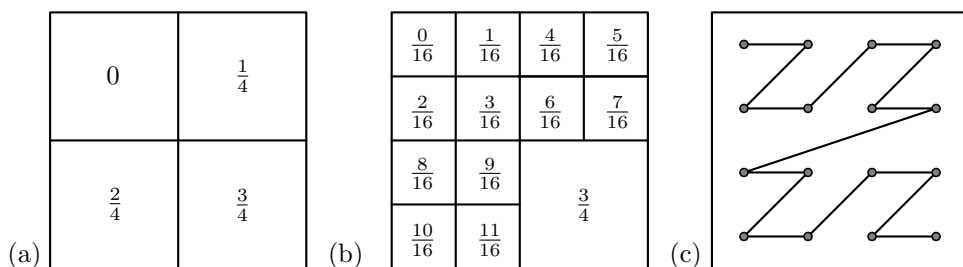
**Proof** Consider a leaf cube  $C \in T$ . It is an object with  $O(d)$  facets and we can hence check if  $p \in C$  in  $O(d)$  time. If  $p$  is not in  $C$  we do the following: we denote by  $A$  the smallest  $d$ -dimensional cube, rooted at a vertex of  $C$ , that contains both  $p$  and  $C$  (the cube  $A$  can be computed in  $O(d)$  time). If  $|A| \leq 2|C|$  we replace  $A$  with a cube of size  $2|C|$  rooted at one of the vertices of  $C$ .

We now split  $A$  into  $2^d$  children and we check which of its children contains  $C$  and which contains  $p$ . Since  $p \notin C$  and  $A$  is at least twice as large as  $C$ , the resulting two children must be two distinct children of  $A$ . We compare the two children in their fixed ordering as children of  $A$ . If the child containing  $p$  precedes the child containing  $C$ , the leaf containing  $p$  has an associated number which is lower than  $n_C$ . Otherwise it has an associated number which is greater than  $n_C$ . ■

**Theorem 7.5** *Let  $T$  be a quadtree in constant dimension on a real RAM. We can maintain a point location structure on  $T$  that supports point location queries in time logarithmic in the number of leaves of  $T$  and with constant update time per split and merge in  $T$ .*

**Proof** Fleischer [93] shows how to dynamically maintain a search tree on a set of ordered numbers on a comparison-based pointer machine. The search tree has logarithmic query time and can be updated in constant time per finger insertion or deletion. Here, a finger insert consists of a new value  $r$  to insert into the ordered set of numbers together with a pointer to two adjacent values  $a, b$  in the ordered set such that  $a \leq r \leq b$ .

We simply enhance  $T$  by storing the data structure from Fleischer on the set of quadtree leaves. By Lemma 7.6, if  $T$  is a quadtree in constant dimension we can compare any point to the numbers associated to the quadtree leaves in constant time and thus the data structure by Fleischer supports point location queries in time logarithmic in the number of leaves in  $T$ . Per assumption every split operation has access to a *finger* to the leaf  $C$  that is split. Every split operation removes one value from the sorted set of numbers and inserts  $2^d$  new values. Given this finger, the structure by Fleischer can be updated in constant time if  $d$  is constant. ■



**Figure 7.13** (a) Four quadtree leaves and their associated number. (b) If we further refine the quadtree, we also further refine the set of associated numbers. (c) The quadtree leaves ordered along their associated numbers resemble some space-filling Hilbert curve [54].

## 7.6 Quadtree compression

In previous sections, we showed how to dynamically maintain a  $2^{d+1}$ -smooth quadtree over  $\mathbb{R}^d$  subject to split and merge operations in the ‘core’ quadtree  $T$ . In this section and the next, we show how to use this abstract quadtree to dynamically store  $P$  subject to point insertions and deletions in  $P$ . The first challenge that presents itself is that if  $P$  is a point set in constant dimension then we want the associated quadtree to have size linear in the number of points in  $P$ . To this end, we need compressed quadtrees and a notion for smoothness for compressed quadtrees.

In this section we study the abstract concept of an  $\alpha$ -compressed quadtree. Let  $R$  be some bounding cube in  $\mathbb{R}^d$ . An  $\alpha$ -compressed quadtree on  $R$  is a tree where every node/cube  $C$  in the tree represents a cube in  $\mathbb{R}^d$  and either:

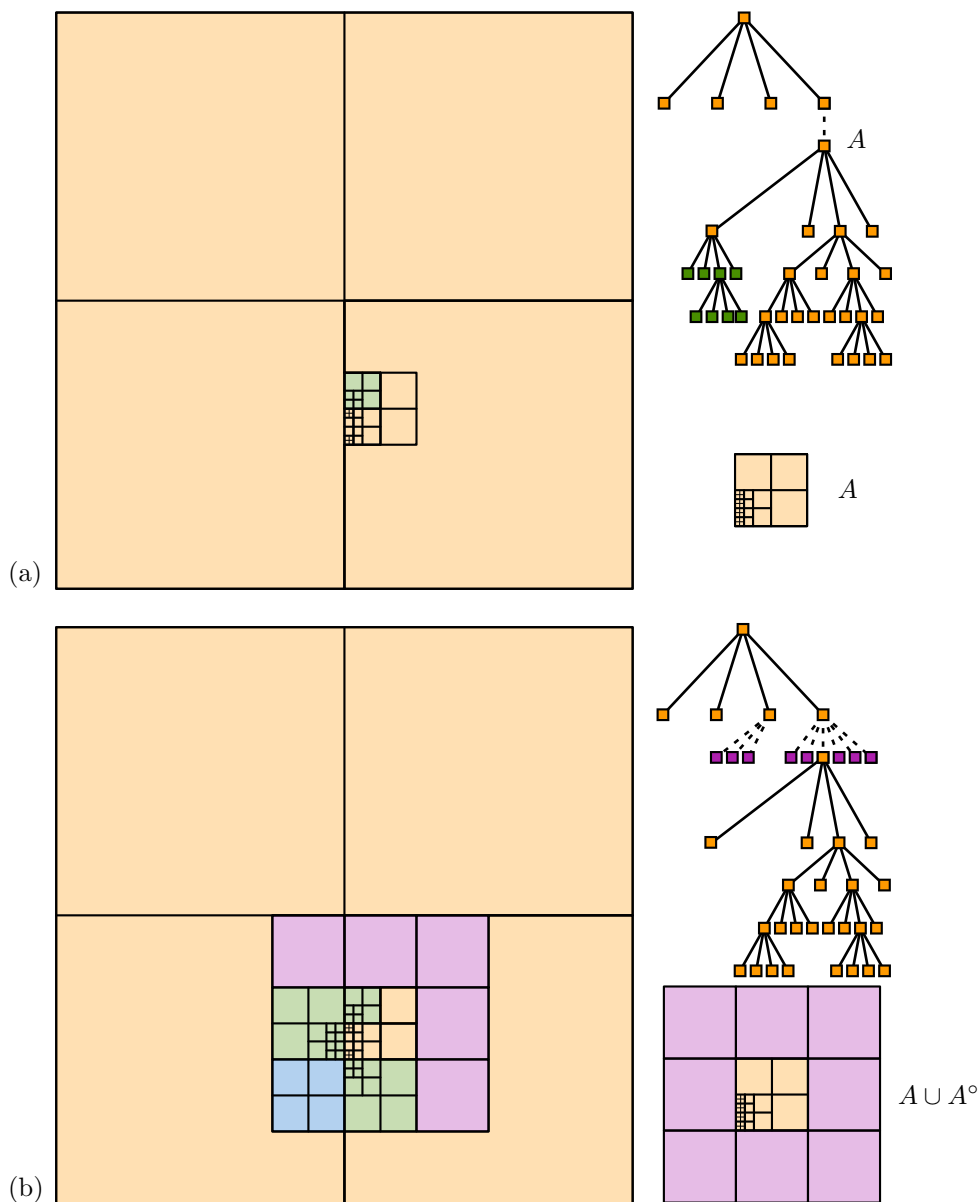
- the cube  $C$  is a leaf,
- the cube  $C$  has  $2^d$  children that partition  $C$ , or
- the cube  $C$  has a compressed link to a cube  $C'$  with  $|C|/|C'| > \alpha$ .

We denote by  $T$  the compressed quadtree and by  $\mathcal{A}(T)$  the set of its uncompressed components (the minimal partition of  $T$ , such that for every part  $A \in \mathcal{A}(T)$  there are no two cubes  $C, C' \in A$  where  $C$  has a compressed link to  $C'$ ). We refer for every uncompressed component  $A \in \mathcal{A}(T)$  to the *standalone quadtree*  $A$  as the quadtree obtained by transforming every cube  $C \in A$  that has a compressed link to a component  $B \in \mathcal{A}(T)$ , into a leaf. We study the compressed quadtree  $T$  subject to the split and merge operation on every *standalone* tree  $A \in \mathcal{A}(T)$ , insertions of a new compressed component  $C'$  in a leaf  $C$ , deletions of a compressed component  $C'$ , and the upgrowing and downgrowing operation (refer to Section 7.2.1 for the formal definition of each operation). In this section we make a critical assumption: for any value of  $\delta$  and leaf  $C$ , we assume that we can compute an arbitrary descendent of  $C$  denoted by  $C'$  with  $|C'| = 2^\delta$  (and the number  $n_{C'}$ ) in constant time. In the next section we elaborate on this assumption.

**Defining smooth quadtrees.** Let  $T$  be an  $\alpha$ -compressed quadtree for some large, fixed constant  $\alpha$  and  $\mathcal{A}(T)$  be its set of uncompressed components. Henceforth, we refer to  $T$  as a compressed quadtree (where the fixed constant  $\alpha$  is implicitly defined). Following Löffler and Mulzer [146], we say that a compressed quadtree is smooth whenever for every uncompressed component  $A \in \mathcal{A}(T)$ :

- (i) the standalone quadtree  $A$  is  $O(1)$ -smooth, and
- (ii) there are at most  $O(1)$  leaf cubes in  $A$  which intersect the minimal bounding square of  $A$ .

Observe that for an arbitrary compressed quadtree  $T$ , property (ii) is not guaranteed. Even if for every uncompressed component  $A \in \mathcal{A}(T)$  we were to make the standalone tree  $A$  smooth, property (ii) is still not guaranteed (refer to Figure 7.14 (a)).



**Figure 7.14** Perimeter cubes are purple. (a) An uncompressed component  $A$  with many leaves that border  $A$ . We show the quadtree as a spacial decomposition and a tree where compressed links are dotted lines. (b) The perimeter  $A^\circ$  consists of 8 cubes. We show the tree that includes  $A^\circ$ : observe that nodes can receive additional children that are perimeter cubes. In addition, we show the set of cubes  $f(A \cup A^\circ)$ .

Given the above definition for a smooth quadtree, we define our extended quadtree  $f(T)$  in two steps. For every uncompressed component  $A \in \mathcal{A}(T)$  we denote by  $|A|$  the diameter of the root of  $A$ . We denote by  $A^\circ$  its *perimeter*, defined as the  $O(3^d)$  cubes with diameter  $|A|$  which intersect the boundary of  $A$  (in two dimensions, the perimeter of a cube  $C$  are the eight cubes: two for each cardinal direction and four that intersect only a vertex of  $C$ ). Note that Löffler and Mulzer call this set the *frontier* of  $A$  [146]. The extended quadtree  $f(T)$  is subsequently defined by adding to the quadtree, for every uncompressed component  $A$ , the set  $A^\circ$  and by applying the previous definition for extended quadtree to each standalone quadtree (Figure 7.14(b)):

$$f(T) := \bigcup_{A \in \mathcal{A}(T)} f(A \cup A^\circ).$$

We add the additional condition that  $f(T)$  contains no duplicates and that for every cube  $C \in f(T)$ , its parent cube in the tree  $f(T)$  is the smallest cube  $C' \in f(T)$  for which  $C$  is strictly contained in  $C'$  (these properties ensure that  $f(T)$  functions as a quadtree). We observe that following this definition, a cube  $C$  may be the parent of several compressed links (at most one uncompressed component  $A$ , and several perimeter cubes). We show that  $f(T)$  functions as a smooth compressed quadtree:

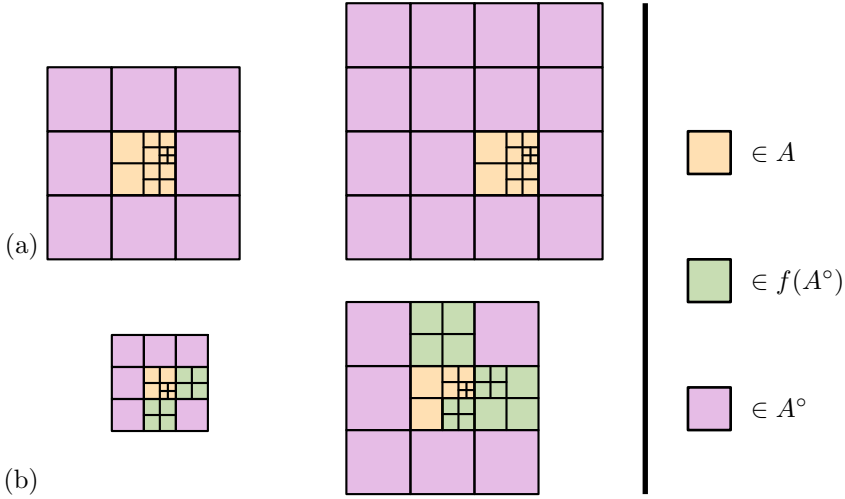
**Theorem 7.6** *Let  $T$  be a compressed quadtree over  $\mathbb{R}^d$ . For any uncompressed component  $A \in \mathcal{A}(T)$ , the standalone quadtree  $f(A \cup A^\circ)$  is  $2^{d+1}$ -smooth and there are at most  $O(6^d)$  leaves in  $f(A \cup A^\circ)$  which intersect the minimal bounding square of  $A \cup A^\circ$ .*

**Proof** Per definition of the extended quadtree, and by Corollary 7.2 the standalone quadtree  $f(A \cup A^\circ)$  is  $2^{d+1}$ -smooth. Löffler and Mulzer note in Section 3.2 of [146] that if we make the set  $(A \cup A^\circ)$  2-smooth, then all cubes that intersect the minimal bounding square of  $(A \cup A^\circ)$  have size at least  $\frac{1}{2}|A|$ . There are at most  $6^d$  cubes of size  $\frac{1}{2}|A|$  in the perimeter of  $A$  and half of them may intersect the border of the perimeter. Our extended tree  $f(A \cup A^\circ)$  is a subtree of this 2-smooth tree and therefore their analysis immediately applies. ■

What remains is to show how to dynamically maintain  $f(T)$ . The following lemma shows that the set  $f(T)$  (without pointers to parent cubes) can be easily maintained:

**Lemma 7.7** *Let  $T$  be an uncompressed quadtree with  $n$  cubes. Then the set  $f(T)$  (without the required quadtree pointers) contains  $O((2d)^{d+2}n)$  cubes and can be maintained in  $O((2d)^{d+2})$  time per split, merge, component insertion or deletion, upgrowing or downgrowing in  $T$ .*

**Proof** Consider an uncompressed component  $A$  with  $k$  cubes and its perimeter  $A^\circ$ . These cubes are conceptually contained within a quadtree  $A'$  with  $O(k)$  cubes (where the root of  $A'$  has size  $2|A|$ , refer to Figure 7.15). It therefore immediately follows from the previous sections that for every split and merge in  $A$ , we can update the extended tree  $f(A \cup A^\circ)$  in  $O((2d)^{d+2})$  time.



**Figure 7.15** (a) An uncompressed component  $A$  with its perimeter in purple. There exists a quadtree (not necessarily aligned with the parent cube of  $A$ ) whose root has size  $2|A|$  that contains all cubes in  $A$ . (b) A quadtree  $A$  and the set  $f(A \cup A^\circ)$ . When upgrowing, we create a component  $A'$  by replacing the root  $R$  of  $A$  by its parent  $R'$  and extend the perimeter. Observe that in  $f(A' \cup (A')^\circ)$ , only cubes adjacent to the original component  $A$  need to be split and at most once (all other cubes required to maintain the smoothness of  $A$  are already present).

Insertions and deletions of a compressed component can be trivially performed in  $O(3^d)$  time. Indeed, whenever we insert an uncompressed component  $A$ , the set  $f(A \cup A^\circ)$  is per definition equal to  $A \cup A^\circ$ . Assuming that we can compute the root of  $A$  in constant time (which is an assumption we made at the beginning of this section), we can trivially create the cubes  $A$  and its perimeter in  $O(3^d)$  time (crucially, we do not yet identify for every cube in the perimeter its parent cube as we maintain just the set  $f(T)$ ). Per definition, we can only delete a compressed component  $A$  when the component consists of a single leaf. Hence we can delete  $A$  and  $A^\circ$  in  $O(3^d)$  time.

Whenever we upgrow a component in  $A$  to create a component  $A'$ , we replace the root  $R$  of the uncompressed component by its parent  $R'$ . The key observation here is that to obtain  $f(A' \cup (A')^\circ)$ , each cube in the perimeter  $(A')^\circ$  only needs to be split at most once (and only when it is adjacent to the original component  $A$ ). Hence, we can create these cubes in  $O(3^d \cdot 2^d)$  time. Via a symmetric argument, downgrowing (where we replace a root  $R$  with one of its children  $R'$  and that we have the added condition that  $R'$  is the only child of  $R'$  which is, possibly, not a leaf) only deletes at most  $O(3^d \cdot 2^d)$  cubes. The observation that the dimension is at least 1 implies that  $O(3^d \cdot 2^d) = O((2^d)^{d+1})$ , and this concludes our argument. ■

The above lemma shows that for every uncompressed component  $A \in \mathcal{A}(T)$ , we can dynamically maintain  $f(A \cup A^\circ)$ . The difficulty encountered in this section, is that in addition we want the tree  $f(T)$  to be a functional quadtree (every cube needs a pointer to its respective parent and  $f(T)$  may not contain duplicate cubes). We illustrate these two properties and why they are nontrivial to maintain in Figure 7.17:

- (a) For every cube  $C$  in  $f(T)$ , we want to have a pointer from and to its parent (the smallest cube  $C'$  in  $f(T)$  that strictly contains  $C$ ).
- (b) Whenever there are two cubes  $C, D \in f(T)$  with  $C = D$ , we want to have pointers between  $C$  and  $D$  (this allows us to remove duplicates).

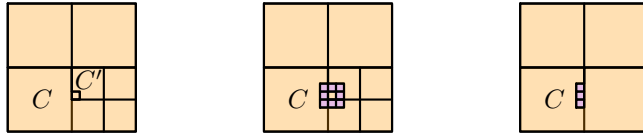
Before we show that we can dynamically maintain the above properties we first show two helper lemmas:

**Lemma 7.8** *Let  $C$  be a cube in an uncompressed component  $A \in \mathcal{A}(f(T))$ , such that  $C$  is a leaf in the standalone quadtree of  $A$ . Then there are at most  $O(18^d)$  cubes  $D$  contained in  $C$  for which  $C$  is the smallest cube in  $f(T)$  that strictly contains  $D$ .*

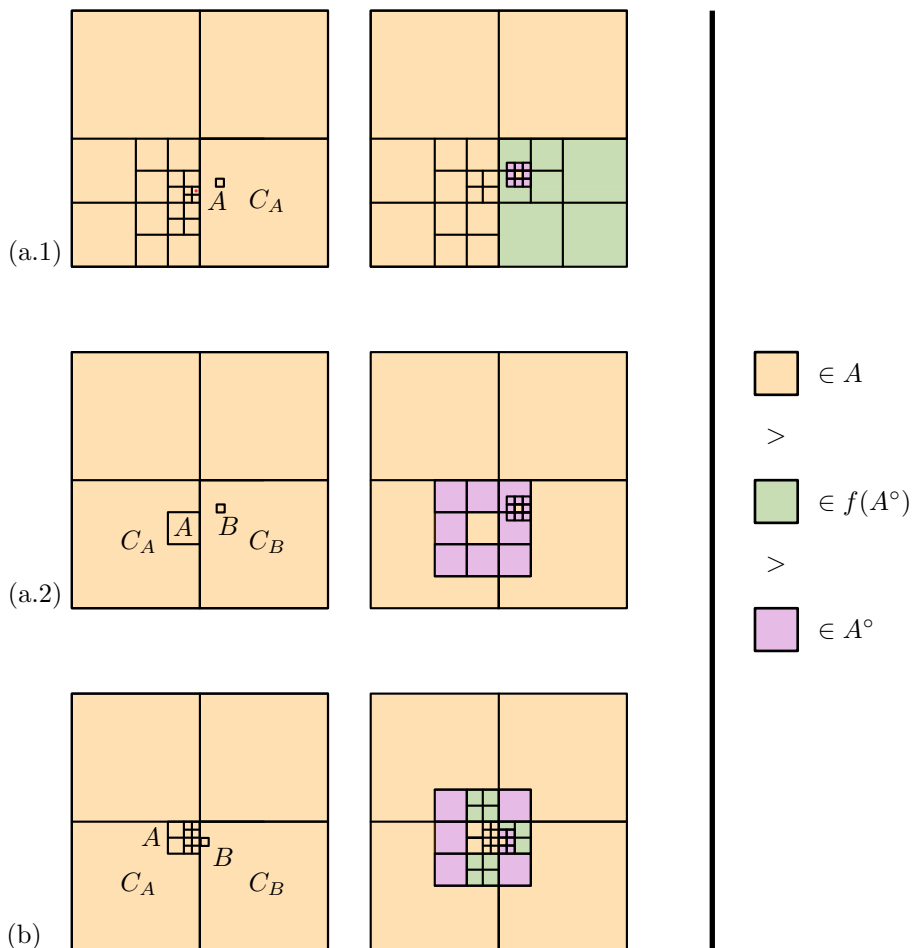
**Proof** The cube  $C$  may contain the root of one uncompressed component plus  $O(3^d)$  perimeter cubes of that component and additional perimeter cubes  $D$  of some uncompressed component  $B \in \mathcal{A}(f(T))$  with  $B$  not in  $C$ .

Let  $D$  be a perimeter cube of some uncompressed component  $B \in \mathcal{A}(f(T))$  with  $B$  not in  $C$  and  $D \subset C$  (Figure 7.16). We first claim that the parent cube  $C'$  of  $B$  is a leaf of the standalone quadtree of  $A$ . Indeed, suppose for the sake of contradiction that  $B$  is an uncompressed component, contained in some uncompressed component  $B' \neq A$ . Then the perimeter of  $B$  is contained in the perimeter of  $B'$ . This contradicts the assumption that  $C$  is the parent of  $D \in B^\circ$ . Our second claim is that  $C'$  intersects the boundary of  $C$  and follows immediately from the observation that  $|D| = |B|$  is (much) smaller than  $C$  and that  $D$  is adjacent to  $B$ .

In the extended tree  $f(T)$ , the standalone quadtree  $A$  is  $2^{d+1}$ -smooth. Thus, for each of the  $O(3^d)$  cubes that intersect a facet of  $C$ , there are at most  $O(2^{d+1})$  leaves that intersect that facet. Since every leaf cube contains at most one uncompressed component, and since every uncompressed component contains at most  $3^d$  cubes in its perimeter, it follows that there are at most:  $O(3^d \cdot 2^{d+1} \cdot 3^d) = O(18^d)$  such cubes  $D$  contained in  $C$ . ■



**Figure 7.16** The construction of Lemma 7.8. We illustrate a quadtree cube  $C$ , and a cube  $C'$  that contains some uncompressed component  $B$ . The perimeter  $B^\circ$  may partially coincide with  $C$ . We count the number of perimeter cubes in  $C$ .



**Figure 7.17** (a.1) An uncompressed component  $A$  within a cube  $C_A$ . In the extended tree  $f(T)$ , the cube  $C_A$  is split. Thus, the parent of  $A$  in  $T$  does not equal the parent of  $A$  in  $f(T)$ . (a.2) An uncompressed component  $A$  within a cube  $C_A$  and an uncompressed component  $B$  within  $C_B$ . If in the extended tree  $f(T)$  we add the perimeter of these two uncompressed components, they overlap. Thus the parent of  $B$  in  $T$  is not equal to the parent of  $B$  in  $f(T)$ . (b) A non-trivial uncompressed component  $A$  and an uncompressed component  $B$ , The perimeter of  $B$  partially coincides with cubes already present in  $T$ .

**Lemma 7.9** *Let  $B$  be the root of an uncompressed component, where  $C$  is the parent of  $B$  and  $C$  is a leaf in a standalone tree  $A$ . Then for all cubes  $D \in B^\circ$ , its parent cube is either:*

1. *the cube  $C \in A$ , or*
2. *a leaf  $C'$  of the standalone quadtree of  $A$  that is adjacent to  $C$ , or*
3. *the perimeter cube of an uncompressed component  $B$  whose parent cube is a leaf of the standalone quadtree of  $A$ , that is adjacent to  $C$ .*

**Proof** Observe that the cube  $D$  must always be contained in the cube corresponding to a leaf of the standalone quadtree  $A \cup A^\circ$  that is adjacent to  $C$  (this leaf does not have to be the parent of  $D$ ). Suppose for the sake of contradiction that the parent of  $D$  is not any of these three options. It follows that there are two distinct possible cases. In the first case, the cube  $D$  is contained in some uncompressed component  $B' \neq A$ . Per definition,  $|B'| > |D|$ . It follows that the parent of the new component  $B$  must then lie in the perimeter of  $B'$ . However, this implies contradicts the assumption of the lemma that  $C$  is contained in an uncompressed component  $A$ . In the second case,  $D$  is contained in the perimeter of some uncompressed component  $B' \neq A$  where the parent cube of  $B'$  is not a leaf in the standalone tree  $A$ . Denote by  $B^*$  the uncompressed component that contains  $B'$ . Per definition  $|B^*| > |B|$ . It follows that the parent cube of  $B$  must lie in  $B^*$  or the perimeter of  $B^*$ , which contradicts the assumption that the parent of  $B$  is a leaf in the standalone tree of  $A$ . It follows that for every cube  $D$ , the parent cube is of  $D$  is either of these three options. ■

We use the above helper lemmas to prove that we can maintain properties (a) and (b):

**Lemma 7.10** *Let  $T$  be an uncompressed quadtree and  $f(T)$  its extended quadtree. We can maintain properties (a) and (b) in:*

- $O(36^d)$  additional time per split and merge in  $f(T)$ ,
- $O(d \cdot 36^d + 54^d)$  time per upgrowing or downgrowing in  $f(T)$ ,
- $O(d \cdot 36^d + 54^d)$  time per insertion or deletion of an uncompressed component.

**Proof** We prove this by showing how to maintain properties (a) and (b) when we split a cube, upgrow, or insert a new uncompressed component. For all three operations, we essentially count all cubes that may either have a new parent, or equal a newly inserted cube. In addition, we observe that we can reach each of these candidates in amortized constant time through traversing neighbor and parent pointers. Given a set of  $k$  candidates, we simply check all of them in  $O(k)$  time and adjust all corresponding pointers as needed. Since our algorithm simply checks all possible candidates in a ‘brute force’ manner, the inverse quadtree operations immediately also maintain properties (a) and (b) through a trivial inverse procedure.

Throughout this proof, we denote by  $C$  a cube in an uncompressed component  $A$  where  $C$  is a leaf in the standalone quadtree of  $A$  ( $C$  is either a leaf of  $f(T)$ , or has a compressed link to an uncompressed component). We detail how we maintain the appropriate pointers after each of the three operations, one by one.

– Suppose that we split the cube  $C$  into  $2^d$  equal-sized cubes.

**Maintaining property (a).** Each of the new child cubes of  $C$  receives a pointer from and to its parent  $C$ . What remains, is to identify the cubes in the extended quadtree  $f(T)$  that now do not have  $C$  as their parent, but a child of  $C$  instead. By Lemma 7.8, there are at most  $O(18^d)$  cubes contained in  $C$  where  $C$  is the corresponding parent. For each of the  $2^d$  children  $C'$  of  $C$ , we check each of these  $O(18^d)$  cubes. When the respective cube is contained in  $C'$ , we relocate the parent pointer from  $C$  to  $C'$ . This maintains property (a) in  $O(36^d)$  total time.

**Maintaining property (b).** Suppose that for one of the new children  $C'$  of  $C$ , there exists a cube  $D \in f(T)$  such that  $C'$  equals  $D$ . There are now two cases: either  $D$  lies in the perimeter of some uncompressed component  $A_2$  or the parent cube of  $D$  is  $C$  ( $\text{PARENT}(D) = C$ ). To identify the latter case, we simply iterate over all outgoing parent pointers from  $C$ . In the former case, we note that by Lemma 7.8 there are at most  $O(18^d)$  such cubes  $D$  contained in  $C$ . We iterate for each of these over all  $2^d$  children of  $C$  to see if a child  $C'$  equals  $D$  in  $O(36^d)$  total time.

– Suppose that we upgrow the uncompressed component  $B$  contained in  $C$ . We replace the root  $R$  of the uncompressed component  $B$  with its parent  $R'$  and we add the  $O(3^d)$  perimeter cubes around  $R$ .

**Maintaining property (a).** The parent of  $R$  is  $R'$  and the parent of  $R'$  is the cube  $C$  that stores the compressed link to the uncompressed component  $B$ . Thus, we can adjust the parent pointers corresponding to  $R$  and  $R'$  in constant time. We subsequently want to identify the cubes  $D \in f(T)$  where  $R'$  (instead of  $C$ ) is now the parent of  $D$ . By Lemma 7.8, there are at most  $O(18^d)$  cubes in  $f(T)$  where  $C$  is the parent of that cube. We check each of them in constant time using the parent pointers from  $C$  and relocate the pointer to  $R'$  if the cube is now contained in  $R'$  instead. What remains is to maintain property (a) for each of the  $O(3^d)$  new perimeter cubes of  $R'$ . By Lemma 7.9, for every cube  $D$  in the perimeter of  $R'$  the parent of  $D$  is one of three options. We count the occurrence of every option. In the first option, the parent of  $D$  is the cube  $C$  and there is only one such cube. In the second option, the parent of  $D$  is a cube  $C'$  of the standalone quadtree that is adjacent to  $C$ . Since the quadtree is  $2^{d+1}$ -smooth, there are at most  $O(d2^{d+1})$  such cubes  $C'$ . In the third option, the parent of  $D$  lies in the perimeter of some uncompressed component  $B$ , whose parent cube is a leaf of the standalone quadtree of  $A$  that is adjacent to  $C$ . Again, there are at most  $O(d \cdot 2^{d+1})$  leaves adjacent to  $C$ . By Lemma 9.3, each of these leaves contains at most  $O(18^d)$  perimeter cubes. It follows that the total number of cubes that could be a parent of a cube  $D$  in the perimeter of  $R'$  is  $O(d \cdot 2^{d+1} \cdot 18^d) = O(d \cdot 36^d)$ . We can reach each of these cubes in amortized constant time through traversing the neighbor and parent pointers. Thus, we identify the parent cube of every cube  $D$  in the perimeter of  $R'$  in  $O(d \cdot 36^d)$  total time. Finally, we need to identify for each of these cubes  $D$ , if

$D$  is now the parent of some other cube in  $f(T)$ . Observe that if  $D$  is the parent of some cube  $C^*$ , then the original parent of  $C^*$  was the parent of  $D$ . By Lemma 7.8, the parent of  $D$  has at most  $O(18^d)$  child cubes. Since we already established a pointer to the parent of  $D$ , we can check each of these in  $O(18^d)$  additional time. The above procedure maintains for a single cube  $D$  in the perimeter of  $R'$  property (a) in  $O(d \cdot 36^d)$  total time by checking the neighbors of  $C$  in all directions. Naively, applying this procedure to all  $O(3^d)$  perimeter cubes would take  $O(d \cdot 36^d \cdot 3^d)$  time. However, both the neighbors of  $C$  and the perimeter cubes have an established cyclic order in  $\mathbb{R}^d$ . Hence, by traversing the neighbors of  $C$  in the same order that we traverse the perimeter cubes, we can check every neighbor of  $C$  at most once and maintain property (a) for all perimeter cubes in  $O(d \cdot 36^d)$  total time.

**Maintaining property (b).** To maintain property (b), we have to identify all cubes in  $f(T)$  that equal  $R'$ , and all cubes in  $f(T)$  that equal a cube  $D$  in the perimeter of  $R'$ . Cubes that equal one another, share the same parent. Since we established the pointers to all parents and, by Lemma 7.8, every such parent has at most  $O(18^d)$  children, it follows that we can maintain property (b) in at most  $O(3^d \cdot 18^d) = O(54^d)$  total time.

– Finally, suppose that we insert a new uncompressed component  $B$  into  $C$ .

**Maintaining property (a).** To maintain property (a), we have to identify for every cube  $D \in B^\circ$ : its parent cube *and* all cubes in  $f(T)$  that have  $D$  as its parent. By Lemma 7.9 the parent of cube  $D$  is one of three options. We count the occurrence of every option. This procedure is identical to the procedure that we deploy when we upgrow an uncompressed component and thus takes  $O(d \cdot 36^d)$  total time.

**Maintaining property (b).** To maintain property (b), we have to identify all cubes in  $f(T)$  that equal the root of  $B$ , and all cubes in  $f(T)$  that equal a cube  $D$  in the perimeter of  $B$ . Again, this procedure is identical to when we upgrow an uncompressed component and thus takes  $O(54^d)$  time. ■

We observe that every split, merge, upgrowing and downgrowing operation can cascade into  $O((2d)^{d+2})$  additional splits and merges to maintain  $f(T)$ . At the same time, inserting or removing an uncompressed component does not cascade into any additional operations as for a single cube  $A$ ,  $f(A \cup A^\circ) = A \cup A^\circ$ . We combine Lemma 7.7 and Lemma 7.10 with this observation to conclude:

**Theorem 7.7** *For any compressed quadtree  $T$  over  $\mathbb{R}^d$  we can, assuming that we can compute the root of new uncompressed components in  $O(1)$  time, maintain the extended  $2^{d+1}$ -smooth quadtree  $f(T)$  in:*

- $O((2d)^{d+2} \cdot 36^d)$  time per split and merge in  $T$ ,
- $O((2d)^{d+3} \cdot 54^d)$  time per upgrowing and downgrowing in  $T$ , and
- $O(d \cdot 36^d + 54^d)$  time per insertion or deletion of an uncompressed component.

## 7.7 Concluding remarks

We showed how to, for an uncompressed quadtree  $T$  over  $\mathbb{R}^d$ , dynamically maintain an extended quadtree  $f(T)$  that is  $2^{d+1}$ -smooth in  $O((2d)^{d+2})$  time per split and merge in  $T$ . If  $T$  is an uncompressed quadtree then the definition of a quadtree  $f(T)$  matches the definition of a  $2^{d+1}$ -smooth quadtree as seen in the literature [22]. This implies that if  $T$  is a quadtree over constant dimension, then we maintain a smooth quadtree with worst-case constant update time, which solves a longstanding open problem [22]. We extend our result to support dynamic planar point location in  $T$  in logarithmic time, whenever  $T$  is a quadtree in the real RAM (in the word RAM point location in a quadtree is trivial). If  $T$  is a quadtree over constant dimension our update time remains worst-case constant.

If  $T$  is a compressed quadtree, we created a new definition of a smooth quadtree  $f(T)$  with the same the properties as the smooth compressed quadtree by Löffler and Mulzer in [146]. Specifically, whenever  $T$  is a compressed quadtree over constant dimension, our smooth quadtree maintains the following desirable properties:

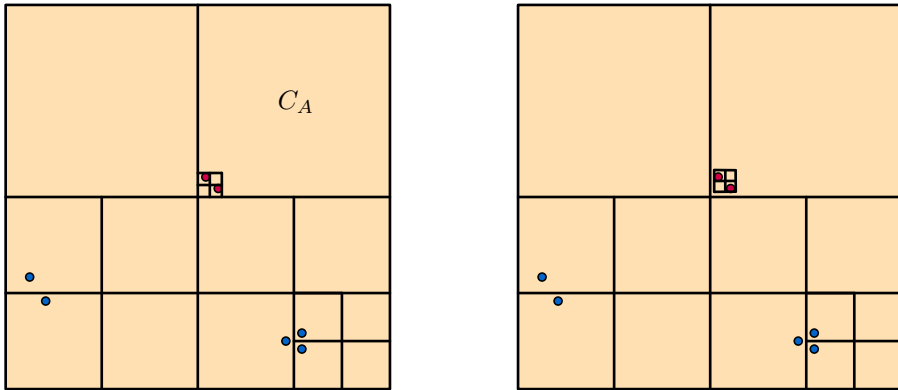
- for every cube  $C \in f(T)$  the cubes in the same uncompressed component that neighbor  $C$  are at most a constant factor larger than  $C$ ,
- the extended quadtree  $f(T)$  can be maintained in constant time, and
- for every uncompressed component  $A$  there are at most constantly many leaves of the standalone quadtree of  $A$  that intersect the border of  $A$ .

These three properties are critical for the analysis by Löffler and Mulzer, in which they show that a smooth compressed quadtree  $T(P)$  that stores a planar point set of  $n$  points can be converted into, amongst others, the Euclidean Minimum Spanning tree of  $P$  in linear time.

**Storing point sets.** For a point set  $P$  and a fixed bounding box  $R$  that contains  $P$ , the (compressed) quadtree  $T(P)$  is the unique compressed minimal quadtree where every leaf of  $T(P)$  contains at most one point of  $P$ . The results in this chapter allow us to store a constant-dimension point set in a linear-size quadtree. Indeed, it is known that if  $p$  contains  $n$  points then  $T(P)$  contains at most  $O(n^d)$  cubes. Hence, if  $P$  is a point set over constant dimension then the tree  $T(P)$  uses linear space. Constant-time location in a linear-size word RAM quadtree is trivial with the use of the floor operation. If  $P$  is a point set in the word RAM with the typical word size  $w = O(\log n)$ , the tree  $T(P)$  has linear size without the use of compression. If, on the other hand,  $P$  is a real-valued point set of  $n$  points then the tree  $T(P)$  may be a compressed quadtree to maintain the property that  $T(P)$  contains  $O(n^d)$  cubes. In addition, for such a quadtree we require the additional work from Section 7.5 to support point locations:

If  $P$  is a point set over constant dimension then Theorem 7.5 implies that we can perform point locations in logarithmic time. Theorem 7.7 shows how to maintain the extended compressed quadtree  $f(T(P))$  in constant time per point addition or removal (once the leaf that contains the point in question is identified). Indeed, any point may be inserted with either a constant number of splits, a single upgrowing operation or by inserting a single new uncompressed component. Thus, its inverse also requires at most a constant number of operations. However, Theorem 7.7 operates on the assumption that whenever we insert an uncompressed component  $A$  into a leaf  $C \in T$ , we can compute the cube  $R$  corresponding to the root of  $A$  (and with it, the number  $n_R$ ) in constant time. In the word RAM, this is not an issue as the tree  $T(P)$  does not need to be compressed. However, the real RAM supports point sets that break this assumption as there exists point sets  $P$  where there are pairs of very close points, such that computing the root of an uncompressed component of the quadtree  $T(P)$  requires an arbitrarily high number of divisions.

**Alignment issues and future work.** The above problem that accompanies real-valued point sets is a well-known issue for quadtrees in a real RAM. For constructing static quadtrees there is a classical solution [38, 114, 151]: when inserting the root of a new uncompressed component  $A$ , we do not compute  $|A|$  exactly. Instead, we compute some cube  $R'$  smaller than  $|A|$ , that roughly approximates the size  $|A|$  and use this as the root of  $A$  instead (in the same way, we can approximate the number  $n_R$  by computing  $n_{R'}$ ). We say that  $A$  and its parent are not *aligned* (Figure 7.18).



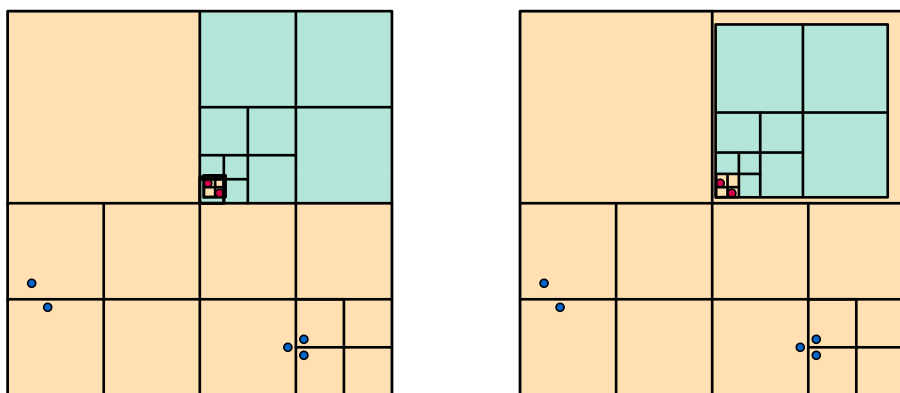
**Figure 7.18** Computing the exact minimal quadtree square  $A$  within  $C_A$  that contains both red points might be a costly operation. Instead, we can compute a square of roughly the appropriate size that contains both red points and that is contained within  $C_A$  in constant time. We call the root of this uncompressed component  $A$ , *not aligned* with  $C_A$ .

This nonalignment causes problems in a dynamic setting. Suppose that for a cube  $C$  that stores a compressed link to a nonaligned component  $A$ , we keep splitting  $C$ . If we repeatedly split  $C$ , we eventually create the cube  $C'$  that is the parent of the root of  $A$ . However, if  $A$  is not aligned with  $C$  it may not be aligned with its parent  $C'$ . Thus, in this scenario, the children of  $C'$  no longer partition  $C'$ . This is problematic because it violates one of our core properties of the quadtree (refer to Figure 7.19).

It is not obvious how to remedy this problem. However, with some added restrictions this problem is solvable. We solve this problem in each of the two cases by spending additional computation time after every operation on  $T$ . Suppose that at all times:

- the compressed quadtree  $T$  contains a constant number of compressed links.
  - In this case, whenever we perform an operation we spend additional constant computation time for every compressed link to (partially) compute the correct root of the uncompressed component  $A$  attached to that link. When we have already computed the correct link, we relocate  $O(2^d)$  cubes in  $A$  so that they align with the correct root (essentially, we spread the realignment of  $A$  over all future split and merge operations). Since there are at most a constant number of compressed links, it must be that for every uncompressed component  $A$  all cubes in  $A$  are realigned with the correct root before an insertion joins  $A$  with its parent component.
- every root-to-leaf path in  $T$  contains at most one compressed link.
  - In this case, whenever we perform an operation on a cube  $C$ , we identify the unique compressed link  $L$  that lies on at least one root-to-leaf path that contains  $C$  in constant time. We spend constant additional time to (partially) compute the correct root of the uncompressed component  $A$  attached to  $L$ . When we have already computed the correct link, we relocate  $O(2^d)$  cubes in  $A$  so that they align with the correct root in the same way as above.

Both algorithmic schemes guarantee that if we ‘join’ a cube  $C$  with a compressed link to a component  $A$  then we have already computed the correct root of  $A$  and realigned all cubes in  $A$  with that root. Indeed, if the compressed path  $(C, \dots, A)$  has length  $L$  then computing the exact coordinates of the root can be done in  $O(\log L)$  time whereas ‘joining’  $C$  with  $A$  requires at least  $O(L)$  split and upgrow operations. Whenever the quadtree  $T$  does not contain any of these two properties, it becomes unclear which computation needs to be prioritized. Thus, our result does not allow us to maintain an arbitrary real-valued point set  $P$ , but rather one whose quadtree follows our restrictions. Future work could study how to dynamically maintain a smooth compressed quadtree on a real RAM, without our added alignment assumption.



**Figure 7.19** Suppose that we recursively split cubes in  $C_A$ . Then eventually, we create the cube  $C'$  that would have been the root of  $A$ . The cube  $C'$  is now *misaligned* with  $A$  and intersects  $A$ . Similarly, whenever we upgrow  $A$  we eventually create the cube that would coincide with  $C_A$ : but is now misaligned. A combination of split and upgrow operations can create this misalignment at any cube on the path from  $C_A$  to  $A$ .



## PART IV

# **Imprecise points**



## Chapter Eight

# Geometric imprecision

A fundamental assumption in classic algorithmic analysis is that the input data given to an algorithm is exact. Clearly this assumption is generally not justified in practice: real-world data tends to have (measurement or labeling) errors, heterogeneous data sources introduce yet other type of errors. We present four concrete examples:

1. The first example is input data derived from GPS samples. A GPS sample models the locations associated with a set of entities. Since GPS samples have an inherent significant measuring error, the algorithmic input that results from a round of GPS sampling is an inaccurate measurement (representation) of the set of locations it expresses.
2. In the second example we assume that we have a perfectly accurate measurement of the location of objects in motion. Even in this example the resulting algorithmic input is not precise: during the time that passes between the measurement and the algorithmic computation the objects in motion can change locations. Hence the algorithmic input that results from the accurate measurement is an inaccurate representation of their current location (this gives an error bounded by the object's speed and the time that has passed).
3. In the third example we study a probability distribution where we are interested in the median outcome of the distribution. Every sample of the distribution gives an inaccurate representation of some 'median' point set.
4. The final example is input obtained from floating point arithmetic. Sharma and Yap note in Chapter 45 in the Handbook of Discrete and Computational Geometry [186], that computations performed with bounded bitprecision result in an inaccurate representation of the actual value (where the error is bounded by the number of bits available for the computation).

To increase the relevance of algorithmic techniques for practical applications, various paradigms for dealing with uncertain data have been introduced including:

- a paradigm to upper bound the eventual error after a computation on imprecise values (whose imprecision itself is bounded by some parameter) such as the work by Segal [182] and Sugihara and Iri [189],
- a paradigm to represent geometric primitives as matrices to perform exact geometric computations (EGCs), often through a *lazy evaluation*. This includes work by Yap and Dubé [203] and Burnikel et al. [42]. The well-used CGAL CORE library uses the EGC principle [207].
- a paradigm where the uncertainty, imprecision or error of a data point is represented as a *disk* in a suitable distance metric. The famous example is  $\varepsilon$ -geometry by Guibas, Salesin, and Stolfi [110]. Here, the error is assumed to be bounded by some known value  $\varepsilon$  and the goal. The algorithmic input is a collection of disks with radius  $\varepsilon$  and the goal is to compute the output with some bound on the computational error.

The four examples that create geometric imprecision have the following property in common: for each of the imprecise values it would be possible to obtain a more precise representation of the value at some (computational) cost. Indeed, GPS samples can be refined through sending several location pings, moving objects can be measured again, probability distributions can be sampled more frequently to obtain a more precise image of the median outcome and for floating point arithmetic it is possible to run the computation with more bits to receive a more accurate representation. It is for this reason that in many papers about geometric imprecision that use the third paradigm, the authors additionally assume that it is possible to replace each disk with a more exact value at some computational cost [1, 40, 68, 87, 88, 118, 127, 144, 147, 149, 150, 161, 65, 195, 197]).

Held and Mitchell [117] introduced the *preprocessing model* to study if faster than  $\Omega(n \log n)$  convex hull construction is possible under additional assumptions. On an intuitive level, the preprocessing model functions as follows: algorithmic input is delivered in two stages. In the first stage, the input is a set of geometric regions  $\mathcal{R}$ . In the second stage, we may retrieve for every region  $R \in \mathcal{R}$  a corresponding exact value  $p \in R$ . Löffler notes that this models imprecise data [145]: the center of every geometric region  $R$  models the imprecise sampling of some true value  $p \in R$ . The degree of imprecision is represented by the size of the uncertainty region. Retrieving for every region  $R$  the point  $p \in R$  captures the intuitive notion that it is possible to obtain a more precise representation of the value, when so desired. Part IV of this thesis is dedicated to studying algorithmic problems within this preprocessing model. We argue for a novel definition of output in the preprocessing model. Using this novel definition of output, we show a nontrivial definition for algorithmic lower bound (called uncertainty-region lower bound) in the preprocessing model. This lower bound is more strict than the well-known worst-case lower bound (any algorithm that matches our uncertainty-region lower bound matches the worst-case lower bound) and can be seen as input-sensitive towards the input  $\mathcal{R}$ .

## 8.1 Introduction

We elaborate on the motivation for the preprocessing model and introduce a new variant that, we argue, more closely matches the motivation for this model. The preprocessing model models imprecise input with what are called uncertainty regions. Specifically, in this model the algorithmic input is a set of geometric (uncertainty) regions  $\mathcal{R} = (R_1, R_2, \dots, R_n)$  with an associated “true” planar point set  $P = (p_1, p_2, \dots, p_n)$ . For any pair  $(\mathcal{R}, P)$ , we say that  $P$  *respects*  $\mathcal{R}$  if each  $p_i$  lies inside its associated region  $R_i$ ; we assume that when we receive  $P$ , it respects  $\mathcal{R}$ . This model has two consecutive phases: a preprocessing phase where we have access only to  $\mathcal{R}$  and a reconstruction phase where we can, for each  $R_i \in \mathcal{R}$ , retrieve the true location  $p_i$  for some unspecified cost  $C$ . We want to preprocess  $\mathcal{R}$  to create some auxiliary structure  $\Xi_{\mathcal{R}}$ . Afterwards, we want to reconstruct some output on  $P$  using  $\Xi_{\mathcal{R}}$  faster than would be possible without preprocessing.

**Motivation for the preprocessing model.** Besides the four examples mentioned in the introduction, research into the preprocessing model cites additional motivation for this use. This model of computation follows a tradition of algorithmic analysis under partial information: Kahn and Kim [123] (later Cardinal, Jungers and Munro [47]) study sorting under partial information: given a partially ordered set  $\mathcal{R}$ , how fast can one compute an underlying linear order  $P$ ? The preprocessing model is a geometric generalisation of this question, where the partial order (or a partial construction of some combinatorial output) is implied by the uncertainty regions.

Van Kreveld, Löffler and Mitchell [197] note that this model is useful when sampling data from a probability distribution: suppose that we want to study a structure on a planar point set that is sampled from a suitably dense probability distribution per point. Then each point is expected to lie within a bounded region. We are then interested in preprocessing the set of regions, such that given a sample we can construct the structure efficiently.

Busto, Evans and Kirkpatrick [43] provide a practical use case for the preprocessing model: consider a set of  $n$  moving entities that are tracked by GPS. One can query a moving entity  $e$  and receive its exact location. However, as time passes, the set of possible locations of  $e$  grows. Hence  $e$  can be modeled as a region, for which a true location can be queried.

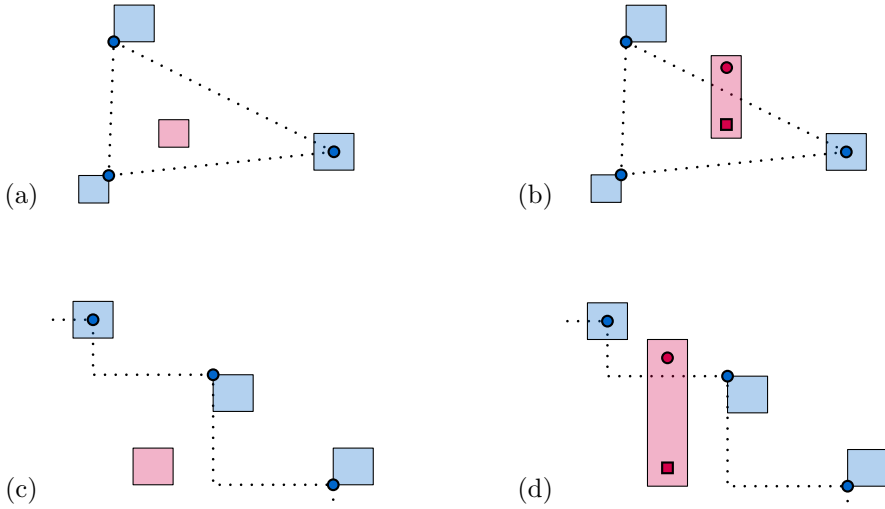
Finally, Bruce et al. [35] mention the Exact Geometric Computation paradigm that assumes that the algorithmic input is the result of some prior (floating point) computation. When comparing two values that originate from a floating point computation, it is possible to obtain a more precise representation of each involved value; albeit pre-stored on disk memory or through some normal form representation. We model the cost of obtaining an exact value by  $C$  and in this context it is clear that  $C$  can equal a large number of RAM instructions.

### 8.1.1 The preprocessing model with indirect representations

In the preprocessing model, the input is a set of geometric (uncertainty) regions  $\mathcal{R} = (R_1 \dots R_n)$  with an associated “true” planar point set  $P = (p_1 \dots p_n)$ . The model has two consecutive phases: a *preprocessing* phase where we have access only to  $\mathcal{R}$  and a *reconstruction* phase where we can, for each  $R_i \in \mathcal{R}$ , retrieve the true location  $p_i \in R_i$  for some fixed cost  $C$  (depending on the application,  $C$  may be large or small or even 0. For example, in I/O-sensitive computations  $C$  is often assumed to be so large that a single point retrieval dominates a polynomial number of RAM instructions [35]). The idea is to preprocess  $\mathcal{R}$  to create some auxiliary structure  $\Xi_{\mathcal{R}}$ . Afterwards, we want to reconstruct some output  $S(P)$  (using  $\Xi_{\mathcal{R}}$ ) faster than would be possible without preprocessing.

One possible interpretation of reconstruction is to explicitly return the desired output  $S(P)$  as a structure based on a set of true points. For example, the output might be a polygon whose vertices are equipped with the precise coordinates of the points which form the convex hull of  $P$ . This interpretation was widely adopted within computational geometry and there are many recent results for constructing Delaunay triangulations [39, 40, 68, 197], convex hulls [87, 88, 150, 166] and other planar decompositions [147, 195]. As all points in the output must be retrieved, this setting has a trivial worst-case lower bound of  $\Omega(n)$  for the reconstruction phase. However, in some cases, it is possible to determine the combinatorial structure of the output directly from the uncertainty regions, without retrieving the true points (which may be faster than  $\Omega(n)$ , see Figure 8.1). There can be two reasons for this: first, some points may not occur on the output and thus do not need to be retrieved (we wish to note that this could be captured by the notion of output-complexity). Second, for some points the fact that they are part of the output can be deduced from just their uncertainty region. We hence propose a different interpretation of reconstruction to enable more fine-grained analysis. Instead of returning  $S(P)$  after the reconstruction phase, we want to return an *indirect representation*  $\Xi_P^*$ . Formally, we assume that the desired algorithmic output  $S(P)$  is a labeled graph where labels are points in  $P$  (e.g. a convex hull is a cyclic graph of points in  $P$ ). The output  $\Xi_P^*$  is a labeled graph where every label is a region in  $\mathcal{R}$  such that if we replace every  $R_i$  with  $p_i$ , we obtain  $S(P)$ .

**Examples of  $\Xi_P^*$  depending on the structure.** We provide three examples of such indirect output  $\Xi_P^*$ . The first example is the convex hull, where the output  $\Xi_P^*$  is a graph on the regions  $\mathcal{R}$  such that if we replace every vertex  $R_i$  in the graph with the point  $p_i$  we obtain the convex hull of  $P$ . The second example is the Pareto front (sometimes called the set of maximal points). For two points  $p$  and  $q$  in  $\mathbb{R}^2$ , we say that  $p$  *dominates*  $q$  if both its  $x$ - and  $y$ -coordinates are greater than or equal to the respective coordinates of  $q$ . The *Pareto front* of  $P$  is the boundary of all points in  $\mathbb{R}^2$  that are dominated by a point in  $P$ . Given  $P$ , the output  $\Xi_P^*$  is a maximal staircase graph where top vertices are regions  $R_i$  such that  $p_i$  is not dominated by a point in  $P$ . The third example is a quadtree, where given a bounding box that contains  $P$ , the output  $\Xi_P^*$  is a quadtree such that every leaf contains a constant number of points.



**Figure 8.1** (a) The structure of the convex hull of  $P$  follows from  $\mathcal{R}$ . (b) The true point in the red region determines the combinatorial outcome. (c/d) A similar construction for the Pareto front.

## 8.2 Results and implications

In Part III we show how to preprocess a set of regions  $\mathcal{R}$  to efficiently construct an indirect representation of a sorted order, or the Pareto front of the underlying true point set  $P$ . In both cases, we show input-sensitive optimal results. Specifically, we show that for every set of regions  $\mathcal{R}$  there can be no algorithm that (for *every* point set  $P$  that respects  $\mathcal{R}$ ) computes an indirect output  $\Xi_P^*$  faster than our solution. This non-trivial lower bound contrasts prior results in the preprocessing model, which constructed the explicit output  $S(P)$  in worst-case optimal time due to the  $\Omega(n)$  lower bound of the output size. Our implicit representation and the following algorithmic optimality have three implications for future literature on the preprocessing model.

The first implication is with respect to the motivation for the model. Recall the third motivation, where we preprocess  $\mathcal{R}$  such that given a randomly generated sample  $P$  we want to efficiently compute the output on  $P$  to estimate the ‘average’ algorithmic outcome. Prior results in this model require that for every sample  $P$ , the following construction takes linear time. With an indirect representation this computation can be sub-linear (which allows much more efficient computations of the ‘average’ output). Moreover, consider the fourth motivation where uncertain values represent outcomes of computations and where retrieving a point at cost  $C$  is performing an Exact Geometric Computation. Prior results demand that the user performs an exact geometric computation on every value in  $P$  which, in the context, is not sensible. In our results, the number of EGCs performed is optimal for every instance  $(\mathcal{R}, P)$ .

The second implication is with respect to algorithmic analysis. There is a subfield of theoretical computer science with a focus on efficient partial computations. In this computational paradigm, described by Lin, Natarajan and Lui [142], the input is some partially computed structure or information on some set of values  $P$  and the goal is to compute the final output on  $P$  as efficiently as possible. The goal is to analyse the partial structure or information and reason about how efficient the outcome on  $P$  can be constructed. The classical example is partial sorting [47, 123] where the input is a partial order on  $P$  and the goal is to output the sorted order of  $P$  as fast as possible. The preprocessing model is a geometric interpretation of this paradigm: the partial information is all geometric information that can be extracted from the uncertainty regions  $\mathcal{R}$  before the reconstruction phase and the goal is to output the outcome as efficiently as possible. It is not immediately clear what ‘as efficiently as possible’ means: many results in this area for example explicitly do not count the time required to import and analyse all partial information and instead formalise a limited set of computational instructions that they *do* count for the algorithmic analysis. E.g. when sorting under partial information, it is assumed that one receives the partial order on  $P$  as some directed graph and one counts only the number of comparisons between elements in  $P$ . The preprocessing model with indirect representation further formalises how to count the time required to complete the partial computation on  $P$ : we assume that in the reconstruction phase one is free to compute *any* data structure on the partial information (the uncertainty regions  $\mathcal{R}$ ). For a large class of desired output  $\Xi_P^*$ , we show for any set of regions  $\mathcal{R}$ , a nontrivial corresponding lower bound (irrespective of the data structure on  $\mathcal{R}$ ) on the time required to compute the output  $\Xi_P^*$ . This new form of lower bound is a natural definition of algorithmic optimality, whose granularity falls in between worst-case and instance optimality.

Lastly, Part III is the first step towards studying arbitrarily overlapping input  $\mathcal{R}$  in the preprocessing model. Many prior results in the preprocessing model require the set of input regions  $\mathcal{R}$  to be pairwise disjoint. More recent results [87, 86, 129, 148] relax the assumption that the regions are pairwise disjoint and instead parametrize the degree of overlap with a global parameter. Specifically, for any set of regions  $\mathcal{R}$  the *ply* is the maximum over all points, of the number of regions that intersect that point. These works show algorithmic running times that either require the ply to be a constant, or that use the ply as a parameter in their running time. As soon as the ply becomes not constant, these result either do not apply or have an undesirable running time. One of the reasons for the lack of results for regions with arbitrary overlap, is that analysing algorithmic performance was, until now, not possible. Indeed, when the regions may overlap arbitrarily there is a worst-case lower bound of  $\Omega(n \log n)$  for the reconstruction phase which equals the lower bound of their counterparts in classical geometry. Hence there exist worst-case optimal algorithms that ignore the regions, which negates the use of the model. The new definition of uncertainty-region optimality provides the first (algorithmically matchable) qualitative measure for algorithms that consider overlapping input.

## 8.3 Algorithmic lower bounds

This section is dedicated to defining the uncertainty region lower bound for the preprocessing model using implicit representations. The remaining chapters of this part will show how to preprocess regions  $\mathcal{R}$ , to construct the indirect output  $\Xi_P^*$  in time that matches this uncertainty region lower bound.

### 8.3.1 Lower bounds and the definition of an algorithm

We first review the definition of worst-case lower bound together with the three main techniques to obtain such a lower bound. Then we discuss models of computation that facilitate this lower bound. The folklore worst-case lower bound definition of an algorithmic problem  $\mathcal{P}$  with input  $x$  of size  $n$  is:

$$\text{Worst-case lower bound}(\mathcal{P}) := \min_A \max_{x \in [0,1]^n} \text{Runtime}(A, x),$$

where each  $A$  is an algorithm that solves the algorithmic problem  $\mathcal{P}$  for some definition of solving. Afshani, Barbay and Chan [2] observe that there are three common techniques to prove lower bounds:

- direct arguments based on counting, or *information theory*;
- topological arguments, as used by Yao [202] or Ben-Or [18] (sometimes directly referred to as *algebraic decision tree* arguments); or
- arguments based on Ramsey theory, as used by Moran, Snir and Manber [164].

Additional techniques for obtaining lower bounds include an adversarial argument as by Erickson [81] (or the more recent argument by Chan [49]) but we restrict our attention to the information-theoretic arguments presented in the list. Such an information-theoretic argument decomposes algorithms into what is called an *algebraic decision tree* and reasons about its depth. This construction can only be formalised as soon as the definition of the model of computation is fixed.

**Models of computation.** We briefly review the models of computation used for worst-case lower bounds in algorithmic analysis. The classical model of computation by Ben-Or [18] is the abstract algebraic decision tree model. This model assumes that given a fixed input size  $n$ , any algorithmic computation on input of size  $n$  can be modeled by a binary algebraic decision tree. The leaves of this tree contain a possible outcome of the algorithmic computation, for a given input of size  $n$ . Every outcome has at least one corresponding leaf in the tree. At each node in the binary tree, a binary decision is taken. This binary decision compares an algebraic polynomial whose variables depend on the input that is evaluated. The algorithm branches based on the outcome of the algebraic evaluation until the algorithm reaches a leaf which contains the algorithmic outcome.

This model of computation facilitates the classical information-theoretic lower bound. Suppose that for a given algorithmic problem  $\mathcal{P}$  and fixed input size  $n$ , the set  $e(\mathcal{P})$  is the set of all different algorithmic outcomes. Then  $\Omega(\log |e(\mathcal{P})|)$  is a worst-case lower bound in this computational model. Indeed, the decision tree contains at least  $|e(\mathcal{P})|$  leaves. Thus, there exists at least one input that requires  $\Omega(\log |e(\mathcal{P})|)$  binary decisions before reaching the outcome.

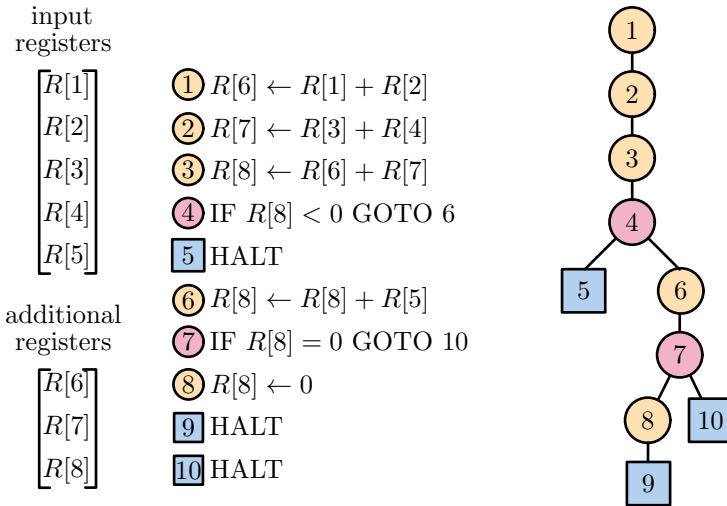
Afshani, Barbay and Chan [2] observe that the computational power that comes from the abstract algebraic decision tree model by Ben-Or (where every node in the tree can contain an arbitrary algebraic test, where the function is unbounded in the number of arguments or the total degree) is unrealistically great. They show an involved example where this model of computation does not allow for the fine-grained analysis that is required for constructing input-sensitive lower bounds. They define a model of computation that is analogous to the abstract algebraic decision tree model by Ben-Or, where they add the following restriction: at every node of the decision tree, the function that is evaluated is required to be a multilinear function (a function that is linear, separate in each of its variables) with a constant number of variables. This algebraic decision tree supports the information-theoretic  $\Omega(\log |e(\mathcal{P})|)$  lower bound via the same argument as Ben-Or, but in addition allows for fine-grained analysis for the information provided by partial computations. We share the opinion of Afshani, Barbay and Chan that a computational model that allows arbitrary algebraic computations in constant time is unrealistically powerful. However, we note that their alternative model is perhaps too restrictive, as it becomes difficult, if not impossible, to express common computations using only constant-variate multilinear functions. Consider for example a real-weighted cycle graph of  $n$  vertices, where the input marks two vertices  $a, b$  and we ask if the distance between  $a$  and  $b$  is at most  $X$ . For at least one of the paths from  $a$  to  $b$ , the length of the path is an algebraic expression that contains a linear number of variables. Hence, if we assume that we can only evaluate constant-length expressions, it is impossible to decide if the length of such a path is less than  $X$ .

For our analysis, we offer a model of computation that:

- supports the information-theoretic lower bound of  $\Omega(\log |e(\mathcal{P})|)$ ,
- avoids the unreasonable computational power of the abstract algebraic decision trees,
- allows for the evaluation of decision functions that are polynomials with arbitrary variables and arbitrary total degree,
- supports (and counts!) operations that are not comparisons such as accessing memory.

Specifically, in Part II of this thesis, we (re)defined the real RAM. Throughout this chapter, we use this definition of RAM to define our lower bounds. For completeness, we recall in the next section its definition and show how it enables the classical information theoretic lower bound that originates from the depth of the decision tree, even when dealing with a pre-stored data structure.

**The real RAM definition and decision trees.** In Part II of this thesis, we defined the real RAM in two steps. First, we define computations based on the (discrete) word RAM, so that discrete memory can be accessed without unreasonable computational power. Then, we augment the word RAM with separate real-valued computations that only work on values stored within the discrete memory cells. Operations include memory manipulation, real arithmetic and comparisons (which verifies if the real value stored in a memory cell is greater than 0). For an extensive overview of the computations that they allow, we refer to Table 3.1. We say a program on the real RAM consists of a fixed, finite indexed sequence of read-only instructions. The machine maintains an integer *program counter*, which is initially equal to 1. At each time step, the machine executes the instruction indicated by the program counter. Every real RAM operation increases the program counter by one, apart from a comparison operation. A comparison operation compares the value in any (real or word) register and verifies if it is greater or equal to 0. The value in any register can be represented as a polynomial that depends on the input value where the total complexity of the polynomials in all registers is bounded by the running time of the algorithm (for example, if we want to evaluate an expression that is the sum of  $O(n)$  variables, we need to spend  $\Omega(n)$  time to construct this expression first). The comparison operation ends in a GOTO statement that can set the program counter to any discrete value.



**Figure 8.2** All programs can be transformed into a tree where every vertex has degree at most 3 and all leaves terminate the program. Every decision leaf represents an algebraic expression on the input variables. For example, the 4-th instruction contains the expression:  $R[1] + R[2] - R[3] - R[4] < 0$ . Constructing an expression of  $k$  variables requires at least  $\Omega(k)$  instructions.

We show that this model facilitates a decomposition of an algorithm into an algebraic decision tree (Figure 8.2). Indeed, let the input size  $n$  be fixed and  $\mathcal{P}$  be an algorithmic problem such that  $e(\mathcal{P})$  is the set of all distinct algorithmic outcomes (for all input of size  $n$ ). Each outcome in  $e(\mathcal{P})$  may be described by the sequence of instructions that construct the outcome and place it in the memory registers, together with a HALT instruction that tells the program to stop and output the result. Suppose that our program contains no comparison operations. Then after every operation, the program counter increases by one and program must always reach the same HALT instruction (if any). Via this same argument, if the program contains one comparison operation it may reach one of two possible HALT instructions. It follows that algorithm can be transformed into a tree where the HALT instructions are the leaves of the tree, all instructions other than comparison instructions have a single child and comparison instructions have two children. This intuitive transformation of every algorithm facilitates the information-theoretic lower bound: if there are at least  $|e(\mathcal{P})|$  outcomes and the algorithm can reach every outcome (an assumption guaranteed by the assumption that the algorithm is *correct*), then this tree has at least  $|e(\mathcal{P})|$  leaves. Since every node in the tree has at most two children, there is at least one input that requires  $\Omega(\log |e(\mathcal{P})|)$  instructions before we reach the corresponding leaf. This model does not have the unrealistic computational power of the abstract algebraic decision trees: as it takes computational time to construct the decision function in a comparison node. At the same time it is less restrictive than the model of Afshani, Barbay and Chan as, given enough computational time, it allows for decision nodes to contain arbitrarily complex expressions.

### 8.3.2 Reviewing lower bounds

At the beginning of this section we noted a that folklore worst-case lower bound definition of an algorithmic problem  $\mathcal{P}$  with input  $X$  of size  $n$  is the minimum over all algorithms  $A$ , of the worst-case running time of  $A$ :

$$\text{Worst-case lower bound}(\mathcal{P}) := \min_A \max_{X \in [0,1]^n} \text{Runtime}(A, X),$$

A natural more refined lower bound is the instance lower bound. Given an algorithmic problem  $\mathcal{P}$  with input (instance)  $X$ , the *instance lower bound* is defined as:

$$\text{Instance lower bound}(\mathcal{P}, X) := \min_A \text{Runtime}(A, X).$$

We illustrate the difference between the worst-case lower bound and the instance lower bound via the searching problem where the input  $X$  is sorted set of  $n$  numbers  $X$  and some value  $q$ . The goal is to report the index of the number in  $X$  that equals  $q$  (if such a number exists). Depending on  $q$ , the outcome could be any index and the set of all outcomes  $e(\mathcal{P})$  thus contains  $n$  elements. Via the information-theoretic lower bound, the worst-case lower bound for this algorithmic problem is hence  $\Omega(\log |e(\mathcal{P})|) = \Omega(\log n)$ . In contrast, the instance lower bound when  $q \in X$  is  $\Omega(1)$ :

Suppose that we fix an instance  $(X, q)$  with  $q \in X$ . Then there exists a “lucky” algorithm that guesses the location in the sequence of  $X$  that contains the number in  $X$  nearest to  $q$  in constant time, and simply outputs the index in  $O(1)$  time. An algorithm is instance optimal if for every instance  $(X, q)$  the algorithm matches the instance lower bound. We know that in every comparison-based model there can be no algorithm that solves the searching problem in constant time thus we have found an example where instance optimality is unobtainable.

Instance-optimality can be seen as the holy grail for fine-grained algorithmic analysis. As instance optimality is provably unobtainable in many cases, it is unsurprising that there is a rich tradition of finding alternative algorithmic analyses that capture an algorithmic performance that is better than worst-case optimality. To reach a degree of optimality that lies in between instance and worst-case optimality, many approaches parametrize the algorithmic problem. For example, there is *output-sensitive* analysis [132]. Here, for any input instance  $X$  the size  $k$  of the corresponding output is a lower bound for the time required to construct that output. Output-sensitive algorithms have a running time where the output size  $k$  is a parameter of the runtime expression. Other parameters include geometric restrictions such as *fatness*, the *spread* of the input, and the number of *reflex vertices* in a (simple) polygon.

Such parameters, however, are hard to apply in the preprocessing model with indirect representation. This is because the auxiliary structure  $\Xi_{\mathcal{R}}$  allows one to bypass the natural lower bound that these parameters bring. For example: the  $\Omega(k)$  output-sensitive lower bound is not applicable, as output of any size can be computed in the preprocessing phase. The reconstruction phase algorithm may subsequently only consist of a HALT instruction that refers to the precomputed output in  $O(1)$  time.

**A note on referencing versus reporting output.** The above example, and the remainder of Part IV, assumes that we may ‘reference’ the output. That is, we may terminate our program whenever the output is present in the registers and simply output pointers to the location of the output. We explicitly assume that we are not required to spend  $O(k)$  time to ‘print’ or ‘report’ a pointer structure of  $O(k)$  size. In many cases (after preprocessing), the size of the output dominates the time required to construct that output. Hence, by discarding the additional  $O(k)$  time required to report the output we allow for a more fine-grained algorithmic analysis (e.g. many of the prior results in the preprocessing model had linear reconstruction time. Maybe, with some additional algorithmic analysis, some of these approaches can have faster than linear running time. However, as  $\Omega(n)$  is a lower bound for reporting these linear-size structures there was no incentive to further investigate the efficiency of these approaches). Apart from the clear benefit for algorithmic analysis, we also wish to note a practical justification for this assumption. Many results in the preprocessing model (including those in this thesis) construct some sort of data structure which, presumably, will be used for queries after the construction. In our algorithmic analysis, we terminate a program as soon as this data structure is present within the memory as from that point on, one may immediately start querying the structure on the device.

### 8.3.3 Lower bound definitions in the preprocessing model

Given our review of previous lower bounds and lower bound techniques we are ready to discuss the definitions of lower bounds that we use in the remainder of this thesis. An algorithm  $A$  in the preprocessing model consists of two pairs of algorithms  $A = (A_{\text{pre}}, A_{\text{rec}})$ . The algorithm  $A_{\text{pre}}$  is the preprocessing algorithm where the input are the uncertainty regions  $\mathcal{R}$  and the output is the auxiliary structure  $\Xi_{\mathcal{R}}$ . The algorithm  $A_{\text{rec}}$  is the reconstruction algorithm that constructs the output  $\Xi_P^*$ . It has access to  $\Xi_{\mathcal{R}}$  and can for every region  $R_i \in \mathcal{R}$ , retrieve the corresponding point  $p_i \in P$  in  $O(C)$  time (where  $C$  is some fixed number of RAM instructions). We show definitions for lower bounds of the running time of  $A_{\text{rec}}$  (the running time in the reconstruction phase). To this end, we denote for any input pair  $(\mathcal{R}, P)$  and any algorithm pair  $A = (A_{\text{pre}}, A_{\text{rec}})$  by  $\text{ReconstructionTime}(A, \mathcal{R}, P)$  the time required by  $A_{\text{rec}}$  to compute the output  $\Xi_P^*$  in the reconstruction phase (where time is the number of real RAM instructions before we reach the HALT instruction).

**Worst-case lower bounds.** The worst-case comparison-based lower bound of an algorithmic problem  $\mathcal{P}$  considers each algorithm pair  $A = (A_{\text{pre}}, A_{\text{rec}})$ . The worst-case lower bound then considers for each  $(A, \mathcal{R})$  the maximal running time over each input  $(\mathcal{R}, P)$ :

$$\text{Worst-case lower bound}(\mathcal{P}) := \min_A \max_{(\mathcal{R}, P)} \text{ReconstructionTime}(A, \mathcal{R}, P).$$

We briefly translate the information-theoretic lower bound to the preprocessing model. We assumed that the output for the preprocessing model was some labeled graph  $\Xi_P^*$ . Fix an input size  $n$  and denote by  $e(\mathcal{P})$  the set of all outputs  $\Xi_P^*$  for all pairs  $(\mathcal{R}, P)$  where  $\mathcal{R}$  and  $P$  contain  $n$  regions and points respectively. The algorithm  $A_{\text{rec}}$  can be decomposed into a decision tree where every node is a single real RAM instruction and has at most two children and where each leaf is an element of  $e(\mathcal{P})$ . Since there are at least  $|e(\mathcal{P})|$  leaves, there exists a pair  $(\mathcal{R}, P)$  where  $A_{\text{rec}}$  requires at least  $O(\log |e(\mathcal{P})|)$  instructions to reach the corresponding leaf of the decision tree and output the result. This lower bound hence holds regardless of the structure  $\Xi$  computed by  $A_{\text{pre}}$ .

**Instance lower bounds.** The instance lower bound ([2, 85]) is stronger than the worst-case lower bound since any algorithm that matches the instance lower bound also matches the worst-case lower bound. In this setting, for a given instance  $(\mathcal{R}, P)$ , the corresponding instance lower bound is defined as:

$$\text{Instance lower bound}(\mathcal{P}, \mathcal{R}, P) = \min_A \text{ReconstructionTime}(A, \mathcal{R}, P).$$

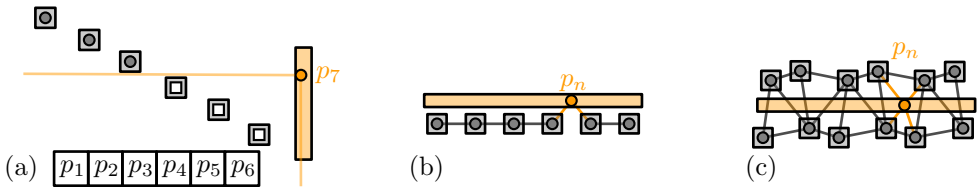
An algorithm pair  $A$  is instance optimal if for every input instance  $(\mathcal{R}, P)$  the runtime of  $A_{\text{rec}}$  matches the instance lower bound. We prove the following:

**Theorem 8.1** *Let the retrieval cost  $C$  be  $o(\log n)$  RAM instructions and  $\mathcal{R}$  be any set of pairwise disjoint uncertainty rectangles. Then there exists no algorithm  $A$  in the preprocessing model with indirect representation that can construct a Pareto front, EMST, GG, Delaunay triangulation or convex hull on the true points which is instance optimal.*

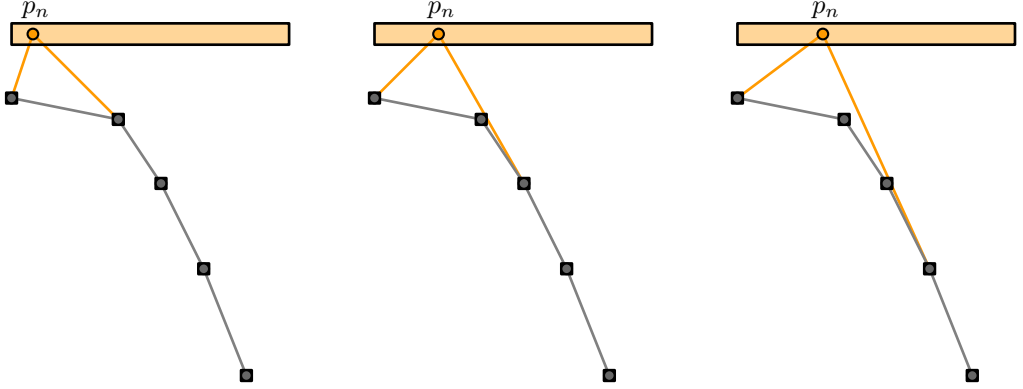
**Proof** Fix the desired data structure to be either a Pareto front, Euclidean minimum spanning tree, Gabriel Graph, or convex hull. Per assumption of the preprocessing model with indirect representation, each of these structures is a labeled graph where the labels of the graph correspond to points in  $P$ . In these algorithmic problems, for each vertex of the output  $\Xi_P^*$  that corresponds to a point  $p \in P$ , the neighboring vertices in  $\Xi_P^*$  correspond to points  $p'$  that are adjacent to  $p$  in the structure (e.g. the neighbors of a point  $p$  on the convex hull are the two points that precede and succeed  $p$  in the cyclic order of the cyclic graph that is the output  $\Xi_P^*$ ).

Fix any of the desired output structures. We can create a set  $\mathcal{R}' = (R_1, R_2, \dots, R_{n-1})$  of uncertainty regions for which the indirect data structure  $\Xi_P^*$  of  $\mathcal{R}'$  can be known in the preprocessing phase (Figures 8.3 and 8.4) and all points corresponding to regions in  $\mathcal{R}'$  appear on the structure (e.g. all points appear on the Pareto front). For each of these sets  $\mathcal{R}'$ , we can construct a region  $R_n$  with the following property: for any  $p_j \in (p_1, \dots, p_{n-1})$  there exists a point  $p \in R_n$  such that  $p$  appears in the structure  $\Xi_P^*$ , with a constant number of neighbors, one of which is  $p_j$ . See Figure 8.3 for an example of the Pareto front, the EMST, GG, and the Delaunay triangulation (with it, Voronoi diagrams) and Figure 8.4 for the convex hull. Hence, given the set  $\mathcal{R} := \mathcal{R}' \cup \{R_n\}$ , there are at least  $(n - 1)$  different combinatorial outcomes for the final structure  $\Xi_P^*$  of  $\mathcal{R}$  based on the placement of the point in  $R_n$ .

Via the information theoretic lower bound (and additionally, through a reduction from the searching problem) there is no algorithm  $A$  that for every instance can decide the correct neighbor of  $p_n$  faster than  $\Omega(C + \log n)$  time. Yet, for every instance, there exists a naive algorithm that correctly guesses the constantly many neighbors of  $p_n$  and verifies this guess in  $O(C)$  time. Thus, there can be no algorithm that matches the instance lower bound for every choice of  $p_n$ . ■



**Figure 8.3** The indirect Pareto front (a), EMST + GG (b) or Delaunay triangulation (c) of the grey points is implied by the regions; The point  $p_n$  can neighbor any grey point in the final structure.



**Figure 8.4** A collection of  $n - 1$  grey pairwise-disjoint uncertainty rectangles, for which the convex hull of their underlying points is implied by the convex hull of their bottom left vertices. The region  $R_n$  is shown in orange. Depending on the placement of  $p_n$ , it can neighbor any grey point in the convex hull of all the points.

**Uncertainty-region lower bounds.** Worst-case optimality is easily attainable and we proved that instance optimality is unattainable in the preprocessing model. Yet for the examples in Figure 8.1(a,c) and 8.3(a), we have a matchable information-theoretic lower bound of  $\Theta(1)$ , and  $\Theta(\log n + C)$ . We capture this intuition for a problem  $\mathcal{P}$  with a given set of uncertainty regions  $\mathcal{R}$ :

$$\text{Uncertainty-region lower bound}(\mathcal{P}, \mathcal{R}) := \min_A \max_{(P \text{ that respects } \mathcal{R})} \text{ReconstructionTime}(A, \mathcal{R}, P),$$

and say an algorithm pair  $A$  is uncertainty-region optimal if for every  $\mathcal{R}$ ,  $A_{\text{rec}}$  has a running time that matches the uncertainty-region lower bound. Denote by  $|e(\mathcal{R})|$  the number of distinct outcomes for all  $P$  that respect  $\mathcal{R}$ . Via the information theoretic lower bound we know:

$$\forall \mathcal{R}, \quad \log |e(\mathcal{R})| \leq \Omega(\text{Uncertainty-region lower bound}(\mathcal{P}, \mathcal{R})).$$

**Challenges of Part IV.** In the remainder of part IV, we try to match the uncertainty-region lower bound for computing a sorted order or Pareto front on  $P$  and we encounter two main challenges. First, for a given  $\mathcal{R}$ , computing  $\log |e(\mathcal{R})|$  might be  $\#P$ -hard (see [47] for hardness results on partial information). To match this lower bound, we use the value  $\log |e(\mathcal{R})|$  without ever explicitly computing (an approximation of) its actual value. Secondly, for constructing any aforementioned planar structure, the value of  $\log |e(\mathcal{R})|$  ranges between 0 and  $n \log n$  (when regions are tiny, or overlapping respectively). Consequently, an optimal algorithm cannot necessarily afford to explicitly retrieve the entire point set  $P$  and we need to smartly use pre-stored data.

## Chapter Nine

# Imprecise order

In this chapter we study how to efficiently order a set of imprecise points. In one dimension, the order of a set of points is their sorted order. A set of imprecise points in the preprocessing model consists of a set of  $n$  uncertainty regions  $\mathcal{R} = \{R_1, R_2, \dots, R_n\}$  and a set of  $n$  points  $P = \{p_1, p_2, \dots, p_n\}$  such that for every  $R_i \in \mathcal{R}$  there is an associated point  $p_i \in P$  with  $p_i \in R_i$ . In one dimension, the set  $\mathcal{R}$  is a set of intervals which induces a partial order such that the total order of the underlying true points  $P$  extends that partial order. We show how to preprocess the partial order induced by  $\mathcal{R}$ , such that given the point set  $P$  we can uncover the underlying total order in uncertainty-region optimal time. Specifically, we parametrize the degree of overlap by the intervals with a measure we call the *ambiguity* of the set  $\mathcal{R}$  and we show that the ambiguity of  $\mathcal{R}$  is an uncertainty-region lower bound for the time required to sort the points  $P$ . This chapter can be seen as a geometric variant of sorting under partial information, which is a well-studied topic within computer science.

## 9.1 Introduction

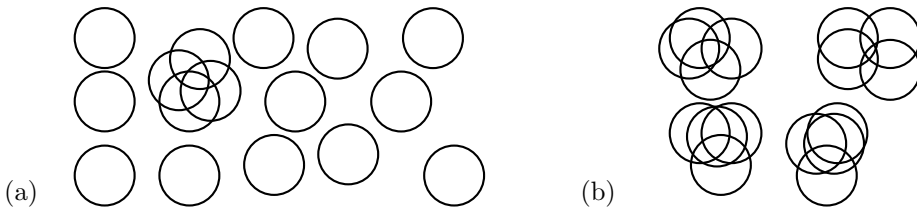
Sorting under partial information is a well-studied problem in computer science [47, 96, 123]. This algorithmic problem is often defined as follows: Let  $P = \{p_1, p_2, \dots, p_n\}$  be a set equipped with an unknown linear order. Given a subset of the relations  $p_i < p_j$ , determine the complete linear order by queries of the form: ‘is  $p_i < p_j$ ’

On an intuitive level, this problem statement is clear: we want to sort  $P$  as efficiently as possible using *only* comparisons between elements in  $P$  and the information given by the partial order. The definition of a real RAM pointer machine in Chapter 2 allows us to formally define the above problem statement: we are given a partial order  $O$  on a set of  $n$  values  $P = (p_1, p_2, \dots, p_n)$ . The goal is to preprocess the partial order  $O$  such that given  $P$ , one can construct a linked list on the indices of  $P$  (where index  $i$  precedes  $j$  in the linked list if and only if  $p_i < p_j$ ) as efficiently as possible. We now adopt the terminology of the preprocessing model with indirect representation and state prior results using this terminology. We refer to this two-stage process as *preprocessing*  $O$  and then *reconstructing* the sorted order of  $P$ . The implicit output  $\Xi_P^*$  is a linked list where elements uniquely correspond to points in  $P$  such that the traversal of this linked list matches the sorted order on  $P$ . This definition matches the intuitive definition of output presented by Kahn and Kim [123]. Given this framework, we review prior results on sorting under partial information and then adapt the problem definition to a geometric setting.

**Sorting under partial information and entropy.** For any partial order  $O$  on  $P$ , we denote by  $e(O)$  the set of linear orders that extend  $O$ . For every such linear order, there exists a unique output  $\Xi_P^*$ . Thus, the previous chapter shows that in the real RAM,  $\Omega(\log |e(O)|)$  is a lower bound on the construction of a linked list on  $P$ . Fredman [96] already showed in the 1970s that  $\Omega(\log |e(O)|)$  is a lower bound for discovering the sorted order of  $P$ . Brightwell and Winkler prove that computing the number of linear extensions  $e(P)$  is  $\#P$ -complete [32]. Thus, efforts have concentrated on computing approximations of this value to show algorithmic optimality. The most notable approximation uses the concept of graph entropy as introduced by Körner [134]. For any graph  $G$ , the graph entropy  $H(G)$  intuitively resembles the degree of information in every vertex of the graph. Kahn and Kim [123] note that given a partial order  $O$ , we can construct what is called a *conflict* graph  $G(O)$  on the set  $P$  where there is an edge between two values  $p, p'$  if both  $p < p'$  and  $p > p'$  are not implied by  $O$ . They prove that the entropy of this graph  $H(G(O))$  implies a lower bound for constructing the sorted order of  $P$  by showing that  $\log |e(O)| = \Theta(n \cdot H(G))$ . To the best of our knowledge there is currently no exact algorithm to compute  $H(G(O))$ . Cardinal et al. [47] describe the fastest known algorithm to approximate  $H(G(O))$ , which runs in  $O(n^{2.5})$  time. They then use  $H(G(O))$  for their algorithmic analysis to show that their construction of the sorted order of  $P$  is optimal.

**Sorting in the preprocessing model for imprecise points.** In this chapter, we study the problem by Kahn and Kim, restricted to a geometric setting using the preprocessing model with indirect representation. In the preprocessing model for imprecise points, the point set  $P$  is some imprecise, one-dimensional point set. The uncertainty regions  $\mathcal{R}$  of  $P$  are intervals and the intervals induce a partial order on  $P$  where  $p_i <_{\mathcal{R}} p_j$  whenever the interval  $R_i$  is disjoint from  $R_j$  and strictly left of  $R_j$ . In the preprocessing model for imprecise points with indirect representations, the goal is to preprocess  $\mathcal{R}$  such that given  $P$  we can construct a linked list  $\Xi_P^*$  on the regions  $\mathcal{R}$  such that if we replace all regions  $R_i$  by their point  $p_i$  we obtain a linked list on  $P$  that is the sorted order on  $P$ . In the remainder of this chapter, we assume that the costs  $C$  to retrieve for every region  $R_i$  the corresponding point  $p_i$  is some fixed constant (in the concluding remarks, we elaborate on how to lift this assumption). The goal is to preprocess  $\mathcal{R}$ , such that we can reconstruct  $\Xi_P^*$  in uncertainty-region optimal time.

Several prior results in the preprocessing model, for various planar geometric problems, restrict  $\mathcal{R}$  to be a set of disjoint (unit) disks in the plane, while others consider partially overlapping disks. The *ply* of  $\mathcal{R}$ , denoted by  $\Delta(\mathcal{R})$ , is the maximum over all points of the number of disks that overlap in that point. We can use the ply as a parameter of our running time but when doing so, the ply may be prove to be a too coarse measure of the degree of overlap. Indeed, consider the following simple iterative reconstruction algorithm: at every iteration we retrieve for a region  $R_i \in \mathcal{R}$  the point  $p_i$ . If the point  $p_i$  is contained in  $k$  uncertainty intervals, we retrieve all  $k$  points and locally sort them in  $O(k \log k)$  time. By definition of the ply, this is at most  $O(\Delta(\mathcal{R}) \log \Delta(\mathcal{R}))$  time. A simple charging argument shows that the total time spent by this simple algorithm is at most  $O(n \log \Delta(\mathcal{R}))$ . However, this running time may severely overestimate the time this simple reconstruction algorithm takes. Consider a set of uncertainty intervals (one-dimensional disks) where  $O(\sqrt{n})$  intervals overlap in a single point and the remainder are pairwise disjoint. By using the ply, we estimate that this naive algorithm takes at most  $O(n \log \Delta(\mathcal{R})) = O(n \log n)$  time whereas in reality the running time of this algorithm is at most  $O(n + \sqrt{n} \log n)$ . The point of this example is to illustrate that for algorithmic analysis we want a more fine-grained and localised measure of overlap.



**Figure 9.1** Two sets of 16 disks each in the plane, both with a ply of 4. The ambiguity of the set in (b) is four times as large as the ambiguity of the set in (a), which captures the intuition that there seems to be four times as much overlap.

### 9.1.1 Contribution

In this chapter we introduce the *ambiguity*  $A(\mathcal{R})$  as a more fine-grained measure of the degree of overlap in  $\mathcal{R}$ . The ambiguity is based on the number of regions each individual region intersects (see Figure 9.1). We count this number with respect to particular permutations of the regions: for each region we count only the overlap with regions that appear earlier in the permutation. A proper technical definition of ambiguity can be found in Section 9.2. We also show how to compute a 3-approximation of the ambiguity in  $O(n \log n)$  time.

In one dimension,  $\mathcal{R}$  is a set of intervals and the ambiguity is linked to the aforementioned graph entropy. Specifically, given  $\mathcal{R}$ , denote by  $G(\mathcal{R})$  the conflict graph induced by  $\mathcal{R}$  (the graph on  $P$  where there is an edge between  $p, p' \in P$  if  $p < p'$  or  $p' < p$  is not implied by  $\mathcal{R}$ ). This graph is equal to the intersection graph of  $\mathcal{R}$ . We prove that the ambiguity  $A(\mathcal{R})$  provides a constant-factor approximation of the entropy of  $G(\mathcal{R})$  (see Section 9.2). If we denote by  $e(\mathcal{R})$  the set of linear orders that extend the partial order induced by  $\mathcal{R}$  then we additionally show that the ambiguity provides a constant-factor approximation of  $\log |e(\mathcal{R})|$ . Since we can compute a constant-factor approximation of the ambiguity in  $O(n \log n)$  time, we can hence also compute a constant-factor approximation of the entropy of interval graphs in  $O(n \log n)$  time, thereby improving the result by Cardinal et al. [47] for the special case where the partial order is induced by a set of intervals.

**Ambiguity and reconstruction.** The ambiguity  $A(\mathcal{R})$ , for a set of  $n$  regions  $\mathcal{R}$ , ranges between  $\Theta(1)$  and  $\Theta(n \log n)$ . If all intervals are pairwise disjoint, then  $A(\mathcal{R}) = \Theta(1)$  and we have essentially no time for the reconstruction phase (the structure  $\Xi_P^*$  then has to be constructed during preprocessing). In this case, we construct a linked list on the left-to-right order of the regions in  $\mathcal{R}$ . In Section 9.3, we show how to preprocess arbitrary sets of intervals to reconstruct the sorted order. That is, we describe how to construct in  $O(n \log n)$  time an auxiliary data structure  $\Xi_{\mathcal{R}}$  on  $\mathcal{R}$  in the preprocessing phase (without access to  $P$ ). In the reconstruction phase (using  $P$ ), we can construct a linear-size AVL-tree  $T$  on  $\mathcal{R}$  where the leaves of the AVL-tree are connected in a linked list, such that if we replace every region  $R_i$  with the corresponding point  $p_i$  this linked list follows the sorted order of  $P$  (Figure 9.2). Our result is uncertainty-region optimal and therefore, tight.



**Figure 9.2** A set of five overlapping intervals  $\mathcal{R}$  with a corresponding point set  $P$ . The output  $\Xi^*$  is an AVL tree where each leaf references a unique region in  $\mathcal{R}$ , such that if we replace each  $R_i$  with  $p_i$ , we obtain the sorted order of  $P$ .

## 9.2 Ambiguity

We introduce a new measure on a set of regions  $\mathcal{R}$  to reflect the degree of overlap of the uncertainty regions which we call their *ambiguity*. We show that if  $\mathcal{R}$  is a set of uncertainty intervals in one dimension then the ambiguity is an uncertainty-region lower bound for constructing a linked list on the sorted order of some point set underlying  $\mathcal{R}$ . The ambiguity is defined as the minimum over all permutations  $\pi$  of  $\mathcal{R}$ , of the product over all regions  $R_i$ , of the number of regions intersected by  $R_i$  that precede or equal  $R_i$  in the permutation  $\pi$ .

Formally we denote by  $\pi$  a permutation of  $n$  elements and by  $\mathcal{R}^\pi = (R_{\pi(1)}, \dots, R_{\pi(n)})$  the sequence of uncertainty regions ordered by  $\pi$ . We denote by  $\mathcal{R}_{\leq i}^\pi := \{R_j \in \mathcal{R} \mid \pi(j) \leq \pi(i)\}$  a prefix of this sequence with  $i$  elements. A permutation  $\pi$  is *containment-compatible* if  $R_i \subset R_j$  implies  $\pi(i) < \pi(j)$  for all  $i$  and  $j$  [91].

**Contact sets (with respect to a permutation  $\pi$ ).** For a region  $R_i$  we define its *contact set*  $\Gamma_i^\pi$  with respect to a permutation  $\pi$  as the set of regions which precede or equal  $R_i$  in the order  $\pi$ , and which intersect  $R_i$ :

$$\Gamma_i^\pi := \{R_j \in \mathcal{R}_{\leq \pi(i)}^\pi \mid R_j \cap R_i \neq \emptyset\}.$$

Observe that a region  $R_i$  is always in its own contact set. A region  $R_i$  whose contact set  $\Gamma_i^\pi$  contains only  $R_i$  itself is called a *leaf region* with respect to  $\pi$  (refer to Figure 9.3).

**Ambiguity.** Using contact sets we define the Ambiguity in two steps. For a set of regions  $\mathcal{R}$  and a fixed permutation  $\pi$  we define the  $\pi$ -ambiguity as (Figure 9.4):

$$A^\pi(\mathcal{R}) := \sum_i \log |\Gamma_i^\pi|.$$

Observe that whenever a region  $R_i$  is a leaf region with respect to  $\pi$ , it does not contribute to the value of the  $\pi$ -ambiguity. The ambiguity of  $\mathcal{R}$  is defined as the minimal  $\pi$ -ambiguity over all permutations  $\pi$ :

$$A(\mathcal{R}) := \min_{\pi \in \Pi} A^\pi(\mathcal{R}).$$



**Figure 9.3** A set of four overlapping intervals. Given a permutation, we can draw the intervals ordered in the  $y$ -direction. In this figure,  $\Gamma_3^\pi = \{R_1, R_2, R_3\}$ .

We show the following properties of ambiguity:

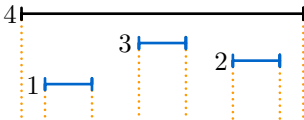
1. the  $\pi$ -ambiguity may vary significantly with the choice of the processing permutation  $\pi$ ,
2. in one dimension, the  $\pi$ -ambiguity for any containment-compatible permutation  $\pi$  on a set of intervals  $\mathcal{R}$  implies a 3-approximation of the uncertainty-region lower bound (the value is also a 3-approximation on the entropy of the interval graph of  $\mathbb{R}$ ), and
3. the permutation that realizes the ambiguity is containment-compatible.

It follows that in one dimension, the ambiguity of a set of intervals  $\mathcal{R}$  implies a 3-approximation of the entropy of the interval graph of  $\mathcal{R}$ . We start with the first property: it is easy to see that the processing permutation  $\pi$  has a significant influence on the value of the  $\pi$ -ambiguity:

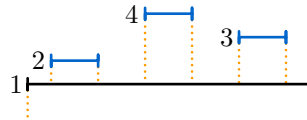
**Lemma 9.1** *For all  $n \in \mathbb{N}$ , there exists a set of  $n$  intervals  $\mathcal{R}$  such that there are two permutations  $\pi_1$  and  $\pi_2$  and:*

$$\frac{A^{\pi_2}(\mathcal{R})}{A^{\pi_1}(\mathcal{R})} = \Omega\left(\frac{n}{\log n}\right).$$

**Proof** Consider a set  $\mathcal{R}$  of  $n$  intervals where  $n - 1$  intervals are pairwise disjoint and the last interval contains all the others (refer to Figure 9.4). Let  $\pi_1$  be a permutation where the short intervals precede the long interval. Then the contact set of each short interval contains only the interval itself, and each short interval has a contribution of  $\log 1 = 0$  to the  $\pi_1$ -ambiguity. The contact set of the long interval consists of all the intervals in  $\mathcal{R}$ , and its contribution to the  $\pi_1$ -ambiguity is  $\log n$ . Thus, the  $\pi_1$ -ambiguity  $A^{\pi_1}(\mathcal{R}) = \log n$ . Now, let  $\pi_2$  be a permutation where the long interval precedes all the short intervals. Then the contact set of each short interval has a size two, and each short interval has a contribution of  $\log 2 = 1$  to the  $\pi_2$ -ambiguity. The contact set of the long interval contains only the interval itself, thus the  $\pi_2$ -ambiguity  $A^{\pi_2}(\mathcal{R}) = n - 1$ . This concludes the proof. ■



$$A^{\pi_1}(\mathcal{R}) = \log 1 + \log 1 + \log 1 + \log 4$$



$$A^{\pi_2}(\mathcal{R}) = \log 1 + \log 2 + \log 2 + \log 2$$

**Figure 9.4** An example of the  $\pi$ -ambiguity induced by permutations  $\pi_1$  and  $\pi_2$ . This construction, when generalised, shows that the difference between the  $\pi$ -ambiguity for the same set of regions can be  $\Omega(n/\log n)$ .

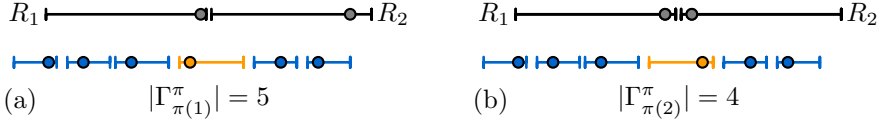
Even though  $\pi$ -ambiguity can vary considerably, we show that if we restrict the permutations to be containment-compatible then the range of their  $\pi$ -ambiguities narrows down to within only a constant factor of the ambiguity (and, hence, the  $\pi$ -ambiguity for all containment-compatible  $\pi$ ).

### 9.2.1 Ambiguity as a lower bound for sorting

We want to show that for any set of regions  $\mathcal{R}$ , the ambiguity  $A(\mathcal{R})$  lower bounds  $\log |e(\mathcal{R})|$ : the logarithm of number of linear extensions of the partial order induced by  $\mathcal{R}$ . If we can show this, then we have shown that the ambiguity lower bounds the uncertainty region lower bound for constructing  $\Xi_P^*$  (a linked list on  $\mathcal{R}$ , such that when we replace every region  $R_i$  with the point  $p_i$ , the linked list is the sorted order of  $P$ ). Moreover, via [123], this implies that we lower bound the entropy of the graph  $G(\mathcal{R})$  (the intersection graph of  $\mathcal{R}$ ). We achieve this by showing that for any containment-compatible permutation  $\pi$  the  $\pi$ -ambiguity  $A^\pi(\mathcal{R})$  implies a lower bound for  $\log |e(\mathcal{R})|$ . We show a stronger result as we show that the  $\pi$ -ambiguity is even a 4-approximation of  $\log |e(\mathcal{R})|$  (and  $nH(\mathcal{R})$  which we explain later):

**Theorem 9.1** *For any set of intervals  $\mathcal{R}$  in one dimension, for any containment-compatible permutation  $\pi$  on  $\mathcal{R}$ ,  $A^\pi(\mathcal{R})$  is a 4-approximation of  $nH(\mathcal{R})$  and  $\log |e(\mathcal{R})|$ .*

**Discussing the naive approach for lower bounding  $\log e(\mathcal{R})$ .** We first want to discuss the “naive” approach to lower bound  $\log e(\mathcal{R})$  before we provide an actual lower bound using an involved intermediary result. Suppose for a given set of regions  $\mathcal{R}$  and permutation  $\pi$  there is a region  $R_i$  where its contact set has at least 5 elements ( $|\Gamma_{\pi(i)}^\pi| = 5$ ). In this scenario, we may argue that there are at least five different linear extensions of the partial order induced by  $\mathcal{R}$  (for an illustration of the construction, refer to Figure 9.5). Indeed, these linear extensions are obtained by placing, for every  $R_k \in \Gamma_i$ , the point  $p_k$  in  $R_k$  and by placing  $p_i$  anywhere between any of these four points. Similarly, if for another region  $R_j$ ,  $|\Gamma_{\pi(j)}^\pi|$  is at least 4 then there are at least four different linear extensions of the partial order induced by  $\mathcal{R}$ . Ideally, we would given  $R_i$  and  $R_j$  immediately reason that there are at least  $|\Gamma_{\pi(i)}^\pi| \cdot |\Gamma_{\pi(j)}^\pi|$  linear extensions of the partial order induced by  $\mathcal{R}$  (and eventually, we do). However, when making this argument we encounter the following problem: suppose that there exists a region  $R_k$  which is both in  $\Gamma_{\pi(i)}^\pi$  and  $\Gamma_{\pi(j)}^\pi$ . Then (if  $R_i$  and  $R_j$  are disjoint) it may be that the point  $p_k$  is present in either  $R_i$ , or  $R_j$ , but never both. Thus, the above argument overcounts the number of linear extensions of the partial order induced by  $\mathcal{R}$ . In this chapter, to avoid this overcounting problem, we choose to relate the ambiguity to the value  $H(\mathcal{R})$ : the entropy of the intersection graph of  $\mathcal{R}$  that was recently studied by Cardinal et al. [47]. This value  $H(\mathcal{R})$  subsequently provides a lower bound on  $\log e(\mathcal{R})$ .

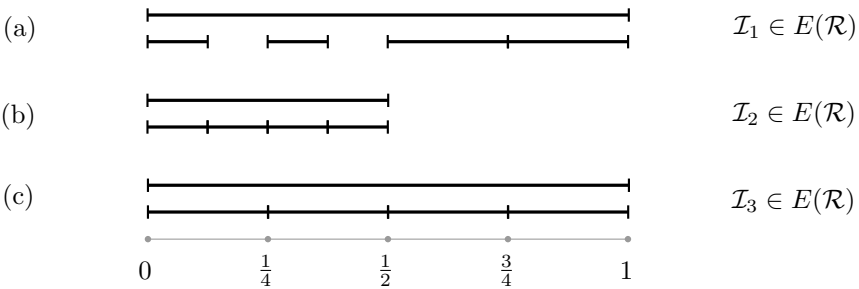


**Figure 9.5** Let  $\pi$  be a containment-compatible permutation such that  $R_1$  and  $R_2$  are the last two regions in the ordering, then  $|\Gamma_{\pi(1)}^\pi| = 5$  and  $|\Gamma_{\pi(2)}^\pi| = 4$ . If  $p_1$  succeeds the orange point then  $p_2$  must also succeed the orange point.

**Interval entropy and its relation to interval embeddings.** We define the *interval entropy*  $H(\mathcal{R})$ , for a set of intervals  $\mathcal{R}$ , as the entropy of the intersection graph of  $\mathcal{R}$ . While investigating sorting under partial information, Cardinal et al. [47] found an interesting geometrical representation of the interval entropy. Let  $\mathcal{R}$  be any set of intervals, we say that a set  $\mathcal{I}$  is an *embedding* of  $\mathcal{R}$  on  $[0, 1]$  if there is a unique correspondence between intervals  $I_i \in \mathcal{I}$  and regions  $R_i \in \mathcal{R}$  where  $\mathcal{I}$  induces the same partial order as  $\mathcal{R}$  (that is each interval  $I_i$  is disjoint from and precedes  $I_j$  if and only if this is true for  $R_i$  and  $R_j$ ). We denote by  $E^{[0,1]}(\mathcal{R})$  the set of all such embeddings  $\mathcal{I}$  on  $[0, 1]$ . Given this notation, we can state the following result by Cardinal et al. [47] (refer to Figure 9.6 for an illustration):

**Lemma 9.2 ([47], Lemma 3.2 paraphrased)**

$$n \cdot H(\mathcal{R}) = \Theta(\log |e(\mathcal{R})|) \quad \text{and} \quad H(\mathcal{R}) = \log n - \min_{\mathcal{I} \in E^{[0,1]}(\mathcal{R})} \left\{ \frac{1}{n} \sum_{I_i \in \mathcal{I}} -\log |I_i| \right\}.$$



**Figure 9.6** Let  $\mathcal{R}$  be a set of five intervals, where four intervals are mutually disjoint and contained in one larger interval. We show three embeddings  $\mathcal{I}_1, \mathcal{I}_2, \mathcal{I}_3 \in E(\mathcal{R})$  of these intervals on the domain  $(0, 1)$  with the same combinatorial properties.

- (a) Embedding  $\mathcal{I}_1$  shows that  $H(\mathcal{R}) \geq \log 5 - \frac{1}{5} \log(1 \cdot \frac{1}{8} \cdot \frac{1}{8} \cdot \frac{1}{4} \cdot \frac{1}{4}) \approx 4.2$ .  
 (b) Embedding  $\mathcal{I}_2$  shows that  $H(\mathcal{R}) \geq \log 5 - \frac{1}{5} \log(\frac{1}{2} \cdot \frac{1}{8} \cdot \frac{1}{8} \cdot \frac{1}{8} \cdot \frac{1}{8}) \approx 4.9$ .  
 (c) Embedding  $\mathcal{I}_3$  is minimal which implies  $H(\mathcal{R}) = \log 5 - \frac{1}{5} \log(1 \cdot \frac{1}{4} \cdot \frac{1}{4} \cdot \frac{1}{4} \cdot \frac{1}{4}) \approx 3.9$ .

The above lemma relies on a value  $H(\mathcal{R})$  which intuitively is an embedding of  $\mathcal{R}$  on the interval  $[0, 1]$  that ‘optimizes’ a certain measure on the embedding. In the remainder of this subsection, we show that for any containment-compatible permutation  $\pi$ ,  $2^{A^\pi(\mathcal{R})} \leq 2^{n \cdot H(\mathcal{R})}$ . The above lemma by Cardinal et al. then shows that, for any containment-compatible permutation  $\pi$ , the  $\pi$ -ambiguity is a lower bound for  $\log e(\mathcal{R})$ . To achieve this, we first rewrite the result by Cardinal et al. so that it matches our definition of ambiguity  $A^\pi(\mathcal{R})$  more easily. Specifically, we rewrite the result to consider not embeddings on  $[0, 1]$  but embeddings on  $[0, n]$  instead.

We denote by  $E^{[0,n]}(\mathcal{R})$  the set of all embeddings of  $\mathcal{R}$  on  $[0, n]$  and write:

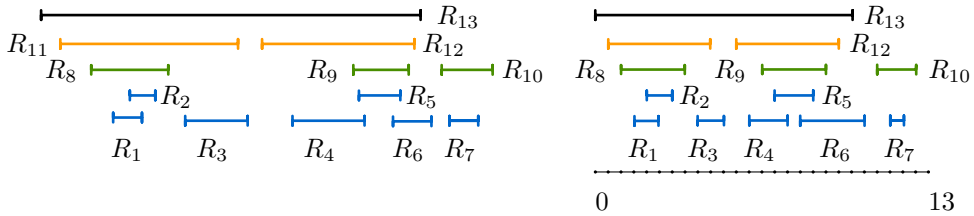
$$H(\mathcal{R}) = \log n - \min_{\mathcal{I} \in E^{[0,1]}(\mathcal{R})} \left\{ \frac{1}{n} \sum_{I_i \in \mathcal{I}} (\log n - \log(n|I_i|)) \right\} \Rightarrow$$

$$n \cdot H(\mathcal{R}) = \max_{\mathcal{I} \in E^{[0,1]}(\mathcal{R})} \left\{ \sum_{I_i \in \mathcal{I}} \log(n|I_i|) \right\} = \max_{\mathcal{I} \in E^{[0,n]}(\mathcal{R})} \left\{ \sum_{I_i \in \mathcal{I}} \log |I_i| \right\}.$$

It follows from the above definition that the value  $2^{n \cdot H(\mathcal{R})}$  is the maximum over all embeddings of  $\mathcal{R}$  on the interval  $[0, n]$ , of the product of the size of each interval. Given a containment-compatible permutation  $\pi$  we will construct a very specific embedding of  $\mathcal{R}$  on  $[0, n]$ , to show the following theorem:

**Theorem 9.2** *Let  $\pi$  be a containment-compatible permutation, then:*

$$\frac{1}{2} A^\pi(\mathcal{R}) = \sum_i \frac{1}{2} \log |\Gamma_{\pi(i)}^\pi| \leq n \cdot H(\mathcal{R}) = \max_{\mathcal{I} \in E(\mathcal{R})} \left\{ \sum_{I_i \in \mathcal{I}} \log |I_i| \right\} = O(\log |e(\mathcal{R})|).$$



**Figure 9.7** In the proof of Theorem 9.2, we construct an embedding of a set of regions  $\mathcal{R}$  as follows: first, we sort the endpoints of the regions of  $\mathcal{R}$  from left to right. Next, we map the  $k$ 'th endpoint to the point  $k/2$ . This creates an embedding of  $\mathcal{R}$  on the interval  $[0, n - 0.5]$ . Finally, we extend every interval by sliding the right vertex distance 0.5 to the right.

**Proof** Given  $\mathcal{R}$  we create a unique embedding  $\mathcal{I}^*$  of  $\mathcal{R}$  in  $[0, n]$ . Specifically, we sort the  $2n$  endpoints in  $\mathcal{R}$  from left to right and we place the  $k$ -th endpoint at the coordinate  $\frac{k}{2} \in [0, n]$ . We then translate every right endpoint an additional  $\frac{1}{2}$  to the right (increasing the width of each interval in the embedding by  $\frac{1}{2}$ ).

Observe that if  $\mathcal{R}$  is a set of open intervals then in the created embedding  $\mathcal{I}^*$  on  $[0, n]$ , a region  $I_i$  is disjoint and precedes a region  $I_j$  if and only if this is the case for  $R_i$  and  $R_j$ . Thus,  $\mathcal{I}^* \in E^{[0, n]}(\mathcal{R})$  and:

$$\sum_{I_i \in \mathcal{I}^*} \log |I_i| \leq \max_{\mathcal{I} \in E^{[0, n]}(\mathcal{R})} \left\{ \sum_{I_i \in \mathcal{I}} \log |I_i| \right\} = n \cdot H(\mathcal{R}) \leq \log |e(\mathcal{R})|.$$

Observe that if  $\pi$  is a containment-compatible permutation then the length of each interval  $I_i$  that corresponds to a region  $R_i \in \mathcal{R}$  is at least  $\frac{1}{2}|\Gamma_{\pi(i)}^\pi| + \frac{1}{2}$ . Indeed, if a region  $R_j$  is in the contact set  $\Gamma_{\pi(i)}^\pi$  then per definition of containment-compatibility the region  $R_j$  does not contain  $R_i$ , hence, the interval  $I_i$  contains at least one endpoint of  $I_j$  in its interior. It follows that the interval  $I_i \in \mathcal{I}^*$  contains at least  $|\Gamma_{\pi(i)}^\pi|$  endpoints in its interior. Thus, the interval  $I_i$  had at least width  $\frac{1}{2}|\Gamma_{\pi(i)}^\pi|$  before its right endpoint was translated further to the right and width  $\frac{1}{2}|\Gamma_{\pi(i)}^\pi| + \frac{1}{2}$  afterwards and we observe:

$$\sum_i \log \left( \frac{1}{2}|\Gamma_{\pi(i)}^\pi| + \frac{1}{2} \right) \leq \sum_{I_i \in \mathcal{I}^*} \log |I_i| \leq n \cdot H(\mathcal{R}) = O(\log |e(\mathcal{R})|).$$

For all real values  $x > 0$ ,  $\log(x)/2 \leq \log(\frac{x}{2} + \frac{1}{2})$ . The fact that for all integers  $i$ ,  $|\Gamma_{\pi(i)}^\pi| \geq 1$  then implies:

$$\frac{1}{2}A^\pi(\mathcal{R}) = \sum_i \frac{1}{2} \log |\Gamma_{\pi(i)}^\pi| \leq \sum_i \log \left( \frac{1}{2}|\Gamma_{\pi(i)}^\pi| + \frac{1}{2} \right)$$

These two inequalities, when combined, conclude the theorem. ■

The above theorem is sufficient to show our main result: in Section 9.3 we show that we can preprocess  $\mathcal{R}$ , to create a containment-compatible permutation  $\pi$  such that we can reconstruct a linked list on  $\mathcal{R}$  that implies the linear order of  $P$  in  $\Theta(A^\pi(\mathcal{R}))$  time.

Cardinal et al. [47] note that  $n \cdot H(\mathcal{R})$  and  $\log e(\mathcal{R})$  are both (asymptotically) a lower bound for the time needed to construct the sorted order of  $P$ . Hence, the algorithm of Section 9.3 shows that the  $\pi$ -ambiguity for a containment-compatible permutation  $\pi$  is a constant factor approximation of  $n \cdot H(\mathcal{R})$  and  $\log e(\mathcal{R})$ . This is interesting, since these values are  $\#P$ -hard to compute [47] and our preprocessing algorithm in Section 9.3 is hence an  $O(n \log n)$ -time constant-factor approximation algorithm for these values. For completeness, we show that the algorithm is actually a factor 4 approximation of the value  $n \cdot H(\mathcal{R})$ .

### 9.2.2 Upper bounding $n \cdot H(\mathcal{R})$

We show that for any containment-compatible permutation  $\pi$ ,  $n \cdot H(\mathcal{R}) \leq 3A^\pi(\mathcal{R})$ . To this end, we first show the following lemma:

**Lemma 9.3** *Let  $\mathcal{R}$  be a set of intervals partitioned into two sets  $X$  and  $Y$ , such that for each  $R \in X$ ,  $R' \in Y$ ,  $R$  and  $R'$  are disjoint and  $R$  is left of  $R'$ . Moreover, let:*

$$\mathcal{I}^* := \arg \max_{\mathcal{I} \in E^{[0, n]}(\mathcal{R})} \sum_{I_i \in \mathcal{I}} \log |I_i|$$

*Then the intervals in  $\mathcal{I}^*$  associated to regions in  $X$  are contained in  $[0, |X|]$  and the intervals associated to regions in  $Y$  are contained in  $[|X|, n]$ .*

**Proof** We denote by  $E^{[0, \lambda]}$  the set of all embeddings of  $\mathcal{R}$  in the interval  $[0, \lambda]$  and denote by  $nH(\mathcal{R}, \lambda)$  the ‘maximal’ embedding of  $\mathcal{R}$  on the interval  $[0, \lambda]$ . Specifically:

$$nH(\mathcal{R}, \lambda) := \max_{\mathcal{I} \in E^{[0, \lambda]}(\mathcal{R})} \sum_{I_i \in \mathcal{I}} \log |I_i|.$$

Note that  $nH(\mathcal{R}, n) = \max_{\mathcal{I} \in E^{[0, n]}(\mathcal{R})} \sum_{I_i \in \mathcal{I}} \log |I_i|$  and thus the embedding that realises  $nH(\mathcal{R}, n)$  is the embedding  $\mathcal{I}^*$  of the lemma statement.

Let  $\lambda, \mu > 0$ ,  $\mathcal{I}$  be the embedding of  $\mathcal{R}$  in  $[0, \lambda]$  that realises  $nH(\mathcal{R}, \lambda)$  and  $\mathcal{I}'$  be the embedding of  $\mathcal{R}$  in  $[0, \mu]$  that realises  $nH(\mathcal{R}, \mu)$ . Observe that we can transform  $\mathcal{I}$  into  $\mathcal{I}'$  by re-scaling  $\mathcal{I}$  where we multiply for every interval  $I_i \in \mathcal{I}$  its width by  $\frac{\mu}{\lambda}$ . It follows that for any  $\mathcal{R}$  and  $\lambda, \mu > 0$ :

$$nH(\mathcal{R}, \mu) = nH(\mathcal{R}, \lambda) + |\mathcal{R}| \log \left( \frac{\mu}{\lambda} \right) \quad \Rightarrow \quad nH(\mathcal{R}, \mu) = nH(\mathcal{R}, 1) + |\mathcal{R}| \log \mu.$$

Now we are ready to prove the lemma. Let  $X$  and  $Y$  be two sets that partition  $\mathcal{R}$  such that all regions in  $X$  are left of all regions in  $Y$ . We want to construct an embedding of  $X$  and  $Y$  in the interval  $[0, n]$  that realises  $nH(\mathcal{R}, n)$ . Note that if we embed the regions in  $X$  in  $[0, \lambda]$  then it is always optimal to embed the regions in  $Y$  in  $[\lambda, n]$ . So:

$$\begin{aligned} nH(\mathcal{R}, n) &= \max_{\lambda \in [0, n]} (nH(X, \lambda) + nH(Y, (n - \lambda))) \Rightarrow \\ nH(\mathcal{R}, n) &= nH(X, 1) + nH(Y, 1) + \max_{\lambda \in [0, n]} (|X| \log \lambda + |Y| \log (n - \lambda)). \end{aligned}$$

The term  $\lambda^{|X|}(n - \lambda)^{|Y|}$  is maximized by choosing  $\lambda = |X|$ . Thus, the embedding of  $\mathcal{R}$  that realises  $nH(\mathcal{R}, n)$  (the embedding  $\mathcal{I}^*$ ) places all intervals of  $X$  in  $[0, |X|]$  and all intervals  $Y$  in  $[|X|, n]$ . ■

Now, we are ready to prove that  $A^\pi(\mathcal{R})$  upper bounds  $n \cdot H(\mathcal{R})$ .

**Theorem 9.3** *Let  $\pi$  be any containment-compatible permutation. Then  $nH(\mathcal{R}) \leq 2A^\pi(\mathcal{R})$ .*

**Proof** We defined  $\mathcal{R}_{\leq i} = (R_1, R_2, \dots, R_i)$  as a prefix of  $\mathcal{R}$ . We fix some containment-compatible permutation  $\pi$  and prove the lemma with induction on  $i$ .

$$\text{Induction Hypothesis: } \forall j \leq i, \quad j \cdot H(\mathcal{R}_{\leq j}, j) \leq 3A^\pi(\mathcal{R}_{\leq j}).$$

The value  $1 \cdot H(\mathcal{R}_{\leq 1})$  is zero and so is  $A(\mathcal{R}_{\leq 1})$  so for the base case where  $i = 1$  the induction hypothesis is true. We continue by assuming that  $i \cdot H(\mathcal{R}_{\leq i}, i) \leq 3A^\pi(\mathcal{R}_{\leq i})$  and prove that  $(i+1) \cdot H(\mathcal{R}_{\leq i+1}, i+1) \leq 3A^\pi(\mathcal{R}_{\leq i+1})$ . We prove this through a case distinction on  $\Gamma_{\pi(i+1)}^\pi$ :

**Case 1:**  $|\Gamma_{\pi(i+1)}^\pi| = 1$ . In this case, per definition, the region  $R_{i+1}$  is disjoint from all regions in  $\mathcal{R}_{\leq i}$ . By Lemma 9.3, the embedding  $\mathcal{I}^*$  that realises  $(i+1) \cdot H(\mathcal{R}_{\leq i+1}, i+1)$  assigns to  $R_{i+1}$  width 1 and embeds  $\mathcal{R}_{\leq i}$  on the interval  $[0, i]$  (or  $[1, (i+1)]$ ) and we conclude:

$$(i+1) \cdot H(\mathcal{R}_{\leq i+1}, i+1) = i \cdot H(\mathcal{R}_{\leq i}, i)$$

Similarly, per definition  $A^\pi(\mathcal{R}_{\leq i+1}) = A^\pi(\mathcal{R}_{\leq i})$  and the inequality remains intact.

**Case 2:**  $|\Gamma_{\pi(i+1)}^\pi| > 1$ . The intervals in  $\mathcal{R}_{\leq i}$  intervals used to be optimally embedded on  $[0, i]$  and now must be embedded in  $[0, i+1]$ . Each of these intervals increases by at most a factor  $\frac{i+1}{i}$ . If we denote by  $|I_{i+1}|$  the size corresponding to  $R_{i+1}$  in the embedding that realises  $(i+1)H(\mathcal{R}_{\leq(i+1)}, i+1)$  we observe that:

$$\begin{aligned} (i+1) \cdot H(\mathcal{R}_{\leq(i+1)}, i+1) &\leq i \cdot H(\mathcal{R}_{\leq i}, i) + \log \left( \left( \frac{i+1}{i} \right)^i \right) + \log |I_{i+1}| \Rightarrow \\ (i+1) \cdot H(\mathcal{R}_{\leq(i+1)}, i+1) &\leq nH(\mathcal{R}_{\leq i}, i) + 2 + \log |I_{i+1}|. \end{aligned}$$

What remains is to estimate the size  $|I_{i+1}|$  in the embedding that realises  $(i+1) \cdot H(\mathcal{R}_{\leq(i+1)}, i+1)$ . There are  $(i - |\Gamma_{i+1}|)$  intervals disjoint from  $R_{i+1}$  so by Lemma 9.3, the size of  $|I_{i+1}|$  is at most  $|\Gamma_{i+1}|$ . It follows that:

$$nH(\mathcal{R}_{\leq(i+1)}, i+1) \leq nH(\mathcal{R}_{\leq i}, i) + 2 + \log |\Gamma_{i+1}| \leq nH(\mathcal{R}_{\leq i}, i) + 2 \log |\Gamma_{i+1}|,$$

which implies the theorem. ■

Theorems 9.2 and 9.3 now imply the main result of this section. The observation that the permutation  $\pi$  that realises the ambiguity must be containment-compatible implies the corollary.

**Theorem 9.1** *For any set of intervals  $\mathcal{R}$  in one dimension, for any containment-compatible permutation  $\pi$  on  $\mathcal{R}$ ,  $A^\pi(\mathcal{R})$  is a 4-approximation of  $nH(\mathcal{R})$  and  $\log |e(\mathcal{R})|$ .*

**Corollary 9.1** *For any set of intervals  $\mathcal{R}$  in one dimension, the ambiguity  $A(\mathcal{R})$  is a 4-approximation of  $nH(\mathcal{R})$  and  $\log |e(\mathcal{R})|$ .*

## 9.3 Sorting imprecise points in $O(A(\mathcal{R}))$ time

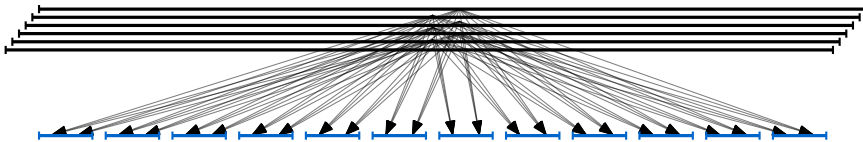
Let  $\mathcal{R} = \{R_1, R_2, \dots, R_n\}$  be a set of intervals and let  $P = \{p_1, p_2, \dots, p_n\}$  be a set of points (values) with  $p_i \in R_i$ . We show how to construct an auxiliary structure  $\Xi_{\mathcal{R}}$  on  $\mathcal{R}$  in the preprocessing phase without using  $P$  such that, in the reconstruction phase, we can construct  $\Xi_P^*$  (a linked list on  $\mathcal{R}$ , such that when we replace every region  $R_i$  with the points  $p_i$  the linked list is the sorted order of  $P$ ) in  $\Theta(A(\mathcal{R}))$  time. To achieve this, we first construct a specific containment-compatible permutation  $\pi$  of  $\mathcal{R}$ , and then show how to maintain  $\Xi_{\mathcal{R}}$  when we process the intervals in this order.

### 9.3.1 Preprocessing $\mathcal{R}$

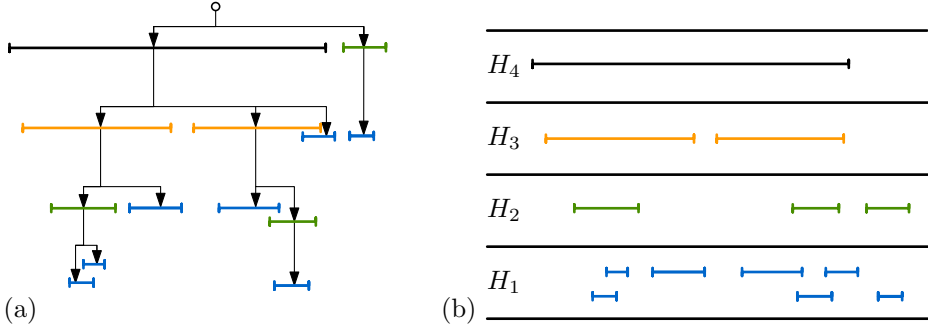
In the preprocessing phase, we show how to preprocess  $\mathcal{R}$  in  $O(n \log n)$  time to create an auxiliary structure  $\Xi_{\mathcal{R}}$  that allows us to construct the output  $\Xi_P^*$  (a linked list on the regions in  $\mathcal{R}$ , such that the sorted order of  $R_i$  matches the sorted order of  $P$ ) efficiently. Before we define our auxiliary data structure  $\Xi_{\mathcal{R}}$ , we first discuss the specific containment-compatible order  $\pi$  of  $\mathcal{R}$  that we will use.

**Containment graph.** We define a permutation  $\pi$  using what we call a containment graph. For a set of intervals  $\mathcal{R}$ , the containment graph is a directed acyclic graph where vertices correspond to regions in  $\mathcal{R}$  and where there is a directed edge from a region  $R_i$  to a region  $R_j$  if and only if  $R_i$  contains  $R_j$ . Observe that this graph is unique and can have quadratic complexity (Figure 9.8), which is why we never explicitly construct it. Instead, we construct a smaller graph  $G(\mathcal{R})$  where there is a directed path from a region  $R_i$  to a region  $R_j$  if and only if  $R_i$  contains  $R_j$ .

**A height partition.** Given the containment graph  $G(\mathcal{R})$ , we define the *height partition* as the partition of  $\mathcal{R}$  into  $m$  levels  $\mathcal{H} = H_1 \dots H_m$ ,  $H_i \subseteq \mathcal{R}$  where all  $R \in H_j$  have *height* (maximal distance from  $R$  to a leaf)  $j + 1$  in  $G(\mathcal{R})$  or equivalently: the intervals in  $H_{j+1}$  contain no intervals in  $\mathcal{R} \setminus H_{\leq j}$  (see Figure 9.9). In every set  $H_j$  in the height partition there can be no two regions  $R, R'$  where  $R$  contains  $R'$  and thus the regions have a unique left-to-right order. If for every  $H_j \in \mathcal{H}$ , we order the regions in  $H_j$  from left to right then this induces a partition  $\pi$  with the two desired properties.



**Figure 9.8** A set of intervals with a containment graph with quadratic complexity.



**Figure 9.9** (a) A set of 13 intervals and their containment graph. (b) The corresponding height partition of the intervals.

**Lemma 9.4** *For any set of intervals  $\mathcal{R}$ , we can construct  $\mathcal{H}$  in  $O(n \log n)$  time.*

**Proof** Observe that an interval  $(x_1, x_2)$  contains  $(y_1, y_2)$  if and only if  $x_1 \leq y_1$  and  $y_2 \leq x_2$ . We map every interval in  $\mathcal{R}$  to its corresponding point in  $\mathbb{R}^2$  and store all points in a dynamic two-dimensional range tree [61]. Two-dimensional range trees have linear size,  $O(\log n)$  query time and  $O(\log n)$  update time. We augment the range tree such that every node  $v$  stores a pointer to the interval  $R_i$  such that:

- (i) The point corresponding to  $R_i$  is stored in  $V$ .
- (ii) The region  $R_i \in H_j$  and all points stored in  $v$  correspond to regions in  $H_{j'}$  with  $j' \leq j$ . We call the integer  $j$  the *level* of the node  $v$ .

Specifically we start with an empty range tree and sort the intervals from narrow to wide. We then insert each interval  $R_i$  into the correct level  $H_j$  in this order and into the range tree in this order. We start with the least wide interval  $(a, b)$ . The interval  $(a, b)$  cannot contain another interval of  $\mathcal{R}$  so we store this interval in  $H_1$  and we insert it in the dynamic range tree  $(a, b)$  where the root of the tree stores a pointer to  $(a, b)$ .

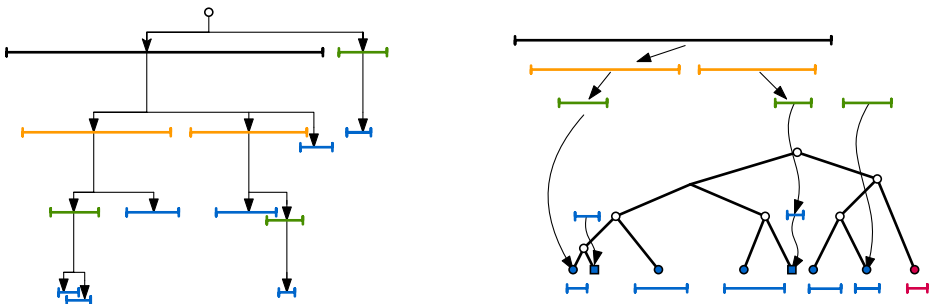
Consider the iteration where we process an interval  $(c, d)$ . Since we process intervals from narrow to wide, we must have already processed all intervals which are contained in  $(c, d)$ . We query the range tree with the following range:  $(c, \infty) \times (-\infty, d)$  and we find at most  $O(\log n)$  nodes of the range tree, such that all intervals stored at the root of those nodes are contained in  $(c, d)$ . For each of these nodes we check the level of the node. If the maximal level of each node is  $j$  then the maximal distance from  $(c, d)$  to a leaf in the graph  $G(\mathcal{R})$  contains  $j + 1$  steps and thus  $(c, d) \in H_{j+1}$ . Finally, we insert  $(c, d)$  into the range tree. The interval  $(c, d)$  is contained in at most  $O(\log n)$  nodes in the range tree. For each of these nodes, we check if the level of the node needs to be changed to  $(j + 1)$  in constant time and the update hence takes  $O(\log n)$  total time. We do the above procedure for every interval in  $\mathcal{R}$  and thus construct the height partition  $\mathcal{H}$  in  $O(n \log n)$  time. ■

We are now ready to define our containment-compatible permutation  $\pi$  and our auxiliary data structure  $\Xi_{\mathcal{R}}$  (refer to Figure 9.10 for an example):

- The containment-compatible permutation  $\pi$  is derived from the height partition  $\mathcal{H}$  of  $\mathcal{R}$ . For every level  $H_j \in \mathcal{H}$ , there can be no two regions  $R, R' \in H_j$  such that  $R$  is contained in  $R'$  or vice versa. Hence, there is a unique left-to-right order of the intervals in  $H_j$  (the left-to-right order of the left endpoints of the intervals matches the left-to-right order of the right endpoints). The permutation  $\pi$  is constructed as follows: the  $k$  regions in  $H_1$  are assigned the integer  $1 \dots k$  in their left to right order. Then the  $k'$  regions in  $H_2$  are assigned the integer  $(k+1) \dots (k+k')$  and so forth. Per definition of  $\mathcal{H}$ , this permutation  $\pi$  is containment compatible.
- The data structure  $\Xi_{\mathcal{R}}$  stores the permutation  $\pi$  as a list, and stores the following additional properties:
  - A Fibonacci tree is a binary tree where for every inner node the height of the left subtree is 1 greater than the height of the right subtree. Denote  $H_0 := \{R_i \mid |\Gamma_{\pi(i)}| = 1\}$ . We store  $H_0$  as an AVL tree, shaped as a Fibonacci tree  $T$  (this may require us to add  $O(n)$  dummy nodes to the tree).
  - We connect the leaves of the Fibonacci tree in a doubly linked list and add all regions in  $H_1 \setminus H_0$  to a queue.
  - For every region  $R \in \mathcal{H} \setminus H_0$ , we supply  $R$  with a pointer to a region  $R'$  such that  $R' \subset R$ . We call  $R'$  the anchor of  $R$ . If no such region exists, we insert a dummy point into  $T$  in the interior of  $R$  as its anchor.

Given Lemma 9.4, we immediately conclude the following theorem by noting that an AVL-tree can be constructed and rebalanced to be a Fibonacci tree in  $O(n \log n)$  time:

**Theorem 9.4** *For any set of one-dimensional intervals  $\mathcal{R}$ , we can construct the data structure  $\Xi_{\mathcal{R}}$  in  $O(n \log n)$  time.*



**Figure 9.10** The auxiliary structure  $\Xi_{\mathcal{R}}$ : we start with a height partition  $\mathcal{H}$ . All regions in  $H_0$  are stored in an AVL-tree  $T$ . Regions in  $H_1 \setminus H_0$  have a dummy point in  $T$ . The red interval is a dummy value to ensure  $T$  is also a Fibonacci tree. All regions not in the AVL-tree have a pointer to their anchor in their interior.

### 9.3.2 Reconstructing the sorted order of $P$

Let  $\mathcal{R}$  be a set of  $n$  uncertainty intervals in  $\mathbb{R}^1$ . We show a reconstruction algorithm that  $\mathcal{R}$  with corresponding point set  $P$  constructs  $\Xi_P^*$  (a linked list on  $\mathcal{R}$ , such that when we replace every region  $R_i$  with the points  $p_i$  the linked list is the sorted order of  $P$ ). We show that given  $\Xi_{\mathcal{R}}$ , this reconstruction algorithm runs in  $O(A(\mathcal{R}))$  time which is uncertainty region optimal.

We first give a high-level overview of our approach: recall that the data structure  $\Xi_{\mathcal{R}}$  stores an AVL-tree on the regions in  $H_0$  in their left-to-right order where the leaves of the tree are in a linked list. The value  $A(\mathcal{R}) = 0$  if and only if regions in  $\mathcal{R}$  are pairwise disjoint. In this case, the sorted order of  $P$  is equal to the sorted order of  $\mathcal{R}$  and we output the linked list immediately. Otherwise, we need to retrieve at least one point from  $P$ . We iteratively retrieve points from  $P$  and add them to the AVL-tree whilst maintaining a linked list of the leaves. The output  $\Xi_P^*$  is the final linked list on the leaves of the AVL-tree after all regions have been inserted.

We denote by  $\pi$  the containment-compatible permutation that was constructed in the preprocessing phase. Observe that all regions  $R_i \in \mathcal{R}$  with  $|\Gamma_{\pi(i)}^\pi| = 1$  are in  $H_1 \in \mathcal{H}$  and thus already inserted into the AVL-tree. For all regions  $R_j$  for which  $|\Gamma_{\pi(j)}^\pi| > 1$ , we insert the point  $p_j$  into the correct location of the AVL-tree in  $O(\log |\Gamma_{\pi(j)}^\pi|)$  time. As we insert points into the AVL-tree, we want to make sure that the tree remains a balanced binary tree. Naively, rebalancing a binary tree after an insertion can take  $O(\log n)$  time (and, if  $\log n > \log |\Gamma_{\pi(j)}^\pi|$ , we may not be able to computationally afford this rebalancing). Melhorn and Tsakalidis [160] note that the tree balance of an insertion-only AVL-tree can be maintained in amortized constant time. They show this via a charging scheme where they charge the cost of rebalancing the tree to values that were inserted prior. We cannot immediately use the result by Melhorn and Tsakalidis, as their analysis assumes that you start with an empty tree whereas our tree can contain up to  $O(n)$  elements that cannot be charged any computational time. Therefore, we adapt their argument to show the following:

**Lemma 9.5** *Let  $T$  be an AVL-tree where each inner node has two subtrees with a height difference of 1. We can iteratively insert  $k$  values in between arbitrary leaves of  $T$  such that we maintain the balance of  $T$  in  $O(k)$  total time.*

**Proof** An AVL tree is a balanced binary tree where each inner node  $v \in T$  has a balance constant  $b(v) \in \{-1, 0, +1\}$ . If  $b(v) = -1$ , the height of the left subtree of  $v$  is 1 greater than the height of the right subtree. If  $b(v) = 0$  then both subtrees of  $v$  have equal height and otherwise  $b(v) = +1$ . Let  $x$  be a value that we want to insert into  $T$  and  $v$  be the leaf that would become the parent of  $x$ . An insertion replaces  $v$  with a decision node that is the parent of  $x$  and  $v$ . After the insertion, the rebalancing scheme of an AVL-tree walks towards root of the AVL-tree and iteratively adjust the balance constants in each node (possibly triggering a rotation).

Specifically, Melhorn and Tsakalidis [160] show that after an insertion this rebalancing procedure travels a path from  $v$  to  $\bar{v}$  where  $\bar{v}$  is the lowest ancestor of  $v$  such that  $b(\bar{v}) \in \{-1, +1\}$ . There, the rebalancing triggers either a rotation or the insert guarantees that the left and right subtree of  $\bar{v}$  have equal height. In both cases the rebalancing terminates. In our tree  $T$ , every inner node stores two children where the difference in their height is 1. Hence originally, all balance constants were either  $-1$  or  $1$  (since in the preprocessing phase we constructed a Fibonacci tree, all constants are  $-1$ ). This immediately implies that whenever we insert a new value in  $T$ , it may change arbitrarily many 0-nodes into  $+1$  or  $-1$ . However, for each of these nodes, there was a unique insert that created the 0 value which can be charged for the time needed to inspect the node. Thus, as in [160], we can maintain the balance of our binary  $T$  in amortized constant time per insertion. ■

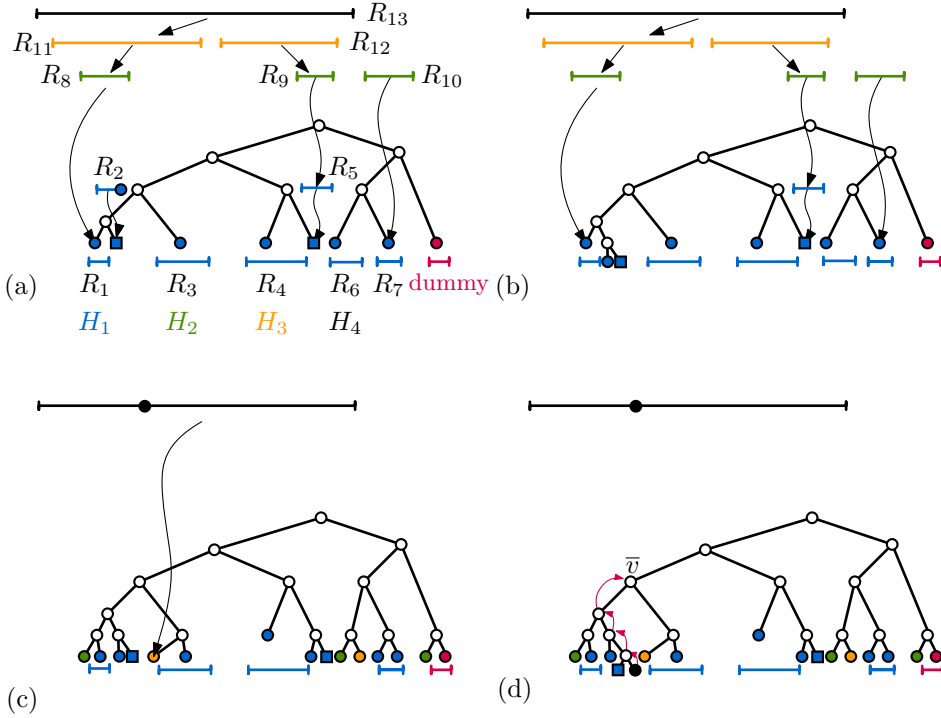
The above lemma provides us with all the tools to prove the following theorem:

**Theorem 9.5** *Let  $\mathcal{R}$  be a set of  $n$  uncertainty intervals and  $P$  be its underlying point set. We can preprocess  $\mathcal{R}$  in  $O(n \log n)$  time, such that given  $P$  we can construct  $\Xi_P^*$  (a linked list on  $\mathcal{R}$ , such that when we replace every region  $R_i$  with the points  $p_i$  the linked list is the sorted order of  $P$ ) in  $\Theta(A(\mathcal{R}))$  time.*

**Proof** Given  $\mathcal{R}$ , we obtain the auxiliary data structure  $\Xi$  in  $O(n \log n)$  time via Theorem 9.4 and the containment-compatible permutation  $\pi$  can correspond to the height partition of  $\mathcal{R}$ . We show that, given indirect access to  $P$  where for every region  $R_i$  we can retrieve the point  $p_i$ , we can construct  $\Xi^*$  in  $O(A^\pi(\mathcal{R})) = \sum_i O(\log |\Gamma_{\pi(i)}^\pi|)$  time. By Theorem 9.1 and Corollary 9.1, this analysis shows that we construct  $\Xi^*$  in  $\Theta(A(\mathcal{R}))$  time, which is uncertainty-region optimal.

Let  $\mathcal{H}$  be the height partition of  $\mathcal{R}$ . The auxiliary data structure  $\Xi$  contains an AVL-tree  $T$  on a subset of  $H_0 := \{R_i \in \mathcal{R} \mid |\Gamma_{\pi(i)}^\pi| = 1\}$ . For all regions in  $\mathcal{R} \setminus H_0$ , we may spend at least constant time and we have access to these regions in their order along  $\pi$  and we will insert their corresponding points into  $T$  in this order.

The remainder of our algorithm is illustrated by Figure 9.11. For every  $R_i \in \mathcal{R} \setminus H_0$  we do the following: we retrieve  $p_i$ . Then, we do a point location for  $p_i$  amongst all intervals and points stored in  $T$ . Note that per definition, there are at most  $|\Gamma_{\pi(i)}^\pi|$  values in  $T$  that are contained in the region  $R_i$  (and  $p_i$  must lie beside or in-between any of these values). We traverse the anchor of  $R_i$  to obtain a leaf in  $T$  that lies in the interior of  $R_i$  in constant time. From this value, we perform a binary search for the location of  $p_i$ . This binary search takes  $O(\log |\Gamma_{\pi(i)}^\pi|)$  time by traversing the binary tree and the corresponding leaf pointers. If we discover that  $p_i$  is contained in some region  $R_j \in T$ , we also retrieve  $p_j$  and charge it to the cost of retrieving  $p_i$ . Note that this can occur at most once, as regions in  $T$  are per definition pairwise disjoint from all other regions in  $T$ . We insert  $p_i$  into  $T$  and connect it into the linked list on the leaves of  $T$  in constant time. By Lemma 9.5, we rebalance  $T$  such that the total time spent rebalancing the tree is  $O(A(\mathcal{R}))$ . It follows that for each region  $R_i$ , we spend  $O(\log |\Gamma_{\pi(i)}^\pi|)$  and thus we construct the output  $\Xi_P^*$  in  $\Theta(A(\mathcal{R}))$  total time. ■



**Figure 9.11** An illustration of our reconstruction algorithm. We indexed the regions such that the permutation  $\pi$  induced by the height partition is the identity. (a) Before the start of the reconstruction phase, the regions  $H_0 = \{R_1, R_3, R_4, R_6, R_7\}$  are in the AVL tree  $T$ . All other regions in  $H_1$  have their corresponding dummy node. Since we follow the order  $\pi$ , we start with  $R_2$  and retrieve  $p_2$  and perform a point location. (b) We insert  $p_2$  into the tree  $T$ . The location of  $p_2$  happens to be next to the dummy node that was the anchor of  $p_1$ . We connect  $p_2$  into the linked list on the (non-dummy) leaves. Note that we no longer require that  $T$  is a Fibonacci tree. (c) After a few interactions only  $R_{13}$  remains. The region  $R_{13}$  had a pointer to  $R_{11}$  with  $R_{11} \subset R_{13}$ . Since  $R_{11}$  must precede  $R_{13}$  in the permutation  $\pi$ , the point  $p_{11}$  is already inserted into  $T$  and it must lie in the interior of  $R_{13}$ . We follow the anchor pointer to  $p_{11}$  and start our binary search for the location of  $p_{13}$  from there. (d) After inserting  $p_{13}$ , we need to rebalance the tree. We recursively step to our parent node  $v$ , and set the balance constant  $b(v)$  from 0 to +1 until we reach the node  $\bar{v}$ . Here, we identify that  $b(\bar{v})$  was  $-1$  (indeed, the left subtree had height 3 and the right subtree had height 2). Now, the difference in height between these two trees is greater than 1 and we rotate around  $b(\bar{v})$ .

### 9.3.3 Instance optimality for point retrievals

If the cost  $C$  of point retrieval is constant then the reconstruction algorithm of Section 9.3.2 is uncertainty-region optimal. However, if the cost  $C$  is not constant then, without any further analysis, it may be that the reconstruction algorithm retrieves more points than necessary. In this section we show that this does not occur. In fact, we show that the number of retrievals performed by our algorithm is instance optimal. The key argument is that for instance optimality we only may consider algorithms  $A$  that are correct. An algorithm that is allowed to make mistakes may (accidentally) report the output  $\Xi_P^*$  with no retrievals. Indeed, consider an algorithm  $A'$  that given  $\mathcal{R}$  outputs a linked list on  $\mathcal{R}$  where the order of the linked list orders  $\mathcal{R}$  by size. It may be, that for a specific instance  $(\mathcal{R}, P)$  the algorithm  $A$  reconstructs  $\Xi_P^*$  in  $O(1)$  time (this is achieved by constructing the linked list on  $\mathcal{R}$  in the preprocessing phase, and by subsequently referencing the list). However, clearly, there are many instances where this algorithm is incorrect. We claim that an algorithm can only be correct whenever it constructs an appropriate *witness* of the output. More formally, let  $A$  be an algorithm that sorts an imprecise point set. Denote, for a given input  $(\mathcal{R}, P)$ , by  $P'$  the set of points retrieved by  $A$  in the reconstruction phase and denote by  $\mathcal{R}'$  the maximal set of regions where points corresponding to regions in  $\mathcal{R}'$  are not in  $P'$ . If the algorithm  $A$  is always correct then, for every  $(\mathcal{R}, P)$ , the set  $(P' \cup \mathcal{R}')$  is a set of pairwise disjoint elements (if this is not the case, then there trivially exist at least two possible linear extensions of the partial order induced by  $(P' \cup \mathcal{R}')$  and the algorithm  $A$  has no way of distinguishing between these two. Hence, the algorithm could be wrong). This observation leads to the following theorem:

**Theorem 9.6** *For any pair  $(\mathcal{R}, P)$ , there exists no correct algorithm  $A$  that constructs  $\Xi_P^*$  with fewer than 3 times the number of retrievals performed by the algorithm in Section 9.3.2.*

**Proof** Consider a preprocessing permutation  $\pi$  and a region  $R_i$  that is processed by the algorithm. We traverse the AVL tree to perform a point location for  $p_i$  amongst: (1) all points that we have retrieved and (2) all regions  $R_j$  for which  $\pi(j) < \pi(i)$ . Specifically, if for all such regions and points  $R_i$  is disjoint of these geometric objects, then we do not retrieve  $p_i$ . If  $R_i$  does overlap with at least one such region or point then we retrieve  $p_i$ . In addition, if  $p_i$  is contained within any such region  $R_j$  then we also retrieve  $p_j$ . We show that there can be no algorithm that creates a ‘witness set’ that performs fewer retrievals than our retrieval strategy.

Indeed, suppose that there exists a point  $p_j \in R_i$  which we already have retrieved. Then we do not know the relative order between  $p_i$  and  $p_j$  and thus, we must retrieve  $p_i$ . Similarly, if  $p_i \in R_j$  for some region  $R_j$  then any algorithm must retrieve  $p_j$ . Suppose otherwise that  $R_i$  overlaps with a region  $R_j$  and that  $\pi(j)$  is the largest number for which  $\pi(j) < \pi(i)$ . Any algorithm must retrieve either  $p_i$  or  $p_j$  to determine their relative order. We charge the time needed for this retrieval to  $p_j$ . Observe that because the permutation  $\pi$  is containment-compatible, there can be at most two regions,  $R_i$  and  $R_k$  that charge  $p_j$  and thus there cannot exist an algorithm that performs fewer than three times the number of retrievals than our reconstruction algorithm. ■

## 9.4 Concluding remarks

In this chapter we studied how to efficiently sort an imprecise point set. We showed how to preprocess a set  $\mathcal{R}$  of  $n$  one-dimensional uncertainty intervals to create an auxiliary data structure  $\Xi_{\mathcal{R}}$  in  $O(n \log n)$  time. Given access to some underlying point set  $P$ , we can use  $\Xi_{\mathcal{R}}$  to construct  $\Xi_P^*$  (a linked list on  $\mathcal{R}$ , such that when we replace every region  $R_i$  with the points  $p_i$  the linked list is the sorted order of  $P$ ) in  $\Theta(A(\mathcal{R}))$  time. In some cases, for example when the algorithmic input  $\mathcal{R}$  consists of pairwise disjoint intervals, the value  $\Theta(A(\mathcal{R})) = \Theta(1)$ . In this case, the required output of the preprocessing phase is the structure  $\Xi_P^*$  and this has a worst-case lower bound of  $\Omega(n \log n)$ . It follows from this observation that our preprocessing running time is worst-case optimal. It is unclear how (in the preprocessing phase) notions of better than worst-case optimality should be defined. One possible avenue for improvement may be to make the running time of the preprocessing phase sensitive to how sorted the original input  $\mathcal{R}$  is. Specifically, the running time of the preprocessing phase is  $O(n \log n)$  due to two operations: sorting the endpoints of  $\mathcal{R}$  and constructing a range tree on  $\mathcal{R}$ . If the input  $\mathcal{R}$  is given together with the sorted order on the left and right endpoints of  $\mathcal{R}$  then both of these operations can be performed in linear time. Hence, for better than worst-case preprocessing performance it may be possible to parametrize the running time by how ‘sorted’ the input  $\mathcal{R}$  is.

Our reconstruction algorithm runs in  $\Theta(A(\mathcal{R}))$  time, under the assumption that the cost of retrieving for a region  $R_i$  the corresponding point  $p_i$  is constant. In addition, we showed that the number of retrievals performed by our reconstruction algorithm is instance optimal. These two observations imply that our reconstruction algorithm is uncertainty region optimal. Note that for every region  $R_i$ , where we retrieve the point  $p_i$ , we perform at most  $O(\log n)$  additional instructions. Thus, if the cost  $C$  for retrieving a point is  $\Omega(\log n)$ , then by Theorem 9.6 our reconstruction algorithm is instance optimal. If the cost  $C$  for retrieving a point is  $o(\log n)$ , our algorithm is uncertainty-region optimal and Theorem 8.1 guarantees that there can exist no instance optimal algorithm. Hence, the result presented here can be considered ‘as good as it gets’.

## Chapter Ten

# Imprecise staircases

In the preprocessing model for uncertain data we are given a set of regions  $\mathcal{R}$  which model the uncertainty associated with an unknown set of points  $P$ . In this model there are two phases: a preprocessing phase, in which we have access only to  $\mathcal{R}$ , followed by a reconstruction phase, in which we have access to points in  $P$  at a certain retrieval cost  $C$  per point. We study the following algorithmic question: how fast can we construct the Pareto front of  $P$  in the preprocessing model?

We show that if  $\mathcal{R}$  is a set of pairwise disjoint axis-aligned rectangles, then we can preprocess  $\mathcal{R}$  to reconstruct the Pareto front of  $P$  efficiently. We rely on the new definition of uncertainty-region optimality presented in Chapter 8, which falls on the spectrum between worst-case optimality and instance optimality. Our results in this chapter are worst-case optimal in the preprocessing phase; in the reconstruction phase, our results are uncertainty-region optimal with respect to real RAM instructions, and instance optimal with respect to point retrievals.

## 10.1 Introduction

In the previous chapter we studied the construction of the sorted order of a set of 1-dimensional imprecise points, in the preprocessing model with indirect representation. In this chapter, we study imprecise points in two dimensions. Specifically, we study a set  $\mathcal{R} = (R_1, R_2, \dots, R_n)$  of pairwise disjoint planar uncertainty rectangles where  $P = (p_1, p_2, \dots, p_n)$  is the corresponding ‘true’ planar point set. Throughout this chapter, we assume that the cost  $C$  of retrieving for a region  $R_i$  the point  $p_i$  is some fixed number of RAM instructions (e.g.  $C = \Theta(\log n)$ ). Intuitively, the Pareto front of a set of points is the set of maximal points. Formally, we say that a point  $p$  dominates a point  $p'$  if the  $x$  and  $y$  coordinates of  $p$  are greater than the respective coordinates of  $p'$ . Given a planar point set  $P$ , there exists a connected area of points  $p' \in \mathbb{R}^2$  that are dominated by a point in  $P$ . The Pareto front is the boundary of this area, which is a subset of  $P$  connected through a staircase (refer to Figure 10.1). This staircase can be seen as a pointer structure on  $P$ . We study reconstruction of the Pareto front of  $P$  in the preprocessing model with indirect representations. Here, the goal in the reconstruction phase is to construct some pointer structure  $\Xi_P^*$  on  $\mathcal{R}$  such that if we replace every region  $R_i \in \mathcal{R}$  with the point  $p_i \in R_i$ , we obtain the Pareto front of  $P$ .

**Related work in the preprocessing model.** The Pareto front is a pointer structure on a planar point set that represents some form of *proximity information* (points that are connected on the staircase are adjacent in the top left to bottom right traversal of the Pareto front). There have been several publications about constructing proximity pointer structures in the preprocessing model. Löffler and Snoeyink [149] study constructing a Gabriel graph of pairwise disjoint unit-size uncertainty disks, in the classical preprocessing model without implicit representation. After  $O(n \log n)$  time preprocessing, they reconstruct a Gabriel graph in  $O(Cn)$  time. When we explicitly require that the output is a Gabriel graph on the point set  $P$ , this result is optimal as we need to retrieve every point in  $P$  to insert into the Gabriel Graph in  $\Omega(nC)$  time. A Gabriel graph, quadtree, Euclidean minimum spanning tree and Delaunay triangulation are linear-time equivalent [149] so their result is worst-case optimal for many problems (when not using indirect representations). Löffler and Mulzer [147] study constructing an onion layer decomposition in the same setting. With  $O(n \log n)$  preprocessing, they reconstruct in  $O(n \log k + nC)$  time where  $k$  is the output size.

Bruce et al. [35] study the construction of the Pareto front for pairwise disjoint axis-aligned uncertainty rectangles in an I/O sensitive setting. In this setting, they assume that the cost  $C$  needed to retrieve a point is very high (they assume it dominates polynomial RAM time which essentially makes all RAM operations free). They present a retrieval strategy that iteratively selects a region  $R_i$  for which to retrieve  $p_i$ . Selecting a suitable region  $R_i$  requires RAM operations that are neither described in detail nor accounted for in the running time. In contrast, our results in this chapter do count RAM instructions and treat the fixed cost  $C$  as an unknown number of RAM instructions and show it as an additional parameter in the running time.

In Chapter 8 we introduced the concept of indirect representations  $\Xi_P^*$  which are not bound to the  $\Omega(Cn)$  worst-case lower bound for the reconstruction phase. In addition, we showed the definition of the uncertainty-region lower bound which is a nontrivial lower bound for any fixed set of regions  $\mathcal{R}$ . In Chapter 9 we showed that if  $\mathcal{R}$  is a set of overlapping uncertainty intervals in one dimensions, then after  $\Theta(n \log n)$  preprocessing we can create an indirect representation of the sorted order of  $P$  in uncertainty-region optimal time. Now, we try to extend this paradigm to higher dimensions with the added assumption that, like most results in the preprocessing model, the uncertainty regions are pairwise disjoint.

**Contribution.** The contribution of this chapter is twofold. First, we show a non-trivial lower bound for reconstructing the Pareto front. We briefly recall the definitions of uncertainty region lower bounds and instance lower bounds. An uncertainty region lower bound is a function  $f$  that for any set of regions  $\mathcal{R}$ , gives a value  $f(\mathcal{R})$  such that there can be no reconstruction algorithm that constructs  $\Xi_P^*$  faster than  $\Omega(f(\mathcal{R}))$  time for *every* point set  $P$  that respects  $\mathcal{R}$ . An instance lower bound is a function  $g$  that for any pair  $(\mathcal{R}, P)$  gives a value  $g(\mathcal{R}, P)$  such that there can be no reconstruction algorithm that constructs  $\Xi_P^*$  faster than  $\Omega(g(\mathcal{R}, P))$  time. In Section 10.3, we create what we call the *Pareto cost function* which we denote by  $CP(\mathcal{R}, P)$ . We prove that for any pair  $(\mathcal{R}, P)$  where  $P$  respects  $\mathcal{R}$ , the value  $CP(\mathcal{R}, P)$  is an uncertainty region lower bound for constructing the Pareto front  $\Xi_P^*$ : In addition, we show that for any pair  $(\mathcal{R}, P)$  the value  $CP(\mathcal{R}, P)$  implies an instance lower bound for the number of disk retrievals that need to be performed (we lower bound the number of times that for a region  $R_i \in \mathcal{R}$ , we need to retrieve the corresponding point  $p_i$ ).

Next, in Sections 10.5 and 10.6, we show how to preprocess  $\mathcal{R}$  in  $O(n \log n)$  time such that given  $P$ , we can reconstruct  $\Xi_P^*$  in  $\Theta(CP(\mathcal{R}, P))$  time. This immediately implies that our results are uncertainty region optimal. Our additional analysis shows that the number of disk retrievals performed by our reconstruction algorithm is instance optimal. Finally, we note that our reconstruction algorithm is at most a log-factor removed from instance optimal. That is for any pair  $(\mathcal{R}, P)$ , every reconstruction algorithm that creates  $\Xi_P^*$  takes at least  $\Omega(\frac{1}{\log n} CP(\mathcal{R}, P))$  time.



**Figure 10.1** (a) A collection of five points. The red points are dominated by at least one green point. (b) The set of green points  $G$  creates an area of all points dominated by points in  $G$ . (c) The Pareto front is the boundary of this area, which is the set  $G$  connected by a staircase of edges.

## 10.2 Geometric preliminaries

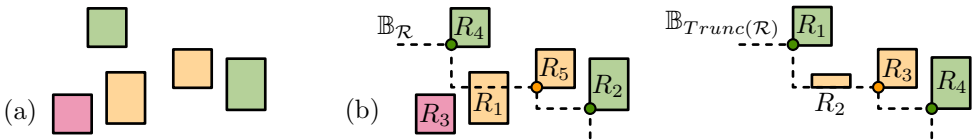
This section is dedicated to defining the concepts used throughout the chapter and to proving useful properties of dominating geometric objects. In Section 10.4 We present additional preliminaries which are required only for the reconstruction algorithm. Throughout the chapter, we denote by  $\mathcal{R}^\circ$  and  $\mathcal{R}^\times$  the original and truncated regions respectively (which we define later). When the set is clear from context, we drop the superscript. Let  $\mathcal{R} = (R_1, R_2, \dots, R_n)$  be a sequence of  $n$  pairwise disjoint closed axis-aligned uncertainty rectangles, with underlying point set  $P$ . For ease of exposition, we assume  $\mathcal{R}$  and  $P$  lie in general position (no points or region vertices share a coordinate). We denote by  $[R_i, R_j] := (R_i, R_{i+1}, \dots, R_j)$  a subsequence of  $j - i + 1$  regions and similarly by  $[p_i, p_j] = (p_i, p_{i+1}, \dots, p_j)$  a subsequence of points. For brevity, with slight abuse of notation, we may refer to points as degenerate rectangles; hence any set  $\mathcal{R}$  may contain points. Whenever we place points on a vertex, we mean placing it arbitrarily close to that vertex. If for two regions  $R_i, R_j$ ,  $i < j$  then we say that  $R_i$  *precedes*  $R_j$  and  $R_j$  *succeeds*  $R_i$ .

We defined in the introduction that a point  $p$  dominates a point  $p'$  if  $p$  has greater  $x$  and  $y$  coordinate. The Pareto front of a point set  $P$  is the boundary of the area of  $\mathbb{R}^2$  that is dominated by points in  $P$ . We say that a point  $p$  (Pareto) *dominates* a rectangle  $R$ , if  $p$  dominates its top right vertex. For any region or point  $R$ , we define its *horizontal halfslab* as the union of all horizontal halflines that are directed leftward, whose apex lies in or on  $R$ . We define the *vertical halfslab* symmetrically using downward vertical halflines. Given a set  $\mathcal{R}$  without knowledge of  $P$ , we say a region  $R_i \in \mathcal{R}$  is (Figure 10.2(a)):

- a *negative* region if for all choices of  $P$ ,  $p_i$  is not part of the Pareto front of  $P$ ;
- a *positive* region if for all choices of  $P$ ,  $p_i$  is part of the Pareto front; or
- a *potential* region if it is neither positive nor negative.

Given the above definition, we observe the following:

**Lemma 10.1** *A region  $R_i \in \mathcal{R}$  is negative if and only if  $\exists R_j \in \mathcal{R}$  such that the top right vertex of  $R_i$  is dominated by the bottom left vertex of  $R_j$ . A non-negative region  $R_i$  is positive if and only if  $\nexists R_k \in \mathcal{R}$  such that  $R_i$  intersects either halfslab of  $R_k$ .*



**Figure 10.2** (a) Uncertainty regions: green regions are positive, red are negative and yellow are potential. (b) Uncertainty regions before and after truncation, note that we re-indexed and resized a region.

**Proof** Let  $R_i$  and  $R_j$  be two axis-aligned rectangular uncertainty regions where the top right vertex of  $R_i$  is dominated by the bottom left vertex of  $R_j$ . We denote by  $p_i$  the point corresponding to  $R_i$ . All choices of  $p_i \in R_i$  are dominated by the top right vertex of  $R_i$ . Similarly, all choices of  $p_j \in R_j$  dominate the bottom left vertex of  $R_j$ . Hence via transitivity,  $p_j$  always dominates  $p_i$  which implies that  $R_i$  is a negative region. If there is no region whose bottom left vertex dominates the top right vertex of  $R_i$ , then  $p_i$  appears on the Pareto front of  $P$  and  $R_i$  is thus positive.

If  $R_i$  is non-negative, and there exists a region  $R_k$  that intersects  $R_i$  in its horizontal or vertical halfslab then  $R_i$  cannot be positive since if  $p_k$  is placed on the top right vertex of  $R_k$  and  $p_i$  on the bottom left vertex,  $p_k$  must dominate  $p_i$ .

Suppose that  $R_i$  is not positive and not negative. Then per definition there exists a point placement of  $p_i$ , and another true point  $p_l$ , such that  $p_l$  dominates  $p_i$ . In this case,  $p_l$  also dominates the bottom left vertex of  $R_i$ , yet the uncertainty region  $R_l$  cannot be entirely contained in the quadrant that dominates the top right vertex of  $R_i$ , else  $R_i$  is negative. Hence  $R_l$  must have a halfslab that intersects  $R_i$  which proves the lemma. ■

Evans and Sember [87] and Nagai et al. [166] study convex hulls and Pareto fronts of imprecise points. They note that for a set of pairwise disjoint convex regions  $\mathcal{R}$  there is a connected area of negative points. They call this area the *guaranteed dominated region*. We refer to the boundary of the guaranteed dominated region as the *guaranteed boundary*  $\mathbb{B}_{\mathcal{R}}$ . We note that for Pareto fronts the guaranteed boundary is the Pareto front of the bottom left vertices in  $\mathcal{R}$ .

**Lemma 10.2** *Let  $\mathcal{R}$  be a set of pairwise disjoint rectangles, then the area of  $R_i$  that lies above  $\mathbb{B}_{\mathcal{R}}$  is either empty or a rectangle.*

**Proof** Per definition, negative regions have a top right vertex that lies below  $\mathbb{B}_{\mathcal{R}}$  and their area above  $\mathbb{B}_{\mathcal{R}}$  is therefore empty. Non-negative regions have a top right vertex that lies above  $\mathbb{B}_{\mathcal{R}}$  and therefore a non-empty area above  $\mathbb{B}_{\mathcal{R}}$ . Their bottom left vertex lies either on  $\mathbb{B}_{\mathcal{R}}$ , or below  $\mathbb{B}_{\mathcal{R}}$  (since  $\mathbb{B}_{\mathcal{R}}$  is the Pareto front of all bottom left vertices). Hence the closure of each uncertainty region intersects  $\mathbb{B}_{\mathcal{R}}$ . The intersection between a connected staircase and an axis-aligned rectangular region is always a connected staircase.

Next, we make the intuitive distinction between the top vertices of a staircase and the bottom vertices. Every ‘top’ vertex of  $\mathbb{B}_{\mathcal{R}}$  corresponds to a bottom left vertex of a region in  $\mathcal{R}$ . Each region  $R_i \in \mathcal{R}$  cannot contain such a top vertex in its interior since regions are pairwise disjoint. Thus, if a region  $R_i$  contains a vertex of  $\mathbb{B}_{\mathcal{R}}$  it can contain only ‘bottom’ vertices. The fact that  $\mathbb{B}_{\mathcal{R}}$  is a staircase and that every region  $R_i \in \mathcal{R}$  cannot contain top vertices implies that every region can contain at most one vertex of the staircase in its interior. It follows that for non-negative regions, their area above  $\mathbb{B}_{\mathcal{R}}$  is rectangle. ■

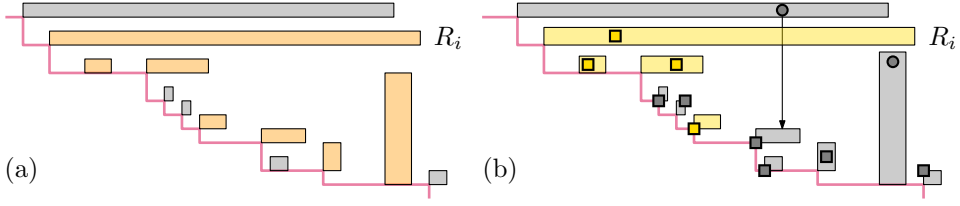
Intuitively, discovering the exact location of a point *below*  $\mathbb{B}_{\mathcal{R}}$  does not provide additional useful information, only discovering *if* a point lies below  $\mathbb{B}_{\mathcal{R}}$  does. We formalise this intuition by defining a procedure *Trunc*. Given an *original* set  $\mathcal{R}^{\circ}$  of  $m$  pairwise disjoint axis-aligned rectangles, *Trunc*( $\mathcal{R}^{\circ}$ ) returns a *truncated set*  $\mathcal{R}^{\times}$  of  $n$  rectangles. Specifically (Figure 10.2(b)), each negative region in  $\mathcal{R}^{\circ}$  gets removed and each potential region  $R_i$  gets replaced by the area of  $R_i$  above  $\mathbb{B}_{\mathcal{R}^{\circ}}$ . By Lemma 10.2 this results in a rectangular area. The resulting set *Trunc*( $\mathcal{R}^{\circ}$ ) consists of disjoint rectangles that touch  $\mathbb{B}_{\mathcal{R}^{\circ}}$ . Their intersections with  $\mathbb{B}_{\mathcal{R}^{\circ}}$  induce a well-defined order and *Trunc* re-indexes the remaining regions according to the top left to bottom right ordering of their bottom left vertices. We obtain a set  $\mathcal{R}^{\times} = (R_1, R_2, \dots, R_n) = \text{Trunc}(\mathcal{R}^{\circ})$  with  $n \leq m$ . Observe that  $\mathbb{B}_{\mathcal{R}^{\circ}} = \mathbb{B}_{\mathcal{R}^{\times}}$ . We say  $\mathcal{R}^{\times}$  is a *truncated set* if it is the result of a truncation of some set  $\mathcal{R}^{\circ}$ .

**Dependency graphs.** Given a truncated set  $\mathcal{R} = \mathcal{R}^{\times}$ , we define a (*directed*) *dependency graph* denoted by  $G(\mathcal{R})$  as follows. The nodes of the graph correspond to the regions in  $\mathcal{R}$ . The directed edges are called either horizontal or vertical arrows. A region  $R_i$  has a *vertical arrow* to  $R_j$  if  $R_j$  succeeds  $R_i$  and is vertically visible from  $R_i$  (that is, there exists a vertical segment connecting  $R_i$  and  $R_j$  that does not intersect any other region in  $\mathcal{R}$ ). A region  $R_i$  has a *horizontal arrow* to  $R_j$  if  $R_j$  precedes  $R_i$  and is horizontally visible from  $R_i$ . If  $\mathcal{R}$  is a truncated set then any region in  $\mathcal{R}$  which is a single point has no outgoing arrows, since after truncation the halfslabs of  $p$  do not intersect the interior of any rectangle in  $\mathcal{R}$ . We note the following property of  $G(\mathcal{R})$ :

**Lemma 10.3** *Let  $R_i \in \mathcal{R}$  such that  $R_i$  is a source in  $G(\mathcal{R})$ . There can be no dependency arrows between any two regions  $R_k, R_l$  with  $k < i < l$ .*

**Proof** Consider such regions  $R_k, R_i$  and  $R_l$ . Per the ordering of  $\mathcal{R}$ , the bottom left vertex of  $R_k$  lies left and above the bottom left vertex of  $R_i$ . Per definition,  $R_k$  can only have a vertical arrow to  $R_l$ . The region  $R_k$  has a vertical arrow to  $R_l$  only if its bottom facet lies above  $R_l$ . However, then either its bottom facet intersects  $R_i$  (contradicting the assumption that the regions are pairwise disjoint) or it lies above  $R_i$  (contradicting the assumption that  $R_i$  is a source node in  $G(\mathcal{R})$ ). The argument for arrows from  $R_l$  to  $R_k$  is symmetrical. ■

**Corollary 10.1** *Let  $\mathcal{R}$  be a truncated set and let  $R_i$  and  $R_j$  be source nodes in  $G(\mathcal{R})$ . There is no region in  $\mathcal{R} \setminus [R_i, R_j]$  that has a directed path in  $G(\mathcal{R})$  to any region in  $[R_i, R_j]$ .*



**Figure 10.3** (a) A region  $R_i$  and the set  $V_i$  in orange. (b) For a given set of points  $P$ , the set  $V_i(P)$  is shown in yellow. The regions in  $V_i(P)$  are not dominated by  $p_i$  but rather by a point preceding it.

**The Pareto cost function.** We show that for any set  $\mathcal{R}^*$  we can construct the Pareto front of the underlying point set using only  $\mathcal{R} = \mathcal{R}^* = \text{Trunc}(\mathcal{R}^*)$ . To show that we construct  $\Xi_P^*$  in uncertainty-region optimal time we define the *Pareto cost function* denoted by  $\text{CP}(\mathcal{R}, P)$ . In Section 10.3 we show that  $\text{CP}(\mathcal{R}, P)$  is an uncertainty-region lower bound for constructing  $\Xi_P^*$ . In the remaining sections, we show that this lower bound can be matched.

Before we formally define the Pareto cost function, we first discuss the intuition of its elements. Let  $\mathcal{R} = \mathcal{R}^*$  be a truncated set. Given  $(\mathcal{R}, P)$ , there is a subset of  $\mathcal{R}$  that needs to be considered by any algorithm in order to construct  $\Xi_P^*$ . We will denote this set by  $\tilde{\mathcal{R}}(P)$ . Given  $P$ , regions in  $\mathcal{R}$  may be dominated by a point preceding or succeeding them. Moreover, any algorithm must somehow decide/discover that these regions do not need further consideration. For a region  $R_i$ , the set  $V_i(P)$  is the set of dominated regions succeeding  $R_i$  where  $R_i$  is ‘responsible’ for discovering that these regions are dominated. The set  $H_i(P)$  is defined symmetrically for regions preceding  $R_i$ . Intuitively, the fastest a region  $R_i$  can discover these subsets  $V_i(P)$  and  $H_i(P)$  is in  $\Omega(\log |V_i(P)| + \log |H_i(P)|)$  time via searching. Later in the chapter, we show that these sets are index-connected sets of regions where we know the start index. Thus, we can identify them through exponential search.

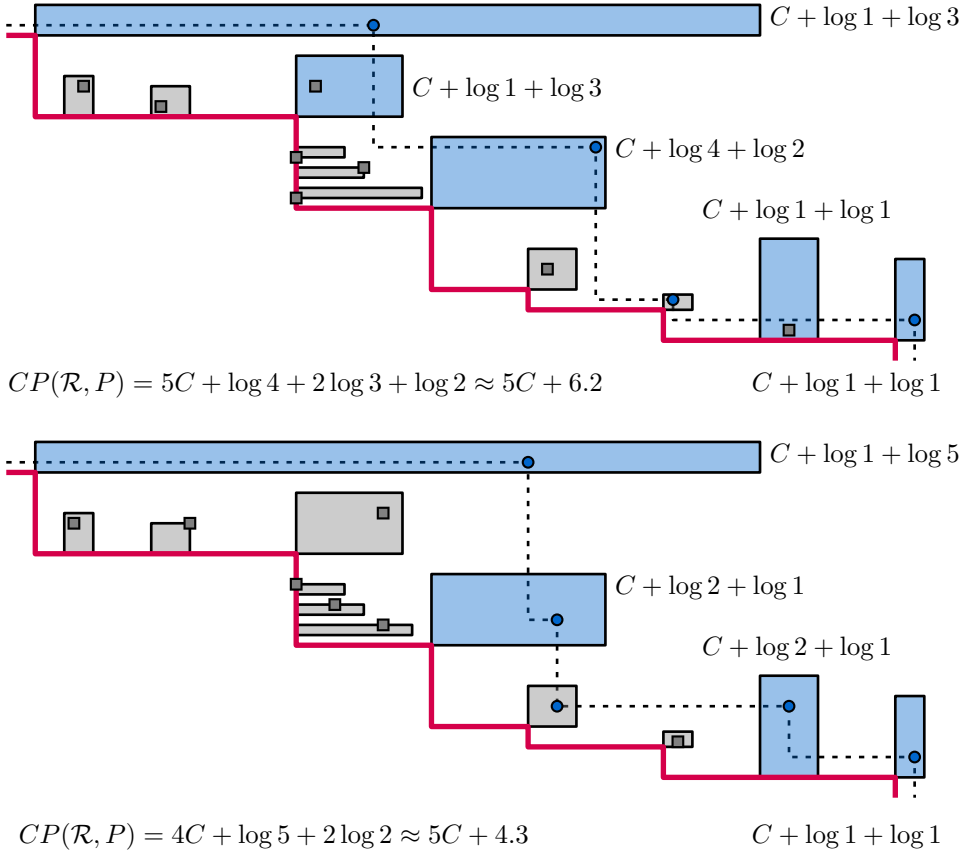
Now we formally define these concepts. For all regions  $R_i \in \mathcal{R}$ , we denote by  $V_i$  the subset of  $[R_i, R_n]$  that is vertically visible from  $R_i$  (including  $R_i$  itself) and by  $H_i$  the subset of  $[R_1, R_i]$  that is horizontally visible from  $R_i$  (including  $R_i$  itself). Given  $P$ , we denote by  $V_i(P) \subseteq V_i$ : the union of  $\{R_i\}$  with the subset of  $V_i$  of regions that are dominated by a point  $p_j$  with  $j \leq i$ . Formally:

$$V_i(P) := \{R_i\} \cup \{R_k \in V_i \mid R_k \text{ dominated by } p_j \in P \text{ with } j \leq i\}.$$

The set  $H_i(P)$  is defined symmetrically as the union of  $\{R_i\}$  with the subset of  $H_i$  of regions that are dominated by a point  $p_j$  with  $i \leq j$  (refer to Figure 10.3).

Given a truncated set  $\mathcal{R}$  and  $P$ , the set  $\tilde{\mathcal{R}}(P)$  is the subset of  $\mathcal{R}$  where each region  $R_i \in \tilde{\mathcal{R}}(P)$  is intersected by the Pareto front of  $P$  and  $R_i$  is not a sink in  $G(\mathcal{R})$ . By  $C$  we denote the unspecified cost for a retrieval. Logarithms are base 2. We define the *Pareto cost function* as (refer to Figure 10.4 for a numerical example):

$$\text{CP}(\mathcal{R}, P) = \sum_{R_i \in \tilde{\mathcal{R}}(P)} (C + \log |H_i(P)| + \log |V_i(P)|).$$



**Figure 10.4** Twice the same set of regions  $\mathcal{R}$  with a different corresponding point set. Blue points lie on the Pareto front of  $P$ . Blue regions are in  $\tilde{\mathcal{R}}(P)$ . Observe that there can exist points on the Pareto front of  $P$  where their corresponding regions are not in  $\tilde{\mathcal{R}}(P)$ . For every region  $R_i \in \tilde{\mathcal{R}}(P)$ , we illustrate the associated value  $C + \log |H_i(P)| + \log |V_i(P)|$  that  $R_i$  contributes to  $\text{CP}(\mathcal{R}, P)$ .

## 10.3 Lower bounds

The goal is to reconstruct a structure  $\Xi_P^*$  (the maximal staircase graph where top vertices are regions  $R_i$  such that  $p_i$  is not dominated by a point in  $P$ ) as efficiently as possible. First, we briefly show a worst-case lower bound for the preprocessing phase: there exists a choice of input  $\mathcal{R}^\circ$  where  $\mathcal{R}^\circ = \mathcal{R}^\times = \text{Trunc}(\mathcal{R}^\circ)$  and  $G(\mathcal{R}^\circ)$  is a graph with no edges. In this case, the structure  $\Xi_P^*$  is isomorphic to  $\mathbb{B}_{\mathcal{R}^\circ}$  hence it is possible to construct  $\Xi_P^*$  in the preprocessing phase. If  $\mathcal{R}^\circ$  has  $m$  elements, constructing  $\mathbb{B}_{\mathcal{R}^\circ}$  has a well-known  $O(m \log m)$  worst-case lower bound.

Next, we show lower bounds for the reconstruction phase. Recall that an algorithm can use *any* auxiliary structure  $\Xi_{\mathcal{R}}$  to aid its computation. In the remainder of this section we consider *any* truncated set  $\mathcal{R} = \mathcal{R}^\times = \text{Trunc}(\mathcal{R}^\circ)$  of  $n$  elements. We provide an information-theoretical lower bound, which depends on  $\mathcal{R}$  and  $P$ , for both the number of RAM instructions and disk retrievals required to construct  $\Xi_P^*$  regardless of auxiliary structure  $\Xi_{\mathcal{R}}$ .

### 10.3.1 A lower bound on RAM instructions

In Chapter 8 we showed by an information-theoretical argument that, for any  $\mathcal{R}$ , the uncertainty-region lower bound (of the number of RAM instructions in the reconstruction phase) is at least  $\Omega(\log |e(\mathcal{R})|)$  (where  $|e(\mathcal{R})|$  is the set of combinatorially distinct Pareto fronts of point sets that respect  $\mathcal{R}$ ). We prove:

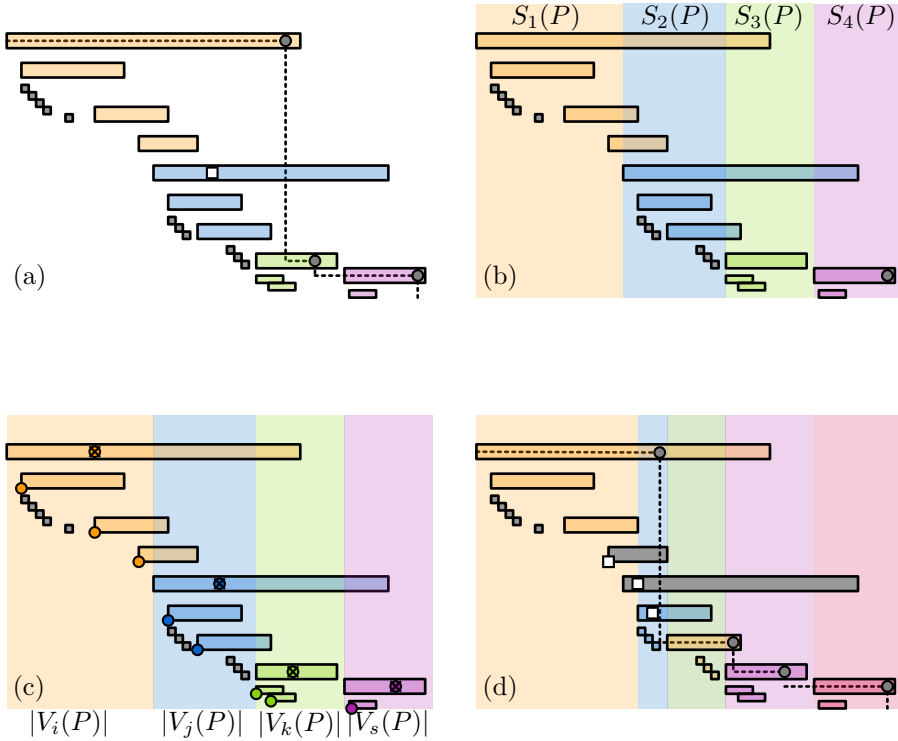
**Lemma 10.4** *Let  $\mathcal{R}$  be a truncated set and  $P$  be any point set that respects  $\mathcal{R}$ . Then*

$$\sum_{R_i \in \tilde{\mathcal{R}}(P)} \log |H_i(P)| + \log |V_i(P)| \leq 2 \cdot \log |e(\mathcal{R})|.$$

**Proof** We show that  $\sum_{R_i \in \tilde{\mathcal{R}}(P)} \log |V_i(P)| \leq \log |e(\mathcal{R})|$ . By a symmetric argument we have  $\sum_{R_i \in \tilde{\mathcal{R}}(P)} \log |H_i(P)| \leq \log |e(\mathcal{R})|$  and the lemma follows. Consider for a fixed set  $P$  the  $M$  regions  $R_i \in \tilde{\mathcal{R}}(P)$  for which  $|V_i(P)| \geq 2$  (recall that  $R_i \in V_i(P)$ ). We sort these regions from lowest index to highest. For ease of exposition, we create a permutation  $\pi$  to refer to these regions as  $(R_{\pi(1)}, R_{\pi(2)}, \dots, R_{\pi(M)})$  (for each point set  $P$  this corresponding permutation is different).

We make a distinction between strips (areas bounded by two vertical lines) and vertical halfslabs (areas bounded by two vertical lines and upper bounded by a point or a bottom facet of a rectangle). We create  $M$  pairwise disjoint strips (we illustrate the construction in Figure 10.5)  $S_1(P), S_2(P), \dots, S_M(P)$ :  $S_1(P)$  is bounded by the left facets of  $R_{\pi(1)}$  and  $R_{\pi(2)}$ ,  $S_2(P)$  by the facets of  $R_{\pi(2)}$  and  $R_{\pi(3)}$  and the  $M$ -th strip is a halfplane. In the degenerate case that a strip has width 0 (this can occur, as after truncation regions can have left vertices that share a coordinate) we give it width  $\varepsilon$  where  $\varepsilon > 0$  is an arbitrarily small constant.

Let  $i = \pi(1)$  and  $j = \pi(2)$ . For all regions  $R_k \in V_i(P)$ , per definition  $i \leq k < j$ . Each of these truncated regions has thus a bottom left vertex that lies left of the bottom left vertex of  $R_j$  and right of the bottom left vertex of  $R_i$ . This implies that their bottom left vertex lies in the strip  $S_1(P)$ . Thus given  $\mathcal{R}$ , there are at least  $|V_i(P)|$  combinatorially different Pareto fronts contained within  $S_1(P)$  (these Pareto fronts are obtained by placing the points of the regions in  $V_i(P) \setminus R_i$  on their respective bottom left endpoints, and by letting  $p_i$  dominate any prefix of these points).



**Figure 10.5** (a) A pair  $(\mathcal{R}, P)$  with the grey points as the Pareto front. For this fixed pair  $(\mathcal{R}, P)$ , we highlight each set  $V_i(P)$  for  $R_i \in \mathcal{R}(P)$ . (b) Based on each set  $V_i(P)$ , we create vertical strips. (c) In each strip  $S_k(P)$  we can create at least  $|V_{\pi(k)}(P)|$  combinatorially distinct (partial) Pareto fronts using only points in the strip. There may exist many more distinct Pareto fronts (for example, those obtained by utilising points in the grey regions). (d) A different point set  $P$  implies vastly different strips. The key argument here is that for *every* fixed  $P$ , we undercount  $\log |e(\mathcal{R})|$ .

Let  $j = \pi(2)$  and  $k = \pi(3)$ . Via the same argument each region in  $V_j(P)$  has its bottom endpoint contained in the strip  $S_2(P)$ . Hence, with the same argument as above, there are at least  $|V_j(P)|$  combinatorially different Pareto fronts contained within the second strip. Moreover, we created the earlier  $|V_i(P)|$  combinatorial outcomes by placing only points in the first strip, using only points preceding  $p_j$ . This means that these combinations can be generated, whilst no point preceding  $p_j$  dominates any point following  $p_j$ . This implies that the total number of combinatorially different Pareto fronts contained in both the first and second strip is  $|V_i(P)| \cdot |V_j(P)|$ . By applying this argument recursively it follows that:

$$\prod_{R_i \in \tilde{\mathcal{R}}(P)} |V_i(P)| \leq e(\mathcal{R}),$$

which concludes the proof. ■

### 10.3.2 A lower bound for disk retrievals

We revisit the instance-lower bound proof by Bruce et al. [35]. The prior result by Bruce et al. enables one to identify for each set of regions, a triple of uncertainty regions for which at least one point needs to be retrieved. The authors then retrieve all three points, remove all areas dominated by a point, and recurse. Here, we do essentially the same. The novelty in our argument lies in the fact that for each pair  $(\mathcal{R}, P)$  we can immediately (non-recursively) characterize the regions which require a disk retrieval using  $\tilde{\mathcal{R}}(P)$  which is a prerequisite for creating a data structure that will return these regions.

Bruce et al. study in their paper the reconstruction of the Pareto front of  $P$  in a variant of (what would later be) the preprocessing model with indirect representation. They present an iterative retrieval strategy that is instance optimal. Their strategy performs at most three times more retrievals than any algorithm must use to discover the Pareto front of  $P$  and they prove that this factor-3 redundancy is the best anyone can do. Their strategy describes the regions that must be considered in a geometric sense, not an algorithmic sense. That is, at each iteration they can identify a triplet of regions to query. But they have no algorithmic procedure to identify these three regions as such, nor a way to beforehand specify which regions should be considered. In their model this is justifiable as they assume that the retrieval cost  $C$  vastly dominates any RAM instructions and hence identifying the triple each iteration is trivial. In this chapter, we drop the assumption that  $C$  is enormous and are interested in a retrieval strategy which not only minimizes the number of retrievals, but which can also elect which points to retrieve efficiently.

We note that the query strategy of Bruce et al. produces a result of the same quality as the lemma below and naturally, our proofs share some elements which we fully wish to attribute to the work of [35]. The novelty in our result is that for each pair  $(\mathcal{R}, P)$  we are able to characterize the regions which require a disk retrieval using  $\tilde{\mathcal{R}}(P)$ . This helps us (in the reconstruction phase) to efficiently identify these regions.

**Lemma 10.5** Let  $\mathcal{R}$  be a truncated set and let  $P$  be any point set that respects  $\mathcal{R}$ . Any algorithm that constructs  $\Xi_P^*$  of  $P$  must perform at least  $\frac{1}{3}|\tilde{\mathcal{R}}(P)|$  retrievals of cost  $C$  each.

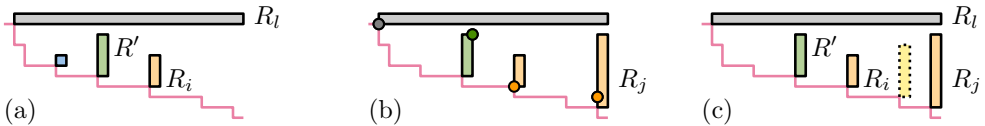
**Proof** For a truncated set  $\mathcal{R}$  with point set  $P$ , the set  $\tilde{\mathcal{R}}(P)$  is the set of all regions intersected by the Pareto front of  $P$  which are not a sink in the dependency graph  $G(\mathcal{R})$ . Consider a region  $R$  that is intersected by the Pareto front of  $P$  and let  $p$  be its corresponding point. Since  $R$  is not dominated by a point in  $P$  it must be that given  $P \setminus \{p\}$ , there exists a choice of  $P$  such that  $p$  appears on the Pareto front of  $P$ . Any algorithm *must* spend a disk retrieval to retrieve  $p$  if there also exists a choice where  $p$  does not appear on the Pareto front of  $P$ . We use a charging argument, where we charge each region in  $\tilde{\mathcal{R}}(P)$  to a region  $R$  for which this is the case.

Let  $R_i$  be a region in  $\tilde{\mathcal{R}}(P)$ . Since  $R_i$  is not a sink there is at least one region  $R' \in H_i \cup V_i$ . We illustrate our argument in Figure 10.6. Denote by  $R'$  a region in  $H_i$  (the case for  $V_i$  is symmetrical). Moreover, let  $R'$  be the region in  $H_i$  with the highest index. We charge the region  $R'$  one disk retrieval. First we show that each region in  $\mathcal{R}$  gets charged at most twice, then we show this charge is justified (that is, for any triple of regions where two regions charge a single region, any algorithm must perform at least one disk retrieval).

Suppose that  $R'$  gets charged by two regions  $R_i, R_j$  with  $R' \in H_i$  and  $R' \in H_j$  (the argument for when  $R'$  lies in two vertical halfslabs is symmetrical) and let  $i < j$ . If  $R'$  lies in  $H_i$  and  $H_j$ , then  $R_i$  must lie in the horizontal halfslab of  $R_j$ , which contradicts the assumption that  $R'$  was the region in  $H_j$  with the highest index (see Figure 10.6).

Second we show that this charge is justified. Consider  $R'$  and the two regions  $R_i$  and  $R_l$  ( $l < i$ ) that charge  $R'$  and all points in  $P \setminus \{p', p_i, p_l\}$ . There is no point in  $P$  that dominates either  $R_i$  or  $R_l$  (since per definition, these two regions are intersected by the Pareto front of  $P$ ).

We now make a case distinction: first suppose that there is no point in  $P \setminus \{p_i, p_l\}$  that dominates  $R'$  (since  $R' \in V_l$  and  $R' \in H_i$ ). This implies that regardless of all other points, there is a choice for  $p_i, p_l, p'$  where all three points appear on the Pareto front of  $P$  (the point placement where  $p_i$  and  $p_l$  appear on the bottom left vertex of their respective regions and  $R'$  appears on the top right vertex). However, there also exists a choice where  $p'$  is dominated by  $p_l$  or  $p_i$ . Any algorithm must therefore consider at least  $p', p_i$  or  $p_l$  in order to find out and this is why the charge is justified.



**Figure 10.6** (a)  $R_l$  charges the blue region and  $R_i$  the green. (b) For  $R_j$ , it must be that either  $R_i \in H_j$ , or (c) there is another region with higher index in  $H_j$ .

Next suppose that there is a point  $p_j$  that dominates  $R'$ . Since  $R'$  is the region in  $H_i$  with the highest index, it must be that  $j > i$  (or  $j < l$  in which case the argument is symmetrical). Since dominating points is transitive, we can assume that  $p_j$  appears on the Pareto front of  $P$ . Moreover, the region  $R_i$  is not dominated by a point in  $P$ . Hence, given  $P \setminus \{p', p_i, p_l\}$  there exists a choice of  $p_i$  where it appears on the Pareto front of  $P$  (namely, the choice where  $p_i$  lies on the top right vertex of  $R_i$  and  $p_l$  and  $p'$  appear on the bottom left vertex). Also, there exists a placement where  $p_i$  does not appear on the Pareto front of  $P$  (the choice where  $p_i$  lies on the bottom left vertex of  $R_i$  and is hence dominated by  $p_j$ ). Thus, any algorithm must consider  $p_i$  and this is why the charge is justified. ■

Given Lemma 10.4 and Lemma 10.5 we can immediately conclude the following:

**Theorem 10.1** *Let  $\mathcal{R}$  be a truncated set and  $P$  be any set that respects  $\mathcal{R}$ . Then  $\text{CP}(\mathcal{R}, P)$  is smaller than three times the uncertainty-region lower bound of  $\mathcal{R}$ .*

We wish to briefly note that for each index  $i$ ,  $V_i(P)$  and  $H_i(P)$  have at most  $n$  elements. It follows that for all region  $R_i \in \mathcal{R}$ , the values  $\log |H_i(P)|$  and  $\log |V_i(P)|$  are at most  $\log n$ . Thus, by Lemma 10.5 the instance lower bound for constructing the Pareto front  $\Xi_P^*$  is at least  $\Omega(\frac{1}{\log n} \text{CP}(\mathcal{R}, P))$ .

## 10.4 Geometric preliminaries for reconstruction

Theorem 10.1 gives an uncertainty-region lower bound for any truncated set  $\mathcal{R}$ . Next, we aim to match this lower bound. To this end, we first introduce some additional concepts that are required to explain our reconstruction algorithm. Throughout this section we assume that  $\mathcal{R}$  is a truncated (later even *canonical*) set of  $n$  regions and that  $P$  is a point set that respects  $\mathcal{R}$ . In Section 10.5 we show that after preprocessing, this assumption is justified. Denote for each index  $i$  by  $V_i^{\text{next}}$  the lowest-index region strictly right of the vertical halfslab of  $R_i$ ;  $H_i^{\text{prev}}$  is defined symmetrically using the highest index.

Let  $R_i \in \mathcal{R}$  be an isolated vertex in  $G(\mathcal{R})$ . By Lemma 10.3,  $p_i$  appears on the Pareto front and connects the Pareto front of  $[p_1, p_{i-1}]$  and  $[p_{i+1}, p_n]$ . Thus, we can split the problem of computing the Pareto front of  $P$  into two, and solve each half independently. We say that a truncated set  $\mathcal{R}$  is *culled* if  $G(\mathcal{R})$  contains no region that is an isolated vertex. Let  $[R_i, R_j]$  be a sequence of sinks in  $G(\mathcal{R})$ , and  $R^*$  be the smallest rectangle that contains  $R_i$  and  $R_j$ . We can use  $R^*$  to capture a ‘streak’ of points which do, or do not, appear on the Pareto front:

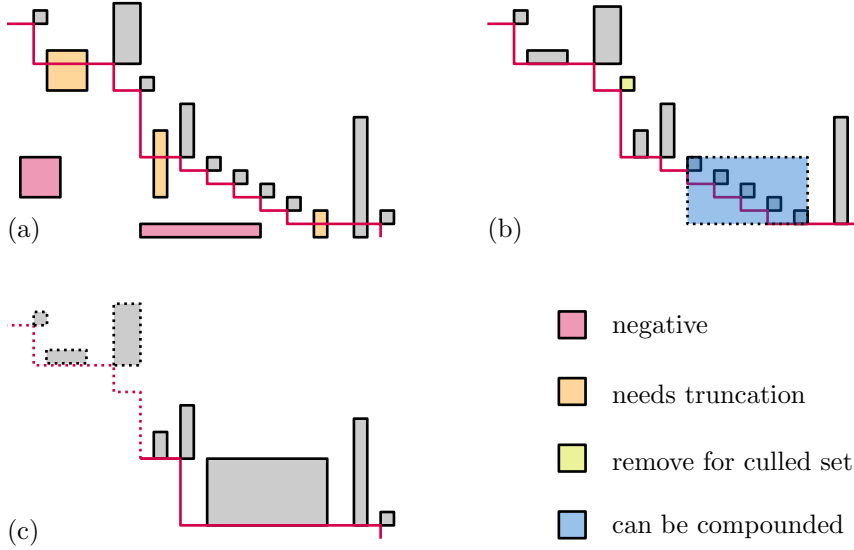
**Lemma 10.6** *Let  $[R_i, R_j]$  be a sequence of sinks in  $G(\mathcal{R})$ . If  $\nexists p_k$  preceding  $p_i$  and dominating  $p_i$  then all points preceding  $p_i$  do not dominate points in  $[p_i, p_j]$ . If such  $p_k$  dominates  $p_j$ , then  $p_k$  dominates all points in  $[p_i, p_j]$ . Similar statements hold for  $p_k$  succeeding  $p_j$ .*

**Proof** Any  $p_k$  that dominates any point  $p_s$  with  $s \in (i, j)$ , but not  $p_i$  or  $p_j$  itself must lie in the interior of  $R^*$ , but  $R^*$  contains only points whose regions are sinks in  $G(\mathcal{R})$ . This contradiction implies all claims of the lemma. ■

This lemma implies that if both  $p_i$  and  $p_j$  are not dominated by other points in  $P$  then all the points in  $[p_i, p_j]$  appear on the Pareto front of  $P$  as a contiguous subsequence. For any maximal sequence of sinks  $[R_i, R_j]$  in a truncated and culled set  $\mathcal{R}$ , we define their *compound region*  $R^*$  as the minimal rectangle that contains  $[R_i, R_j]$  and we replace  $[R_i, R_j]$  in  $\mathcal{R}$  with  $R^*$  (refer to Figure 10.7(b)). Let  $\mathcal{R}^*$  be the resulting set of regions. The region  $R^*$  is a sink in  $G(\mathcal{R}^*)$  and a region  $R$  has an outgoing arrow to  $R^*$  in  $G(\mathcal{R}^*)$  if and only if it had an outgoing arrow in  $G(\mathcal{R}^\infty)$  to at least one region in  $[R_i, R_j]$ . Since  $R^*$  is just another rectangle disjoint from all other rectangles in  $\mathcal{R}^*$ , the definition of *truncated* and *culled* still applies to  $\mathcal{R}^*$ . We say a set  $\mathcal{R}^*$  is a *canonical set* if it is truncated, culled, and if there are no two consecutive regions that are sinks in  $G(\mathcal{R}^*)$ .

**Subproblems.** Let  $\mathcal{R}$  be a *truncated* set. We say two indices  $i < j$  form a *subproblem* with respect to  $G(\mathcal{R})$  if  $R_i$  and  $R_j$  are sources in  $G(\mathcal{R})$  and there does not exist a region  $R_k$  with  $i < k < j$  that is also a source. Abusing notation, we say that  $[R_i, R_j]$  is a subproblem of  $G(\mathcal{R})$ . Later, we consider some altered dependency graph  $G(\mathcal{R}')$  and refer to subproblems  $[R_l, R_m]$  of  $G(\mathcal{R}')$ .

**The subproblem tree.** Given a truncated set  $\mathcal{R}$ , we define a subproblem tree on the dependency graph  $G(\mathcal{R})$ . Intuitively the children of each node  $[R_i, R_j]$  store the maximal intervals that *could* become a subproblem after  $p_i$  and  $p_j$  are retrieved. Formally, the subproblem tree (denoted by  $T_{\mathcal{R}}$ ) is a range tree on the interval  $[1, n] \subset \mathbb{Z}$  (Figure 10.8). The root node of the subproblem tree stores the interval  $[1, n]$ . If  $\mathcal{R}$  is a canonical set, the subproblems of  $\mathcal{R}$  partition  $\mathcal{R}$ , and the root node has a child for each subproblem  $[R_i, R_j]$ . The child stores the interval  $[i, j]$  and a pointer to  $R_i$  and  $R_j$ . We construct the subsequent children as follows: for each node  $[i, j]$ , we remove all outgoing arrows from  $R_i$  and  $R_j$  and we create a child node for each subproblem of  $G([R_i, R_j])$  without these arrows. Note that each node has at least two children as removing the outgoing arrows from  $R_i$  and  $R_j$  creates at least one additional source  $R_k$  with  $k \in (i, j)$  and  $R_i$  and  $R_j$  remain sources.



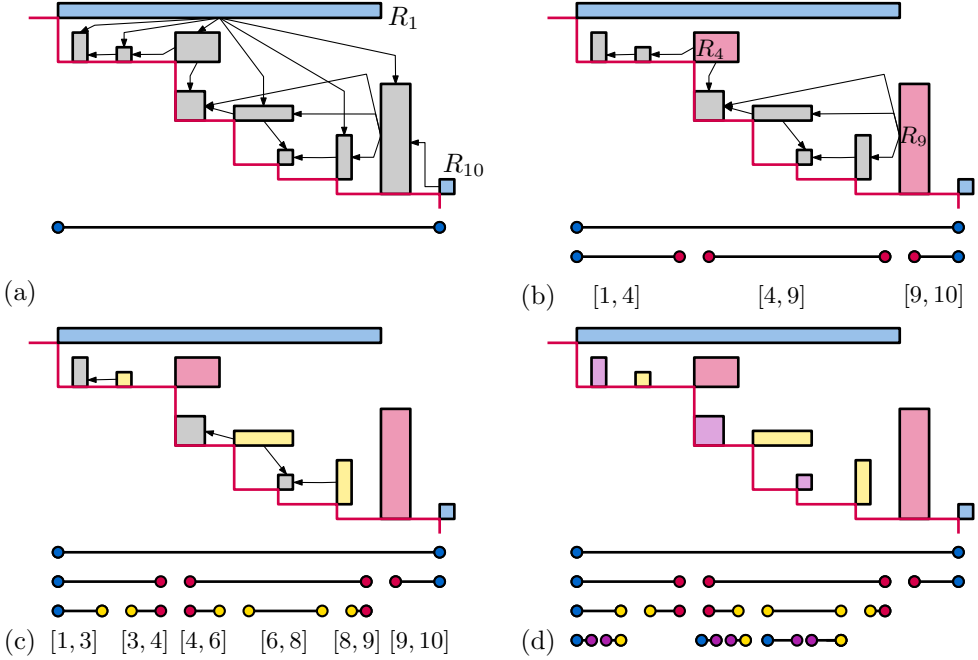
**Figure 10.7** (a) regions  $\mathcal{R}^*$  with  $\mathbb{B}_{\mathcal{R}^*}$  in red. (b) The set  $\mathcal{R}^x$  of regions after truncation. The lime region is an isolated vertex of  $G(\mathcal{R}^x)$  and it splits our algorithmic problem in two. (c) The resulting canonical set, split by culled regions.

## 10.5 Preprocessing phase

Here, we elaborate on the preprocessing procedure. In this section we assume that we are given a set  $\mathcal{R}^*$  of  $m$  regions. We show how to transform  $\mathcal{R}^*$  into the truncated set  $\mathcal{R} = \mathcal{R}^x$  of  $n$  regions. Finally, we transform the truncated set  $\mathcal{R}$  into its canonical set  $\mathcal{R}^*$  in  $O(n \log n)$  time. At the same time, we define the auxiliary data structure  $\Xi_{\mathcal{R}}$  and we show how to construct it whilst transforming  $\mathcal{R}$  into  $\mathcal{R}^*$ .

**Lemma 10.7** *For any set  $\mathcal{R}^*$  of  $m$  axis-aligned, pairwise disjoint axis-aligned rectangles we can construct its truncated set  $\mathcal{R} = \mathcal{R}^x$  of  $n$  rectangles in  $O(m \log m)$  time.*

**Proof** We identify the bottom left vertices of all regions in  $\mathcal{R}^*$  and construct  $\mathbb{B}_{\mathcal{R}^*}$  together with a range tree on the horizontal edges of  $\mathbb{B}_{\mathcal{R}^*}$  [61] in  $O(m \log m)$  time. For each region  $R \in \mathcal{R}^*$  we detect whether  $R$  is negative by performing a point location with its top right vertex on the interior of  $\mathbb{B}_{\mathcal{R}^*}$ ; if it is negative then it is discarded. If a region  $R \in \mathcal{R}^*$  is not negative then by Lemma 10.2 we know that  $R \cap \mathbb{B}_{\mathcal{R}^*}$  is a staircase of constant complexity which we compute in logarithmic time using binary search on  $\mathbb{B}_{\mathcal{R}^*}$ . For each non-negative  $R \in \mathcal{R}^*$  whose interior intersects  $\mathbb{B}_{\mathcal{R}^*}$  we store its region after truncation. This results in a set  $\mathcal{R} = \mathcal{R}^x$  of  $n$  pairwise disjoint axis-aligned rectangles, which we sort and re-index based on their intersection with  $\mathbb{B}_{\mathcal{R}^*}$  in  $O(m \log m)$  time. ■

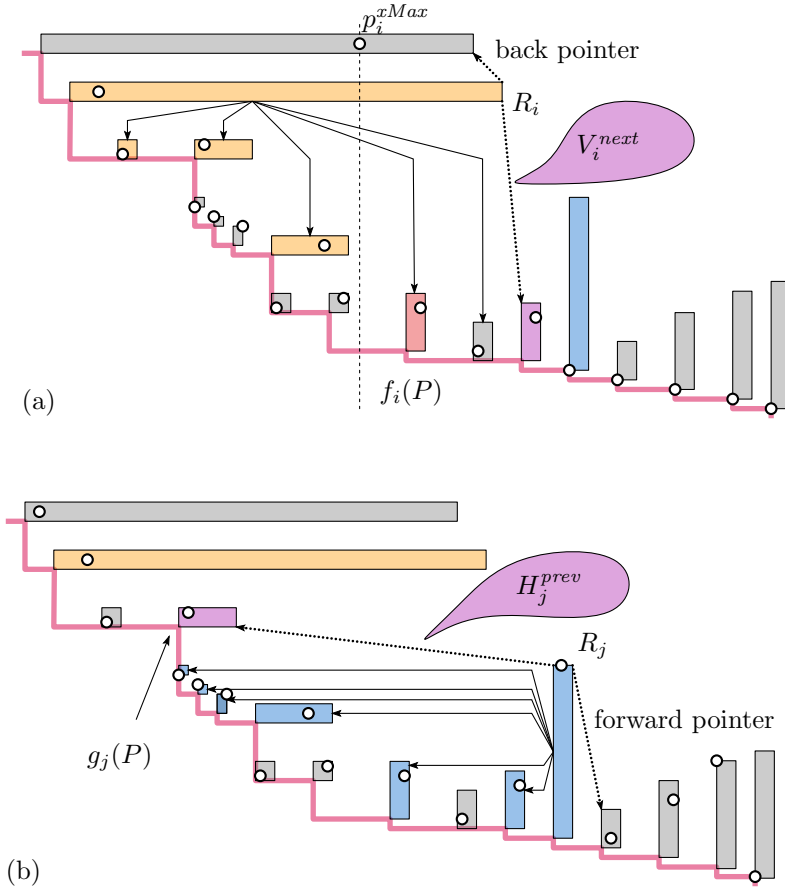


**Figure 10.8** Four iterations that together create the subproblem tree. The additional sources in the dependency graph are blue, red, yellow and purple for the first, second, third and fourth iteration respectively.

**Defining the auxiliary structure  $\Xi_{\mathcal{R}}$ .** Henceforth, we denote by  $\mathcal{R}$  the truncated set  $\mathcal{R} = \mathcal{R}^\infty$ . In the remainder of the preprocessing phase, we spend  $O(n \log n)$  time to transform  $\mathcal{R}$  into a canonical set  $\mathcal{R}^*$ , construct  $G(\mathcal{R})$  and  $G(\mathcal{R}^*)$  and construct the data structure  $\Xi_{\mathcal{R}}$ . To this end, we first define the auxiliary structure. Recall that for any truncated set  $\mathcal{R}$  we denote by  $H_i$  the set of regions  $R_j$  in  $\mathcal{R}$  with  $j < i$  which are horizontally visible from  $R_i$  and by  $V_i$  the set of regions  $R_j$  with  $j > i$  which are vertically visible from  $R_i$ .

Let  $\mathcal{R}$  be a truncated set and  $\mathcal{R}^*$  be the corresponding canonical set. The auxiliary structure  $\Xi_{\mathcal{R}}$  is defined as the dependency graph  $G(\mathcal{R}^*)$  together with the subproblem tree  $T_{\mathcal{R}^*}$  and a linked list of the culled regions. In addition, we store for every region  $R_i \in \mathcal{R}$  the following four *attributes* (Attribute 1 and 2 are illustrated by Figure 10.9):

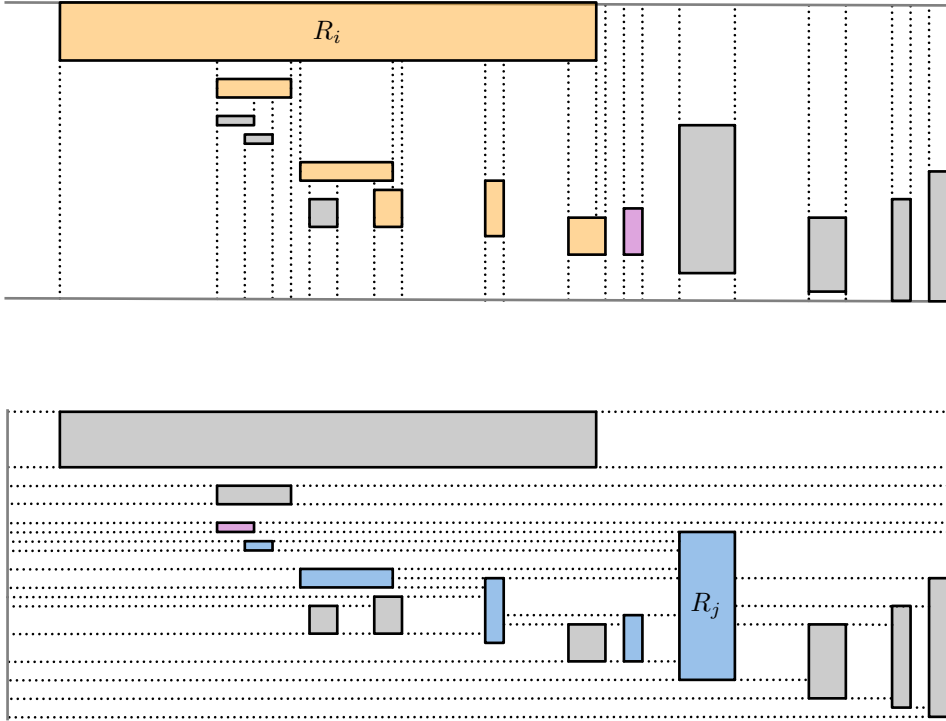
1. A balanced binary tree on  $V_i$  and  $H_i$  from  $\mathcal{R}$ .
2. A pointer to  $V_i^{\text{next}}$  and  $H_i^{\text{prev}}$  in  $\mathcal{R}^*$ .
3. A pointer to the highest node in  $T_{\mathcal{R}^*}$  that stores an interval  $[\cdot, i]$ , and a pointer to the highest node in  $T_{\mathcal{R}^*}$  that stores an interval  $[i, \cdot]$ .
4. If  $R_i$  is a compound region, an array and a linked list of all the regions compounded in  $R_i$ .



**Figure 10.9** (a) We show a region  $R_i$  with the set  $V_i(P)$  in orange (for every point set  $P$ , the set  $V_i(P)$  may change). The red region is the region  $f_i(P)$ : the first region not dominated by a point preceding  $p_i$ . The region  $R_i$  has a pointer to  $V_i^{next}$ . (b) We show region  $R_j$  with the set  $H_j(P)$  in blue. The region  $R_j$  has a pointer to  $H_j^{prev}$ .

To construct this data structure, we first observe the following:

**Observation 10.1** For any truncated set  $\mathcal{R}$ , a region  $R_j \in \mathcal{R}$  is vertically visible from a region  $R_i \in \mathcal{R}$  if and only if there exists a face or edge in the vertical decomposition of  $\mathcal{R}$  which is adjacent to both  $R_i$  and  $R_j$ . Similarly, a region  $R_j$  is horizontally visible from a region  $R_i$  if and only if there exists a face or edge in the horizontal decomposition of  $\mathcal{R}$  which is adjacent to both  $R_i$  and  $R_j$ . This observation is illustrated by Figure 10.10.



**Figure 10.10** We show a collection of regions. We obtain the vertical decomposition of the regions by shooting from every vertex of every region a vertical ray. The horizontal decomposition is obtained by shooting from every vertex of every region a horizontal ray. For the region  $R_i$  we show the set  $V_i$  in yellow and  $V_i^{next}$  in purple. Observe that all regions in  $R_i$  are adjacent to at least one cell that is also adjacent to  $R_i$ . For the region  $R_j$  we show the set  $H_j$  in blue and  $H_j^{prev}$  in purple.

Using Observation 10.1 we obtain the following:

**Lemma 10.8** For any truncated set  $\mathcal{R}$  of  $n$  axis-aligned, pairwise disjoint rectangles we can construct its canonical set  $\mathcal{R}^*$  and  $\Xi_{\mathcal{R}}$  in  $O(n \log n)$  time.

**Proof** A vertical or horizontal decomposition has a number of faces and edges which is linear in the number of input vertices and can be constructed in  $O(n \log n)$  time [61]. Given the vertical decomposition of  $\mathcal{R}^\times$ , we can traverse it in linear time to store for each region  $R_i$  the set  $V_i$ . Similarly we can identify and store  $H_i$  for each  $R_i$ , and in  $O(n \log n)$  total time we construct a balanced binary tree on each set  $H_i$  and  $V_i$  to obtain Attribute 1.

For each set  $V_i$ , we identify  $V_i^{next}$  in logarithmic time by searching for the left-most bottom-left endpoint right of the vertical halfslab through  $R_i$  to obtain Attribute 2.

Through this procedure, we construct the dependency graph  $G(\mathcal{R})$  in  $O(n \log n)$  time by iterating over all nodes in the vertical decomposition. In linear time, we can identify the connected components of  $G(\mathcal{R})$  and the regions which are an isolated vertex in  $G(\mathcal{R})$ . By Lemma 10.3, can solve each connected component of  $G(\mathcal{R})$  independently and that the solutions must be concatenated through the regions that are an isolated vertex. We store the connected components of  $G(\mathcal{R})$  as a doubly linked list and remove all isolated vertices from  $\mathcal{R}^\times$  to create a *culled* set.

To transform a culled set into a canonical set, we identify all sinks in the graph in linear time (by checking if  $|V_i| = |H_i| = 1$ ) and we iterate over all regions in order of their index. Neighboring sinks get recursively grouped into a compound region and this procedure creates a canonical set in linear time. For each region compounding  $k$  regions, we construct the corresponding array and doubly linked list of Attribute 4 in  $O(k)$  time. After having compound all regions, we do a linear-time scan to re-index all the (compound) regions so that all indices are consecutive and we obtain a *canonical set*  $\mathcal{R}^*$ . Moreover, whenever we compound a set  $[R_i, R_{i+k}]$  into a region  $R$ , we make sure to remove  $[R_i, R_{i+k}]$  from  $G(\mathcal{R})$  and replace it with  $R$  (where all arrows pointing to a region in  $[R_i, R_{i+k}]$  now point to  $R$ ). In this way, we simultaneously create  $G(\mathcal{R}^*)$ .

Lastly, we want to obtain from a canonical set  $\mathcal{R}^*$  its subproblem tree  $T_{\mathcal{R}^*}$  in  $O(n)$  time using the prior constructed  $G(\mathcal{R}^*)$ . This can be done as follows: first we identify the subproblems of  $G(\mathcal{R}^*)$  in linear time. Then for each subproblem  $[R_i, R_j]$  of  $G(\mathcal{R})$  we (temporarily) remove all outgoing arrows from  $R_i$  and  $R_j$  from the graph. For each node that has an arrow from  $R_i$  or  $R_j$  we check if it becomes a source node in constant time. This gives us the child nodes of the node that stores  $[i, j]$  in  $T_{\mathcal{R}^*}$ . During this process we store for each region  $R_i$  a pointer to the largest intervals  $[i, \cdot]$  and  $[\cdot, i]$  in the tree  $T_{\mathcal{R}^*}$  in constant additional time per region to create Attribute 3 (these intervals must always exist, since every region  $R_i$  must become a source at some point in the process)). Applying this procedure recursively takes time linear in the number of edges in  $G(\mathcal{R})$ , which is linear in the number of cells of the vertical and horizontal decomposition of  $\mathcal{R}$ . This concludes the lemma. ■

Lemma 10.7 and Lemma 10.8, together with the observation that  $n \leq m$ , immediately imply the following theorem:

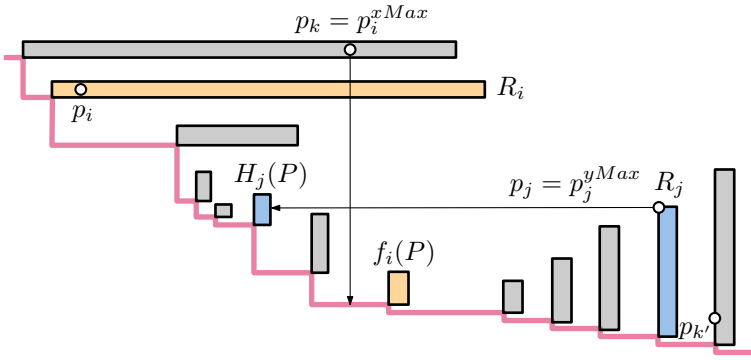
**Theorem 10.2** *For any set  $\mathcal{R}^\circ$  of  $m$  axis-aligned, pairwise disjoint axis-aligned rectangles we can construct its truncated set  $\mathcal{R}^\times$  and its canonical set  $\mathcal{R}^*$  and  $\Xi_{\mathcal{R}}$  in  $O(m \log m)$  time.*

## 10.6 Reconstruction phase

In this section we show how to compute for every pair of regions and points  $(\mathcal{R}, P)$  where  $P$  respects  $\mathcal{R}$  an indirect Pareto front of the point set  $P$ . We assume that we have access to the data structure  $\Xi_{\mathcal{R}}$  from Section 10.5 and that for every region  $R_i$ , we can retrieve its corresponding point  $p_i$  in  $O(C)$  time per point. The desired output, denoted by  $\Xi_P^*$ , is a pointer structure on  $\mathcal{R}$  such that if we replace every region  $R_i$  with its corresponding point  $p_i$ , we obtain the Pareto front of  $P$ . We first sketch our approach. Then, we prove some prerequisite lemmas and finally we formalise our algorithmic solution and analyse its running time.

For every integer  $i$  we denote by  $p_i^{xMax}$  the point in  $P$  with maximal  $x$ -coordinate among points  $p_k$  with  $k \leq i$ . Similarly, we denote by  $p_i^{yMax}$  the point in  $P$  with maximal  $y$ -coordinate among points  $p_k$  with  $i \leq k$ . In addition we denote, for every integer  $i$ , by  $f_i(P)$  the lowest-index region succeeding  $R_i$ , not dominated by a point  $p_k$  with  $k \leq i$  and by  $g_i(P)$  the highest-index region preceding  $R_i$ , not dominated by a point  $p_k$  with  $k \geq i$ . Refer to Figure 10.11 for an example. On a high level, our reconstruction algorithm is an iterative procedure that at every iteration  $t$ , maintains an implicitly truncated set  $\mathcal{R}^t$  and a queue of subproblems of  $G(\mathcal{R}^t)$ . Every iteration we dequeue a subproblem  $[R_i, R_j]$  of  $G(\mathcal{R}^t)$  and retrieve  $p_i$  and  $p_j$  to determine if these points lie on the Pareto front of  $P$ . During this process, we maintain the following invariant:

**Invariant 10.1** *For each iteration when we consider a subproblem  $[R_i, R_j]$ , we have a pointer to  $p_{i-1}^{xMax}$ , where this pointer is null if there is no point in  $[p_i, p_j]$  that can be dominated by a point preceding  $p_i$ . We have a pointer to  $p_{j+1}^{yMax}$  or null through a symmetric condition.*



**Figure 10.11** A collection of regions where we show the points  $p_k$  with  $k \leq i$ , and the point  $p_{k'}$  with  $j \leq k'$ . Observe that the point  $p_i \neq p_i^{xMax}$ . We show  $R_i$  and  $V_i(P)$  in orange and  $R_j$  and  $H_j(P)$  in blue.

Now, we sketch our approach. We denote by the set  $\mathcal{R}^0$ , the canonical set  $\mathcal{R}^*$ . Every iteration  $t$ , we dequeue a subproblem  $[R_i, R_j]$  of the graph  $G(\mathcal{R}^t)$  from the queue and do the following:

- If  $R_i$  is a compound region, insert the subsequence of regions in  $R_i$ , whose respective points appear on the Pareto front of  $P$ , by pointing to the linked list (Attribute 4) in  $O(1)$  time.
- If  $R_j$  is a compound region, insert the subsequence of regions in  $R_j$ , whose respective points appear on the Pareto front of  $P$ , by pointing to the linked list (Attribute 4) in  $O(1)$  time.
- For non-compound regions  $R_i$  and  $R_j$ , retrieve  $p_i$  and  $p_j$  in  $O(C)$  time.
- If  $p_i$  is not dominated by either  $p_j, p_{i-1}^{xMax}$  or  $p_{j+1}^{yMax}$ , then  $p_i$  is on the Pareto front of  $P$  as a successor of  $p_{i-1}^{xMax}$ . Identify this in  $O(1)$  time with Invariant 1.
  - If  $p_{i-1}^{xMax}$  is null,  $p_i$  succeeds the Pareto front of the subproblem left of  $[R_i, R_j]$ .
- If  $p_j$  is not dominated by either  $p_i, p_{i-1}^{xMax}$  or  $p_{j+1}^{yMax}$ , then  $p_j$  is on the Pareto front of  $P$  as a predecessor to  $p_{j+1}^{yMax}$ . Identify this in  $O(1)$  time with Invariant 1.
  - If  $p_{j+1}^{yMax}$  is null,  $p_j$  precedes the Pareto front of the subproblem right of  $[R_i, R_j]$ .
- Identify  $f_i(P)$  with exponential search on  $V_i$  in  $O(\log |V_i(P)|)$  time.
- Identify  $g_j(P)$  with exponential search on  $H_j$  in  $O(\log |H_j(P)|)$  time.
- Create  $\mathcal{R}^{t+1}$  from  $\mathcal{R}^t$  by removing  $R_i$  and  $R_j$  replacing them with  $p_i$  and  $p_j$  in  $O(1)$  time.
- The subproblems of  $G(\mathcal{R}^{t+1})$  are the subproblems of  $G(\mathcal{R}^t)$  plus:  $[R_i, f_i(P)]$ ,  $[g_j(P), R_j]$  and every maximal subproblem in the subproblem tree  $T_{\mathcal{R}^*}$  that is contained in  $[f_i(P), g_j(P)]$ . Using a walk through  $T_{\mathcal{R}^*}$ , add every such subproblem to the queue in constant time each (charge the time needed for queuing to the corresponding subproblem).

The above procedure spends for every subproblem  $[R_i, R_j]$  at most  $O(C + \log |V_i(P)| + \log |H_j(P)|)$  time. In the remainder of this section, we elaborate on every algorithmic step and we prove that for any input  $(\mathcal{R}^0, P)$  our algorithm constructs  $\Xi_P^*$  (the maximal staircase graph where top vertices are regions  $R_i$  such that  $p_i$  is not dominated by a point in  $P$ ). Moreover, we prove that the algorithm running time matches the uncertainty region lower bound and conclude:

**Theorem 10.3** *Let  $\mathcal{R}^0$  be a set of  $m$  pairwise disjoint uncertainty rectangles in  $\mathbb{R}^2$  and denote by  $\mathcal{R} = \text{Trunc}(\mathcal{R}^0)$  its corresponding truncated set. We can preprocess  $\mathcal{R}^0$  in  $O(m \log m)$  time to reconstruct the Pareto front of the underlying point set  $P$  in  $\Theta(\text{CP}(\mathcal{R}, P))$  time, which is uncertainty-region optimal.*

### 10.6.1 Supporting proofs for reconstruction

Before we elaborate on our reconstruction algorithm we show some useful lemmas.

**Reporting points.** We show that given  $p_i^{xMax}$  and  $p_j^{yMax}$  we can identify if  $p_i$  and  $p_j$  appear on the Pareto front of  $P$ :

**Lemma 10.9** *Let  $\mathcal{R}'$  be any truncated set and  $[R_i, R_j]$  be any subproblem of  $G(\mathcal{R}')$ . The point  $p_i$  appears on the Pareto front of  $P$  if and only if  $p_i$  is not dominated by  $p_i^{xMax}$  or  $p_j^{yMax}$ . A symmetrical property holds for  $p_j$ .*

**Proof** If  $p_i$  appears on the Pareto front then per definition it is not dominated by  $p_i^{xMax}$  or  $p_j^{yMax}$ . What remains to prove, is that if  $p_i$  does not appear on the Pareto front, then it must be dominated by either  $p_i^{xMax}$  or  $p_j^{yMax}$ . Suppose for the sake of contradiction  $p_i$  is not dominated by  $p_i^{xMax}$  and  $p_j^{yMax}$ , but dominated by some point  $p_k$ . Then  $k < i$  or  $k > j$ , because  $R_i$  and  $R_j$  are both sources in  $G(\mathcal{R}')$ . If  $k < i$  then the  $x$ -coordinate of  $p_k$  is greater than of  $p_i$ , and thus  $p_i^{xMax} \neq p_i$ . In this case, the point  $p_i^{xMax}$  has greater  $x$ -coordinate than  $p_i$ , it lies in some region  $R' \neq R_i$ . Moreover, since  $R'$  precedes  $R_i$  and contains  $p_i^{xMax}$ , its bottom facet must lie above the top facet of  $R_i$ . Thus  $p_i^{xMax}$  dominates  $p_i$  which is a contradiction. If  $j < k$  then  $p_j^{yMax}$  dominates  $p_i$ . When we assume instead that  $p_j$  is dominated by a point  $p_k$ , the symmetrical argument applies. ■

The previous lemma implies that if Invariant 10.1 is maintained, we can iteratively identify points that appear on the Pareto front in constant time.

**Identifying  $f_i(P)$  and  $g_j(P)$ .** The region  $f_i(P)$  is the lowest-index region succeeding  $R_i$ , not dominated by a point  $p_k$  with  $k \leq i$ . We show that we can find  $f_i(P)$  efficiently using exponential search:

**Lemma 10.10** *Denote by  $\mathcal{R}$  any truncated set. For any subproblem  $[R_i, R_j]$  of  $G(\mathcal{R})$  either  $f_i(P) \in V_i$  or  $f_i(P) = V_i^{next}$ .*

**Proof** Any region in  $[R_i, R_j]$  that is dominated by a point preceding  $p_i$  is dominated by  $p_i^{xMax}$ . The point  $p_{i-1}^{xMax}$  cannot dominate  $R_i$ , as else  $R_i$  would have been removed during a truncation. Hence,  $f_i(P)$  is  $V_i^{next}$  or a region preceding it.

Suppose for the sake of contradiction that  $f_i(P)$  is a region preceding  $V_i^{next}$  and not in  $V_i$ . Consider any vertical ray from a point in  $R_i$ , right of  $p_i^{xMax}$  that intersects  $f_i(P)$  (such a ray must always exist, since  $f_i(P)$  precedes  $V_i^{next}$  and is not dominated by  $p_i^{xMax}$ ). Since  $f_i(P) \notin V_i$ , this ray must also intersect a region  $R' \in V_i$  (else this ray would be a line of sight to  $f_i(P)$ , which would imply  $f_i(P) \in V_i$ ). However, then  $R'$  must precede  $f_i(P)$  which contradicts the assumption that  $f_i(P)$  was the lowest-indexed region succeeding  $R_i$ , not dominated by  $p_i^{xMax}$ . ■

We note that in the auxiliary structure  $\Xi_{\mathcal{R}}$  every region  $R_i$  stores  $V_i$  and  $H_i$  has a balanced binary tree (Attribute 1) and stores a pointer to  $V_i^{next}$  and  $V_i^{prev}$  (Attribute 2). Thus, we conclude:

**Corollary 10.2** *Let  $\mathcal{R}'$  be a truncated set that is a subset of some truncated set  $\mathcal{R}$  and  $P$  be its underlying point set. Let  $[R_i, R_j]$  be a subproblem of  $G(\mathcal{R}')$ . Given Invariant 10.1 and  $\Xi_{\mathcal{R}}$ , we can identify  $f_i(P)$  in  $O(\log |V_i(P)|)$  time and  $g_j(P)$  in  $O(\log |H_j(P)|)$  time using exponential search.*

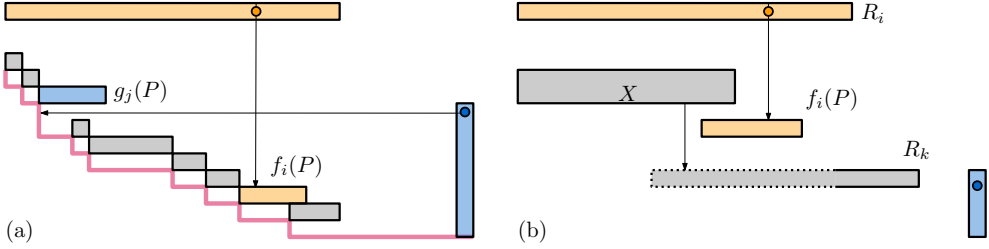
**Swiftly identifying new subproblems.** In our algorithmic sketch, we showed that our reconstruction algorithm will iteratively process a subproblem  $[R_i, R_j]$  of the current graph  $G(\mathcal{R}^t)$ . To achieve this we want that for every iteration  $t$ , we have access to a queue of all subproblems of  $G(\mathcal{R}^t)$ . Suppose that for some iteration  $t$  we processed a subproblem  $[R_i, R_j]$  and that  $[R_k, R_l]$  is a subproblem of  $G(\mathcal{R}^{t+1})$  but not of  $G(\mathcal{R}^t)$ . By Lemma 10.3, it must be that  $i \leq k \leq l \leq j$ . We show how to quickly identify every such ‘new’ subproblem  $[R_k, R_l]$  in three steps.

**Lemma 10.11** *Let  $\mathcal{R}^t$  be any truncated set. Let  $[R_i, R_j]$  be a subproblem of  $G(\mathcal{R}^t)$  and denote by  $\mathcal{R}^{t+1}$  the set obtained by replacing  $R_i$  and  $R_j$  with  $p_i$  and  $p_j$  respectively, and re-truncating. Let be  $v$  the lowest node in  $T_{\mathcal{R}^*}$  such that  $[i, j]$  is stored in  $v$ . For any child node  $[a, b]$  of  $v$ , there is no region  $R' \in [R_a, R_b]$  that is a source node in  $G(\mathcal{R}^{t+1})$  other than possibly  $R_a, R_b, f_i(P)$  or  $g_j(P)$ .*

**Proof** The proof is illustrated by Figure 10.12. Suppose that for the subproblem  $[R_i, R_j]$ ,  $f_i(P)$  succeeds  $g_j(P)$  then per definition all regions in  $[R_{i+1}, R_{j-1}]$  are dominated by a point in  $P \setminus [p_{i+1}, p_{j-1}]$  and thus removed after truncation of  $\mathcal{R}^{t+1}$ . Similarly, if  $f_i(P) = g_j(P)$  then only this region remains after truncation and thus, there cannot be any additional sources in  $G(\mathcal{R}^{t+1})$  (Figure 10.12(a)).

Let  $f_i(P)$  precede  $g_j(P)$ . We denote by  $[a, b]$  a descendent of  $v$  and let  $R_k$  be a region with  $k \in (a, b)$  succeeding  $f_i(P)$  and preceding  $g_j(P)$ . We denote  $\mathcal{R}^0 = \mathcal{R}^*$ . Per construction of  $T_{\mathcal{R}^*}$  each such region  $R_k$  has at least one incoming arrow from a region  $X \in [R_a, R_b]$ . The region  $R_k$  can only become a source in  $G(\mathcal{R}^{t+1})$  if either  $p_i$  or  $p_j$  dominates  $X$  (indeed, if the region  $X$  is already dominated by  $p_i^{xMax}$  or  $p_j^{yMax}$  then it cannot not exist in  $G(\mathcal{R}^t)$ ).

We consider the case where  $p_i$  dominates  $X$  (Figure 10.12(b)), the other case is symmetrical. If  $p_i$  dominates  $X$ , then  $X$  lies strictly left of the vertical line through  $p_i$ , and  $R_k$  intersects the vertical halfslab of  $X$ . Similarly if  $f_i(P) \neq R_k$  then  $R_k$  must lie at least partly right of the vertical line through  $p_i$  and below the bottom facet of  $f_i(P)$ . This means that if  $R_k$  lies in the vertical halfslab of  $X$  then it must also lie in the vertical halfslab of  $f_i(P)$ . The region  $f_i(P)$  is therefore a node in  $G(\mathcal{R}^t)$  with a directed path to  $R_k$ , so  $R_k$  is not a source node in  $G(\mathcal{R}^{t+1})$ . ■



**Figure 10.12** (a) The first case considered in Lemma 10.11, where  $g_j(P)$  precedes  $f_i(P)$ . In this case, all regions in  $[R_{i+1}, R_{j-1}]$  must be dominated by a point in  $P \setminus [p_{i+1}, p_{j-1}]$ . (b) The second case considered by the lemma. Suppose that after the  $t$ -th iteration, the region  $R_k$  loses the incoming arrow from  $X$ . Then there must be a directed path from  $f_i(P)$  or  $g_i(P)$  to  $R_k$ , or either  $R_k = f_i(P)$ ,  $R_k = g_j(P)$ .

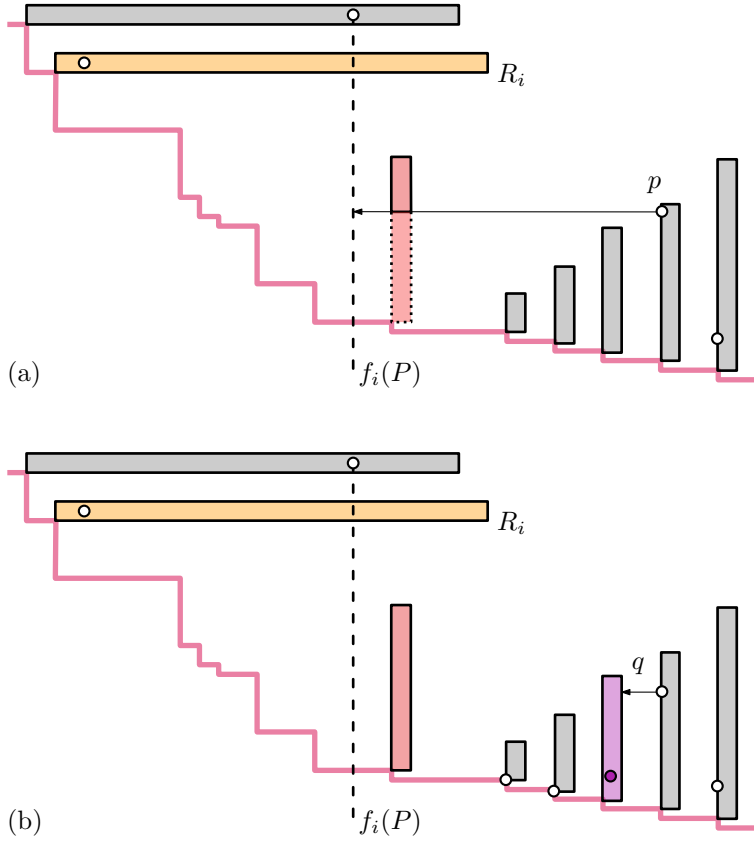
**Lemma 10.12** Let  $\mathcal{R}^t$  be any truncated set that is the result of a truncated set  $\mathcal{R}$ . Let  $[R_i, R_j]$  be a subproblem of  $G(\mathcal{R}^t)$  and denote by  $\mathcal{R}^{t+1}$  the set obtained by replacing  $R_i$  and  $R_j$  with  $p_i$  and  $p_j$  respectively, and re-truncating. Then the region  $f_i(P)$  is a source in  $G(\mathcal{R}^{t+1})$  if and only if:

- there is no region  $R_l \in \mathcal{R}$  for which  $f_i(P) \in H_l$ , or
- there is a region  $R_l$  with  $l > j$  which has a directed path to  $f_i(P)$  in  $G(\mathcal{R})$  and  $f_i(P) = g_l(P)$ .

**Proof** First, we show that if there is no region  $R_l \in \mathcal{R}$  for which  $f_i(P) \in H_l$ , then  $f_i(P)$  has to be a source in the graph  $G(\mathcal{R}^{t+1})$ . Indeed, per definition of  $H_l$  there can be no incoming horizontal arrow into  $f_i(P)$ . Via the same argument as the proof of Lemma 10.11, we note that the region  $f_i(P)$  can have no more incoming vertical arrows. Since after iteration  $t$  the region  $f_i(P)$  has no incoming horizontal or vertical arrows, it must be a source in  $G(\mathcal{R}^{t+1})$ .

Suppose that there is a region  $R_l$  with  $l > j$  which has a directed path to  $f_i(P)$  in  $G(\mathcal{R})$  and  $f_i(P) = g_l(P)$ . We show that in this case,  $f_i(P)$  has to be a source in the graph  $G(\mathcal{R}^{t+1})$ . Corollary 10.1, the point  $p_i$  must be in  $\mathcal{R}^t$  (else, the region  $R_j$  cannot be a source in  $G(\mathcal{R}^t)$ ). Hence, via the same argument as proof of Lemma 10.11, after truncation there can be no more incoming horizontal arrows into  $f_i(P)$  in  $G(\mathcal{R}^t)$ . Again, via the same argument as the proof of Lemma 10.11, we note that the region  $f_i(P)$  can have no more incoming vertical arrows and  $f_i(P)$  is thus a source.

Suppose finally that  $f_i(P)$  is a source. Moreover, suppose that there exists a region  $R_l \in \mathcal{R}$  for which  $f_i(P) \in H_l$ , and there is no region  $R_k$  with  $k > j$  and  $f_i(P) = g_k(P)$ , then there must be at least one region with a horizontal arrow into  $f_i(P)$  which contradicts the assumption that it is a source. ■



**Figure 10.13** (a) The first case of the proof of Lemma 10.12, where  $p$  must dominate the remaining regions with an arrow to  $f_i(P)$ . (b) The second case, where either  $q$  sees  $f_i(P)$  and dominates  $f_i(P)$ , or the purple region keeps its horizontal arrow to  $f_i(P)$ .

**Lemma 10.13** Let  $\mathcal{R}^t$  be any truncated set,  $\mathcal{R}^0 = \mathcal{R}^*$  contain  $\mathcal{R}^t$  and  $P$  be the underlying point set. Let  $[R_i, R_j]$  be a subproblem of  $G(\mathcal{R}^t)$  and denote by  $\mathcal{R}^{t+1}$  the set obtained by replacing  $R_i$  and  $R_j$  with  $p_i$  and  $p_j$  respectively, and re-truncating. We can identify the  $M$  subproblems of  $G(\mathcal{R}^{t+1})$  that are not subproblems of  $G(\mathcal{R}^t)$  in:

$$O(M + \log |V_i(P)| + \log |H_j(P)|) \text{ time.}$$

**Proof** Let  $v$  be the highest node in the tree  $T_{\mathcal{R}^*}$  that contains the interval  $[i, j]$ . Via Attribute 4 of  $\Xi_{\mathcal{R}}$ , we can identify  $v$  in constant time. By Corollary 10.2, we can obtain a pointer to  $f_i(P)$  and  $g_j(P)$  in  $O(\log |V_i(P)| + \log |H_j(P)|)$  time. Using that same exponential search, we identify the children  $[a, b]$  and  $[c, d]$  of  $v$  such that  $f_i(P) \in [R_a, R_b]$  and  $g_j(P) \in [R_c, R_d]$  in constant additional time.

We refer to subproblems of  $G(\mathcal{R}^{t+1})$  that are not already subproblems of  $G(\mathcal{R}^t)$  as *new* subproblems. By Lemma 10.12, the only regions in  $\mathcal{R}^{t+1}$  that are sources in  $G(\mathcal{R}^{t+1})$  but not of  $G(\mathcal{R}^t)$ , are  $f_i(P)$ ,  $g_j(P)$  and regions  $R_a$  and  $R_b$  for children  $[R_a, R_b]$  of  $v$ . Hence, all new subproblems must be contained in the subtree rooted at  $v$ . We show how to swiftly identify each of these subproblems using a case distinction based on  $f_i(P)$  and  $g_j(P)$  (the cases are illustrated by Figure 10.14):

**Case 0:**  $g_j(P)$  **precedes**  $f_i(P)$ . In this case, per definition all regions in  $[R_i, R_j]$  are dominated by a point in  $P$ . It follows that there are no subproblems in  $G(\mathcal{R}^{t+1})$  that are not already subproblems of  $G(\mathcal{R}^t)$  and  $M = 0$ . We can identify this case in constant time by checking the indices associated to  $f_i(P)$  and  $g_j(P)$  and thus we have identified the new subproblems in  $O(M + \log |V_i(P)| + \log |H_j(P)|)$  time. In the following we assume that  $f_i(P)$  precedes or equals  $g_j(P)$ .

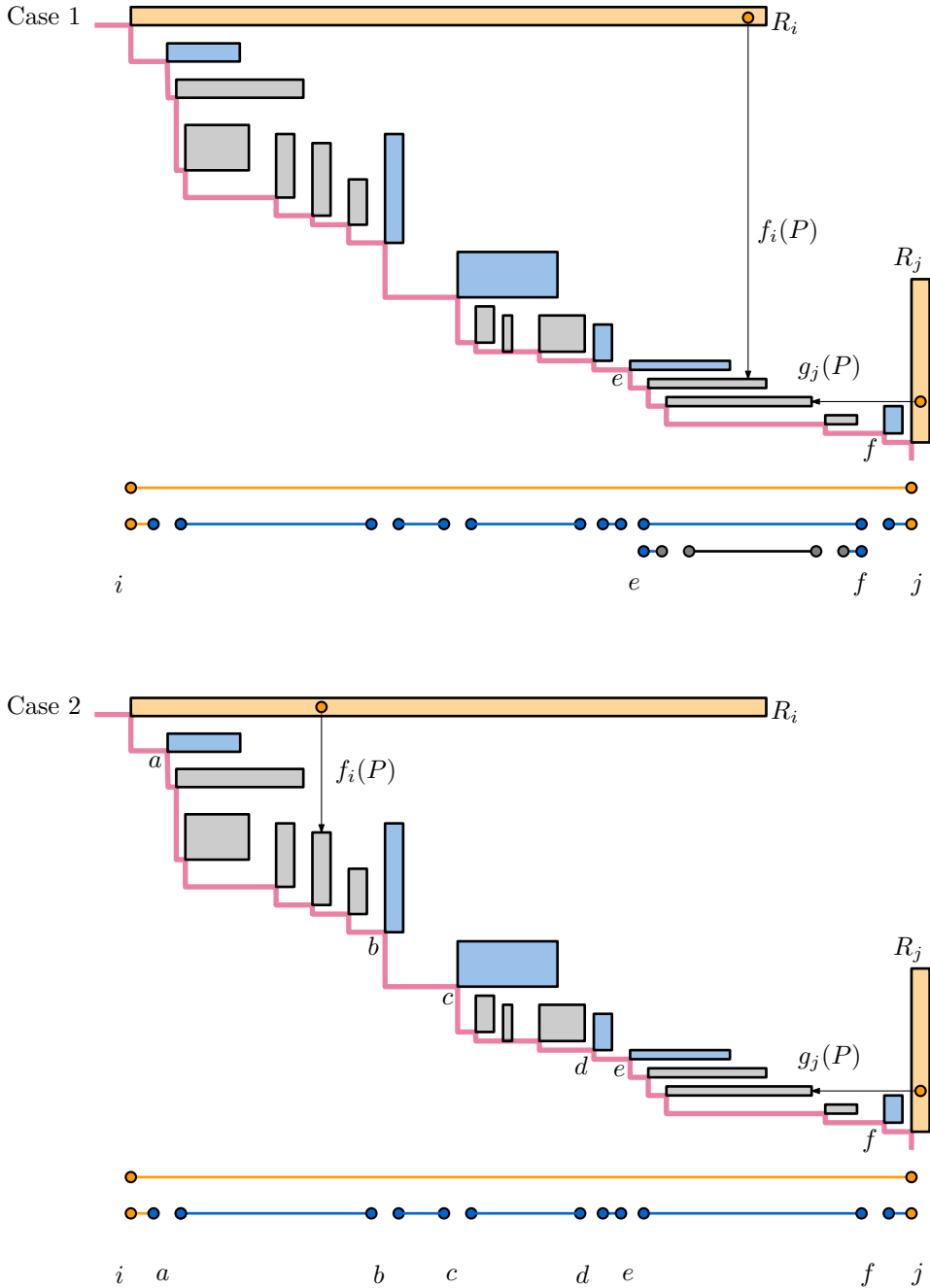
**Case 1:**  $f_i(P)$  and  $g_j(P)$  **are contained in the same child**  $[a, b]$  **of**  $v$ . Note that since  $R_i$  and  $R_j$  are both replaced with their respective points, either  $f_i(P)$  or  $g_j(P)$  must be a source in  $G(\mathcal{R}^{t+1})$ . By Lemma 10.12, We check in constant time whether  $f_i(P)$  and  $g_j(P)$  are sources in  $G(\mathcal{R}^{t+1})$  in constant time. Let  $R_k = f_i(P)$  and  $R_l = g_j(P)$ . We again make a case distinction:

1. If both  $f_i(P)$  and  $g_j(P)$  are sources, then by Lemma 10.11 the only three subproblems in  $G(\mathcal{R}^{t+1})$  and  $[R_i, R_j]$  are:  $[R_i = p_i, R_k]$ ,  $[R_k, R_l]$  and  $[R_l, R_j = p_j]$  hence  $M = 3$ .
2. If  $f_i(P)$  is a source and  $g_j(P)$  is not, by the same reasoning the only subproblems are  $[R_i, R_k]$  and  $[R_k, R_j]$  and  $M = 2$ .
3. If  $f_i(P)$  is not a source and  $g_j(P)$  the only subproblems are  $[R_i, R_k]$  and  $[R_k, R_j]$  and  $M = 2$ .

In all these cases, the number  $M$  of new subproblems is constant and we have thus identified the new subproblems in  $O(M + \log |V_i(P)| + \log |H_j(P)|)$  time.

**Case 2:**  $f_i(P) \in [R_a, R_b]$  **and**  $g_j(P) \in [R_e, R_f]$  **with**  $[a, b] \cap [e, f] = \emptyset$ . First, we show that there are at most two new subproblems contained in  $[R_a, R_b]$  and that we can identify these in constant time. Indeed, by Lemma 10.12, we can check in constant time if  $f_i(P)$  is a source. If it is, then by Lemma 10.11 the only subproblems of  $G(\mathcal{R}^{t+1})$  contained in  $[R_i, R_b]$  are  $[R_i, f_i(P)]$  and  $[f_i(P), R_b]$ . If  $f_i(P)$  is not a source, then by the same lemma,  $[R_i, R_b]$  is the only subproblem of  $G(\mathcal{R}^{t+1})$  in  $[R_i, R_b]$ . We apply a symmetric argument, and find the at most two new subproblems contained in  $[R_c, R_d]$  in constant time.

Per construction of  $T_{\mathcal{R}^*}$ , each child  $[c, d]$  of  $v$  with  $b \leq c < d \leq e$  is a subproblem of  $G(\mathcal{R}^{t+1})$ . Given a pointer to  $[a, b]$  and  $[c, d]$ , we can report these in constant time each, by iterating over the children of  $v$  in their sorted order. We conclude that we find and report the  $M$  new subproblems in  $O(M + \log |V_i(P)| + \log |H_j(P)|)$  time. ■



## 10.6.2 Finalising the reconstruction algorithm

Finally, we fully specify our algorithm for the reconstruction phase. We analyse its runtime using Lemma 10.9 and Lemma 10.13. Our input is a set of compounded regions  $\mathcal{R}^*$ , and an auxiliary structure  $\Xi_{\mathcal{R}}$  that originates from some original truncated set  $\mathcal{R}$ . The goal is to construct a labeled pointer structure, where each label corresponds to a point in  $P$ , that is isomorphic to the Pareto front of  $P$ .

We show an iterative algorithm where at every iteration  $t$ , we dequeue a subproblem  $[R_i, R_j]$  whilst we maintain an invariant which we briefly recall:

**Invariant 10.1** *For each iteration when we consider a subproblem  $[R_i, R_j]$ , we have a pointer to  $p_{i-1}^{xMax}$ , where this pointer is null if there is no point in  $[p_i, p_j]$  that can be dominated by a point preceding  $p_i$ . We have a pointer to  $p_{j+1}^{yMax}$  or null through a symmetric condition.*

**Setup before the algorithm starts.** Before the start of the algorithm we do the following (this is additional work that we perform in the preprocessing phase, where for ease of exposition, we delayed its description until now). In the preprocessing phase we have access to the compounded set  $\mathcal{R}^*$ . We set  $\mathcal{R}^0 = \mathcal{R}^*$ . Via the auxiliary structure  $\Xi_{\mathcal{R}}$ , we have access to the subproblem tree  $T_{\mathcal{R}^*}$  and the dependency graph  $G(\mathcal{R}^*)$ . We add for every child  $[i, j]$  of the root of  $T_{\mathcal{R}^*}$ , the subproblem  $[R_i, R_j]$  to the queue. Per construction, these are the subproblems of  $G(\mathcal{R}^*)$ . Moreover, by Corollary 10.1 for every such  $[R_i, R_j]$  there does not exist a point preceding  $p_i$  or succeeding  $p_j$  that dominates a point in  $[p_i, p_j]$ . Hence we set the pointers to  $p_{i-1}^{xMax}$  and  $p_{j+1}^{yMax}$  to null. It follows that after the preprocessing phase, Invariant 10.1 holds for all subproblems in the queue.

**A note about reporting points of the Pareto front.** We construct the structure  $\Xi_P^*$  (the maximal staircase graph where top vertices are regions  $R_i$  such that  $p_i$  is not dominated by a point in  $P$ ) by iteratively *reporting points* that lie on the Pareto front. Specifically, we note that at the start of the algorithm the subproblems in the queue partition  $\mathcal{R}$ . In further iterations, we consider a subproblem  $[R_i, R_j]$  and replace it with subproblems that partition  $[R_i, R_j]$  (thus, we maintain a partition of  $\mathcal{R}$ ). At all times, we assume that a subproblem  $[R_i, R_j]$  that we consider has a pointer to the subproblem immediately left, and immediately right of  $[R_i, R_j]$ . We maintain these pointers in additional constant time, when we insert the new subproblems in an iteration. Lemma 10.3 guarantees that for each iteration  $t$ , for each subproblem  $[R_i, R_j]$ , the Pareto front of  $[p_i, p_j]$  is a connected subchain of the Pareto front of  $P$ . Hence, given that either  $p_i$  or  $p_j$  is on the Pareto front we can simply append a reference to  $R_i$  or  $R_j$  after the rightmost known subproblem preceding  $R_i$  and before the leftmost subproblem succeeding  $R_j$  in constant time. We refer to this as *reporting points*. Since at all times the subproblems partition  $\mathcal{R}$ , the set of reported points form a linked list. The final structure  $\Xi_P^*$  is the linked list of all reported points.

**The full algorithm description.** For ease of exposition, we first assume that we do not encounter any compound regions. Start after all preprocessing with the set  $\mathcal{R}^0$  and  $t = 0$ . Continue iterating until the queue of subproblems is empty. At every iteration  $t$ , do the following:

1. Dequeue a subproblem  $[R_i, R_j]$  of  $G(\mathcal{R}^t)$  from the queue.
2. Retrieve  $p_i$  and  $p_j$  in  $O(C)$  time.
3. Check if  $p_i$  lies on the Pareto front of  $P$  through comparing it to  $p_j, p_{i-1}^{xMax}$  and  $p_{j+1}^{yMax}$ .  
If  $p_i$  lies on the Pareto front of  $P$ , report it in constant time. By Invariant 10.1 and Lemma 10.9, this can be done in  $O(1)$  time.
4. Check if  $p_j$  lies on the Pareto front (and possibly report it) in a similar manner in  $O(1)$  time.
5. Create  $\mathcal{R}^{t+1}$  by removing  $R_i$  and  $R_j$  and replacing them with  $p_i$  and  $p_j$  in  $O(1)$  time.
6. Identify the  $M$  ‘new’ subproblems of  $G(\mathcal{R}^{t+1})$  that are not already subproblems of  $G(\mathcal{R}^t)$ . Using Lemma 10.13 this takes  $O(M + \log |V_i(P)| + \log |H_j(P)|)$  time.
7. For any ‘new’ subproblem  $[R_a, R_b]$  with  $a = b - 1$ , we immediately retrieve  $p_a$  and  $p_b$  and check if these points appear on the Pareto front of  $P$  in  $O(C)$  additional time. By the proof of Lemma 10.13, for every such subproblem there is at least one unique ‘new’ subproblem  $[R_c, R_d]$  with  $c < d - 1$  and we charge this  $O(C)$  time spent to  $[R_c, R_d]$ .
8. For any ‘new’ subproblem  $[R_a, R_b]$  with  $a < b - 1$  we do the following:
  - (a) If  $f_i(P) \notin [R_a, R_b]$  then per construction of  $T_{\mathcal{R}^*}$  (see the proof of Lemma 10.13, Case 2) there cannot be a point preceding  $p_a$  that dominates a point in  $[R_a, R_b]$ .  
Hence we set  $p_{a-1}^{xMax}$  to *null*. Else, we select the rightmost point between  $p_i$  and  $p_{i-1}^{xMax}$  and set  $p_{a-1}^{xMax}$  as this point. All in  $O(1)$  time.
  - (b) We set  $p_{b+1}^{yMax}$  with a symmetric procedure in  $O(1)$  time.
  - (c) We add  $[R_a, R_b]$  to the queue and this maintains Invariant 10.1.

**Handling compound regions.** If at any iteration  $t$  where we process a subproblem  $[R_i, R_j]$  the region  $f_i(P)$  or  $g_j(P)$  is a compound region, we add a reference to  $p_i^{xMax}$  and  $p_j^{yMax}$  to the compound region.

Suppose that at iteration  $t$ , the algorithm encounters a subproblem  $[R_a, R_b]$  of  $G(\mathcal{R}^t)$  where  $R_a$  is a compound region (if  $R_b$  is a compound region, we follow a symmetric procedure). Then we encountered  $R_a$  in either step 1 or step 7 of the algorithm. Per definition,  $R_a$  is a sink in the original graph:  $G(\mathcal{R}^0)$ . Since not both  $R_a$  and  $R_b$  can be sinks in  $G(\mathcal{R}^t)$ , we continue as normal by retrieving  $p_b$ . In addition, we report  $R_a$  as part of the Pareto front. We add  $R_a$  to a separate queue with a reference to its reported location in the Pareto front.

When the algorithm ends, we process all compound regions in the queue. Note that for any subproblem encountered in step 1, either  $R_i$  or  $R_j$  is no compound region and we can charge the constant time needed to dequeue the compound region to the non-compound region. For any subproblem encountered in step 7, we can charge the constant time together with the  $O(C)$  time to the referenced subproblem  $[R_c, R_d]$ .

When we dequeue a compound region  $R$ , we assume that we have a reference to two points  $p_i^{xMax}$  and  $p_j^{yMax}$  for some integers  $i$  and  $j$ . We use  $p_i^{xMax}$  to identify the prefix of the original regions stored in  $R$  that are dominated by  $p_i^{xMax}$  in  $O(\log |V_i(P)|)$  time using exponential search. We charge the  $O(\log |V_i(P)|)$  time to the region  $R_i$ . Since every region  $R_i$  has at most one associated region  $f_i(P)$ , every region can get charged at most once. We process  $p_j^{yMax}$  similarly, and obtain the subsequence of  $R$  that is not dominated by any point in  $P$  as a linked list (Attribute 4 of the auxiliary structure  $\Xi_{\mathcal{R}}$ ) and we report this linked list in constant time by adding a reference to its starting point and end point in the Pareto front of  $P$ .

### 10.6.3 Proving the correctness

We prove that our algorithm correctly outputs the structure  $\Xi_P^*$  (the maximal staircase graph where top vertices are regions  $R_i$  such that  $p_i$  is not dominated by a point in  $P$ ) with a running time that matches the uncertainty region lower bound.

The correctness of our approach is almost immediately implied by Lemma 10.9.

**Theorem 10.4** *For any set of regions  $\mathcal{R}^*$ , with truncated set  $\mathcal{R}$  and compounded set  $\mathcal{R}^*$  and point set  $P$  the above algorithm constructs  $\Xi_P^*$ .*

**Proof** By Lemma 10.9, every point (or more precisely, every reference to a point in  $P$ ) that we append to our output, is a point on the Pareto front of  $P$ . By Lemma 10.3, all reported points get appended in their correct order along the Pareto front of  $P$ . What remains is to show that every point in the Pareto front gets appended to our output.

Let  $p$  be a point on the Pareto front of  $P$  and let  $p$  be contained in the region  $R \in \mathcal{R}^*$ . Per definition,  $p$  is not dominated by any point in  $P$  and hence  $R$  cannot be dominated by any point in  $P$ . Thus, there cannot be an iteration  $t$ , where we consider a subproblem  $[R_i, R_j]$  of  $G(\mathcal{R}^t)$ , where  $f_i(P)$  succeeds  $R$  or  $g_j(P)$  precedes  $R$ . Hence, there is at least one iteration where either the subproblem  $[R, \cdot]$  or  $[\cdot, R]$  is considered by the algorithm and thus,  $p$  is appended to the output. ■

What remains to show, is that the above algorithm matches the uncertainty-region lower bound.

**Theorem 10.5** *The above algorithm reconstructs  $\Xi_P^*$  in  $\Theta(\text{CP}(\mathcal{R}, P))$  time.*

**Proof** Recall that:

$$\text{CP}(\mathcal{R}, P) = \sum_{R_k \in \tilde{\mathcal{R}}(P)} C + \log |V_k(P)| + \log |H_k(P)|.$$

Each iteration  $t$ , our algorithm dequeues a subproblem  $[R_i, R_j]$  and spends  $O(C + \log |V_i| + \log |H_j|)$  time (all other time spent we charge to subproblems that get dequeued at some iteration  $t' > t$ ).

We note that it is possible that at iteration  $t$  we dequeue a subproblem  $[R_i, R_j]$  and at iteration  $t' > t$  a subproblem  $[R_i, R_k]$  with  $k < i$ . However, in this case, we do not have to spend  $O(C + \log |V_i|)$  time again to retrieve  $p_i$  and identify  $f_i(P)$ . Thus, if for every iteration  $t$ ,  $R_i, R_j \in \tilde{\mathcal{R}}(P)$  then the theorem trivially holds.

The argument, however, is a bit more subtle. We claim that for every iteration  $t$  where we dequeue  $[R_i, R_j]$  either  $R_i$  or  $R_j$  is in  $\tilde{\mathcal{R}}(P)$ . Moreover, if  $R_i \notin \tilde{\mathcal{R}}(P)$  (respectively  $R_j \notin \tilde{\mathcal{R}}(P)$ ) then it must be a sink in  $G(\mathcal{R})$  and thus  $\log |V_i| = 0$  (respectively  $\log |H_j| = 0$ ). Once we prove this claim, the theorem follows by charging the time spent each iteration to the region in  $\tilde{\mathcal{R}}(P)$ .

Note that we never queue a subproblem  $[R_a, R_b]$  where  $a = b - 1$ . So per construction  $i < j - 1$ . Next, we claim that both  $R_i$  and  $R_j$  are intersected by the Pareto front of  $P$ . Indeed, suppose for the sake of contradiction that  $R_i$  is not intersected by the Pareto front of  $P$  (the argument for  $R_j$  is symmetric). Then the region  $R_i$  must be dominated by a point  $p' \in P$ . Denote by  $p_k$  the point on the Pareto front of  $P$  that dominates  $R_i$  (via transitivity of domination, such a point  $p_k$  must always exist). The region  $R_k$  prevents  $R_i$  from being a sink: so  $p_k$  must be in  $\mathcal{R}^t$ . However, then  $R_i$  cannot be in  $\mathcal{R}^t$  after its truncation which is a contradiction.

It follows from the definition of  $\tilde{\mathcal{R}}(P)$  that  $R_i$  is not in  $\tilde{\mathcal{R}}(P)$  only if it is a sink in  $G(\mathcal{R}^t)$ . If  $[R_i, R_j]$  is a subproblem of  $G(\mathcal{R}^t)$  then they cannot both be sinks, so either  $R_i \in \tilde{\mathcal{R}}(P)$  or  $R_j \in \tilde{\mathcal{R}}(P)$ . If  $R_i$  is a sink, then per definition  $|V_i(P)| = |H_i(P)| = 1$  and the theorem follows. ■

By combining Theorem 10.2, Theorem 10.4 and Theorem 10.5, we conclude the main result of this chapter:

**Theorem 10.3** *Let  $\mathcal{R}^*$  be a set of  $m$  pairwise disjoint uncertainty rectangles in  $\mathbb{R}^2$  and denote by  $\mathcal{R} = \text{Trunc}(\mathcal{R}^*)$  its corresponding truncated set. We can preprocess  $\mathcal{R}^*$  in  $O(m \log m)$  time to reconstruct the Pareto front of the underlying point set  $P$  in  $\Theta(\text{CP}(\mathcal{R}, P))$  time, which is uncertainty-region optimal.*

## 10.7 Concluding remarks

In this chapter we studied how to efficiently construct the Pareto front of an imprecise point set. We showed how to preprocess a set  $\mathcal{R}$  of  $m$  pairwise disjoint uncertainty rectangles to create an auxiliary data structure  $\Xi_{\mathcal{R}}$  in  $O(m \log m)$  time. Given access to some underlying point set  $P$ , we can use  $\Xi_{\mathcal{R}}$  construct  $\Xi_P^*$  (the maximal staircase graph where top vertices are regions  $R_i$  such that  $p_i$  is not dominated by a point in  $P$ ) in  $\Theta(\text{CP}(\mathcal{R}, P))$  time. In some cases, for example when the algorithmic input  $\mathcal{R}^{\circ}$  is a truncated set and  $G(\mathcal{R}^{\circ})$  is the empty graph, the value  $\Theta(\text{CP}(\mathcal{R}, P)) = \Theta(1)$ . In this case, the required output of the preprocessing phase is the structure  $\Xi_P^*$  and this has a worst-case lower bound of  $\Omega(m \log m)$ . It follows from this observation that our preprocessing running time is worst-case optimal. It is not immediately clear how the runtime of our preprocessing can be improved as part of the preprocessing is sorting the regions from left-to-right. However, one possible way to improve the running time of the preprocessing phase is to see if it can be made output-sensitive. Specifically, let for the set of input regions  $\mathcal{R}^{\circ}$  there be  $M$  regions that intersect the Pareto front of the bottom left vertices of  $\mathcal{R}$  (the boundary  $\mathbb{B}_{\mathcal{R}^{\circ}}$ ). It may be possible to identify these regions in  $O(M \log M)$  time via an output-sensitive construction of the Pareto front. All  $(m - M)$  regions in this set are negative regions and can hence (potentially) be discarded in linear time.

Our reconstruction algorithm runs in

$$\Theta(\text{CP}(\mathcal{R}, P)) = \Theta \left( \sum_{R_k \in \bar{\mathcal{R}}(P)} C + \log |V_k(P)| + \log |H_k(P)| \right) \text{ time.}$$

By Lemma 10.5, for any pair  $(\mathcal{R}, P)$  there can be no algorithm that constructs the output  $\Xi_P^*$  in fewer than  $\Omega(\sum_{R_k \in \bar{\mathcal{R}}(P)} C)$  time. Moreover, for every region  $R_k$ ,  $\log |V_k(P)| + \log |H_k(P)| \leq 2 \log n$ . It follows that if the cost  $C$  of retrieving a point is at least  $\Omega(\log n)$  then our reconstruction algorithm is instance optimal. In addition, regardless of the value of  $C$ , there can be no algorithm that constructs  $\Xi_P^*$  in time faster than  $\Omega(\frac{1}{\log n} \text{CP}(\mathcal{R}, P))$ . If the cost  $C$  for retrieving a point is  $o(\log n)$ , Theorem 8.1 guarantees that there can exist no instance optimal algorithm. Hence, the result presented here can be considered ‘as good as it gets’.

The results of this chapter may be expanded in two ways. First, we note that we assumed that the cost of retrieving for a region  $R_i$  the point  $p_i$  is some fixed number of RAM instructions  $C$ . This assumption originated from the I/O-motivated setting of Bruce et al. [35]. We can try to expand our results by dropping the assumption that the retrieval cost is uniform (we assume that for every region  $R_i$ , there is some known cost  $C_i$  for retrieving  $p_i$ ). This setting matches more closely the motivation for the preprocessing model where the regions  $R_i$  represent an approximation of the value  $p_i$  as a result from some floating-point computation. The more elaborate the computation, the larger the cost  $C_i$ .

We wish to mention that, the result by Bruce et al. [35] already works in this extended setting. Using an unspecified number of RAM instructions, the authors iteratively identify a triple of regions  $(R_i, R_j, R_k)$  where one needs to retrieve at least either  $p_i$ ,  $p_j$  or  $p_k$  (it is not always clear beforehand, which of the three points needs to be retrieved). In their case, we can simply choose to first retrieve the point with the lowest retrieval cost. We then re-evaluate the triple to see if their corresponding Pareto front is now implied by the regions and the point and if not, retrieve the next. This scheme still gives a 3-approximation of the instance lower bound on the time spent on disk retrievals. This approach, however, is not immediately applicable to the algorithm presented in this chapter. Our algorithmic lower bound resembles that of Bruce et al. where we decompose the set  $\tilde{\mathcal{R}}(P)$  in such triples. However, in our algorithmic analysis of the running time of our algorithm we repeatedly retrieve points for regions  $R_i \notin \tilde{\mathcal{R}}(P)$  and charge that retrieval to a region  $R_j \in \tilde{\mathcal{R}}(P)$ . If the cost  $C_i$  is much greater than  $C_j$ , this charging becomes problematic. To remedy this, one would need a much more precise (and verbose) definition of the set of regions  $\tilde{\mathcal{R}}(P)$  for which the corresponding point needs to be retrieved. Second, one could try to extend our approach to a set of overlapping rectangles. In Chapter 9 we showed how to sort the underlying point set of a set of overlapping one-dimensional intervals. The difficulty that we encounter, when we try to extend our result to overlapping regions is that it becomes much more difficult to show an uncertainty-region lower bound. Specifically, it becomes more difficult to show the value  $|e(\mathcal{R})|$ : the number of combinatorially distinct Pareto fronts for point sets  $P$  that respect  $\mathcal{R}$ .





## Concluding remarks

This thesis is dedicated to tackling three distinct challenges within computational geometry, where each challenge corresponds to a distinct part of this thesis. Each chapter (with the exception of Chapters 1, 2, 3 and 8, which present both context and preliminary information) is dedicated to solving a concrete algorithmic or data structure problem. At the end of every such chapter, we present concluding remarks that review the obtained results and we remark upon possible future research directions. Here, we review the contribution of each part of this thesis. We briefly summarise the achievements obtained in this thesis, we remark upon its significance, and we review possible future research directions that relate to the part as a whole.

**In Part II** we studied the real RAM. In the introduction, we explained that the real RAM is a model of computation which assumes that arbitrary real-valued numbers can be compared and manipulated in constant time, using constant space. This assumption is regarded essential for providing algorithmic analysis within computational geometry. At the same time, this assumption is known to be false: real world computers may need significant time and space to compare two irrational real values. In Part II we investigated the real RAM and reason about the space (and thereby the time) required to perform the comparisons made by real RAM programs. The contribution of this part is twofold: first, we presented a rigorous definition of the real RAM and all its available operations. Second, we analysed this model of computation under slight random perturbations using smoothed analysis. We showed for a wide category of algorithmic results in computational geometry that the solutions can be executed on a word RAM using a logarithmic number of bits per variable. In addition, we showed how the square root operator may be included on a real RAM and we showed how to parametrize the usage of the square root operator to include that parameter in an upper bound on the number of bits per variable. The results that we obtained match the conjectured lower bounds in the Handbook of Discrete and Computational Geometry.

Part II of this thesis may contain the largest contribution to computational geometry as a whole, compared to the remaining parts of this thesis. Whilst real RAM definitions are plentiful, the definition in this thesis stands out in two ways. First, it contains extensive reasoning about the relation between real RAM and word RAM operations. Second, it contains an exhaustive list of the operations available to the word RAM and the real RAM. This allowed us to translate real RAM programs into their word RAM equivalents, which enabled us to reason about the number of bits required for each variable. The set of operations on our real RAM allowed a Cook-Levin-esque argument that shows the equivalence between algorithmic problems that lie in  $\exists\mathbb{R}$  and polynomial-time real RAM verification algorithms. Future work in computational geometry may refer to this definition whenever they require reassurance of the existence (or the non-existence) of certain operations within this model of computation. The second contribution of Part II is that the results presented here may alleviate the common criticism of real RAM algorithms. Oftentimes, the real RAM and associated algorithmic results are criticised to be theoretical in nature, as the algorithmic results may not be applicable to a word RAM (or a real-world computer). Future algorithmic research that is based on the real RAM may include amongst the results an upper bound on the size of the decision set of each algorithm. The theorems in Part II of this thesis then immediately provide an upper bound on the expected number of bits per variable and this upper bound, whenever it is polynomial (or even logarithmic) in the input size, may counter this criticism. There is reason to conjecture that the operations that our real RAM allows is the ‘largest set’ of operations that may be available to decidable real RAM programs. Specifically, we explicitly do not allow trigonometric operations and exponents of arbitrary input variables. The following may be additional motivation for this conjecture: currently, there is a one-to-one correspondence between existential real-valued expressions and real RAM programs. The famous Richardson’s theorem [176] shows that if the real-valued expression may include trigonometric operations, the expression (and thus, the corresponding program) may be undecidable (as of now it is unknown whether including exponents keeps real-valued expressions decidable). Thus, if we require our real RAM algorithms to be complete, we may indeed want to restrict ourselves to a real RAM without trigonometric operations and exponents. We hope that all the above argumentation serves as a solid foundation for our definition of the real RAM.

**In Part III** we studied three diverse data structure problems in computational geometry. In Chapter 5 we showed how to preprocess a polygonal domain such that it becomes possible to decide visibility between moving entities in sublinear time. In Chapter 6 we showed how to preprocess a polyline trajectory  $P$  to determine the Fréchet distance between  $P$  and an arbitrary line segment in sublinear time, and in Chapter 7 we showed how to dynamically maintain a smooth quadtree with worst case constant update time. In the respective chapters we reviewed how each of these individual results may be evaluated, extended or improved. Here, we highlight two over-arching future research directions.

In Chapter 5 we applied the ‘non-standard’ semi-algebraic range searching techniques to a ‘standard’ problem in computational geometry. In semi-algebraic range searching, one transforms a computational question into a  $d$ -dimensional halfspace range searching problem (which subsequently may be answered using cutting or partition trees). The downside of this technique is that at times, the dimension  $d$  may become relatively large (i.e. 4 or greater). When the dimension  $d$  is large, the cutting tree based solution may require high-polynomial (infeasible) space and preprocessing time and the partition tree based solution supports only queries with near-linear query time. This fact, combined with the fact that the algebraic analysis that is required to use this technique is sometimes considered cumbersome, makes semi-algebraic techniques not often applied. In this thesis, we successfully combined semi-algebraic range searching with techniques that are standard to computational geometry. Specifically, we showed how to compute the point of intersection between a convex polygon and an algebraic curve. We showed that most cases of intersection may be detected using classical geometric data structures. The remaining case can subsequently be solved using an algebraic predicate that, when linearized, has much smaller dimension than a predicate that decides if there is an intersection for all cases. We expect that the concept behind this approach may be applicable to more geometric problems, especially those that navigate some curved arrangement.

In Chapters 5 and 6 we first studied the data structures in a restricted setting that allows for an efficient solution. In both cases, we then generalize the result to either a polygonal domain, or arbitrarily oriented query segments. In both cases, the best we are able to do is essentially to consider a quadratic number of ‘subconfigurations’ (all paths between two segments in the polygonal domain, or all ‘combinatorially distinct’ segment orientations) and for each subconfiguration, solve the data structure problem separately. The consequence of this approach is, that in both cases the preprocessing time and required space get increased by a quadratic factor. It is possible that the upper bounds on the preprocessing time and space requirement is an over-estimation: consider for example all possible visibility glasses in a polygonal domain. Since the domain still contains  $n$  reflex vertices, either the amortized complexity of every visibility glass must be low or the visibility glasses must share many elements. Ideally, in such cases, one would find some global charging argument to construct a better upper bound on the total complexity.

**Part IV** of this thesis is dedicated to studying geometric problems subject to imprecise input. Specifically, we studied geometric problems in the preprocessing model where we first preprocess a set of regions  $\mathcal{R}$ , and then obtain the input  $P$  where every point  $p_i \in P$  is contained in a unique region  $R_i \in \mathcal{R}$ . We proposed an alteration of the classical definition of this model of computation, where the output is not a pointer structure on the point set  $P$  but rather a pointer structure on  $\mathcal{R}$  that is ‘isomorphic’ to the desired output on  $P$ . This alteration, combined with the added assumption that we may ‘refer’ to the output in constant time as soon as it is present in memory, allowed us to avoid the  $\Omega(n)$  time required to report the output and perform more fine-grained algorithmic analysis in this model of computation.

In addition, we defined a new form of algorithmic optimality that is native to this model of computation which we called uncertainty-region optimality. We showed that the granularity of this definition lies somewhere between the provably unobtainable instance optimality and the often-achieved worst case optimality. The results presented in Part IV are all uncertainty-region optimal.

Both Chapters 9 and 10 contain results that may be improved or generalized, as the concluding remarks of the respective chapters specify. Here, we note two directions for future research that transcend the individual results. Firstly, we note that Chapter 8 extensively reviewed the algebraic-decision tree argument that is often-used to obtain lower bounds. Afshani et al. [2] observe that general algebraic decision trees are more computationally powerful than the real RAM. We explained how our definition of the real RAM can be decomposed into an algebraic decision tree where nodes have outdegree 1 or 2, and where the complexity of the algebraic expression in the node of the tree depends linearly on the depth of that node. This construction provides us with a lower bound argument in a computational framework that is similar to the classically used algebraic decision tree, but not unrealistically powerful. Future research, separate of the preprocessing model, may use such ‘bounded complexity’ decision trees to argue for algorithmic lower bounds in scenarios where the classical algebraic decision trees are too strong. Finally, we note that there are many results obtained within the classical preprocessing model. Naturally, it is interesting to see if they can be adapted to function in the preprocessing model with indirect representations. Specifically, it is interesting to see if those results can be adapted to become uncertainty region optimal, as opposed to only worst case optimal. Out of all possible data structures that may be constructed in the preprocessing model, one thus far remains elusive: at this time, there is no satisfying result towards constructing the Delaunay triangulation of an imprecise point set. When constructing the Delaunay triangulation of an imprecise point set, it is not even evident what the desired output should be: consider the scenario where there are four cocircular pairwise disjoint uncertainty regions. The Delaunay triangulation of these four regions is a trapezoid crossed by a diagonal, but without retrieving all points it is unclear how the diagonal should cross the trapezoid. Do we, in this case, choose to demand that we resolve all such conflicts (and thus, require ourselves to possibly retrieve many points in  $P$ ) or do we strive for some triangulation-adjacent structure?

**The thesis as a whole** contains many contributions to computational geometry. Some are supplemental results that build upon existing work, some are tangential to previous work and explore new research directions, and some are fundamental in nature and may (ever so slightly) alter how we perceive algorithms and their analysis. All of them are open-ended with various alterations, deviations and augmentations to be explored and we expect follow-up work in the future.



# Samenvatting

Dit proefschrift bevat wetenschappelijk onderzoek in de computationele geometrie, een zijtak van de theoretische informatica. In theoretische informatica bestuderen we zogeheten computationele algoritmische problemen: problemen waarvan wij willen dat computers ze voor ons oplossen. Bekende voorbeelden van dit soort computationele problemen zijn het sorteren van data, het vinden van de kortste route van A naar B, en het berekenen van efficiënt tijdsrooster (van bijvoorbeeld de lessen op een school). In de theoretische informatica werken we met een theoretisch model van een computer waarmee we redeneren we over de stappen die een computer moet ondernemen om zulke problemen op te lossen. De kracht van de theoretische informatica is dat we bepaalde eigenschappen van de theoretische programma's kunnen *bewijzen*. Er bestaat bijvoorbeeld een algoritme (programma) waarvan we kunnen bewijzen dat het *altijd* correct het kortste pad van A naar B berekent. In de computationele geometrie bestuderen we computationele problemen die van geometrische aard zijn. Dat betekent dat het onderwerp van het probleem (de *input* van het programma) een geometrisch object is. Deze input kan bijvoorbeeld een puntenwolk zijn (lees: een set aan GPS-locaties), een eenvoudige veelhoek (de lay-out van een huis), of een collectie van rechthoeken (de lay-out van de gebouwen in een stad). Om een gevoel te geven voor het soort computationele problemen dat men bestudeert, presenteren we drie bekende computationele (algoritmische) problemen:

- Gegeven een puntenwolk, bereken de twee dichtstbijzijnde punten (nearest neighbor searching),
- Gegeven een eenvoudige veelhoek en twee punten  $a$  en  $b$  in de veelhoek, vind het kortste pad van  $a$  naar  $b$  (shortest path in a simple polygon),
- Gegeven een collectie van rechthoeken, vind de kleinste cirkel die alle rechthoeken bevat (diameter of a geometric set).

Dit proefschrift is een verzameling van onderzoek naar de geometrische informatie die als invoer dient voor computationele problemen. We bekijken een theoretisch model van een computer en bestuderen hoe dit model omgaat met deze geometrische invoer. Het eerste deel van dit proefschrift is de introductie, waarin we de onderzoeksvragen uiteenzetten. In de rest van het proefschrift onderzoeken we drie onderzoeksvragen, die elk hun eigen deel hebben van dit proefschrift.

**Het modelleren van geometrische informatie: Real RAM berekeningen.** In de computationele geometrie gebruiken we een bijzonder theoretisch model van een computer: de Real RAM. Een RAM is een theoretisch model dat de innerlijke werking van een computer abstraheert. Een RAM modelleert een computer met twee conceptuele componenten:

- De eerste component is het geheugen van de computer. De RAM modelleert het geheugen als een lijst aan geheugenadressen, waar op elk adres één getal kan worden opgeslagen. Stel dat de invoer van een computationeel probleem een collectie van  $n$  cirkels is. Een cirkel bestaat uit een centraal punt (met twee coördinaten) en een radius. Elke cirkel kan dus worden opgeslagen met drie geheugenadressen, en de collectie van cirkels met  $3n$  geheugenadressen.
- Het tweede component is de processor van de computer. De RAM modelleert de werking van een computer op het geheugen als een collectie aan instructies, waarbij elke instructie een operatie uitvoert op één of meer geheugenadressen. Voorbeelden zijn: het ophogen van een getal in een geheugenadres, het optellen van twee getallen uit twee geheugenadressen en het vergelijken van de twee getallen in twee geheugenadressen.

Een daadwerkelijke computer heeft maar een beperkt geheugen. Bijvoorbeeld, de computer waarop ik dit proefschrift maak heeft een werkgeheugen van 16 GB. De RAM modelleert dit aspect van een computer door een grens te zetten op de precisie van elk getal: men kiest een getal  $w$  en eist dat voor elk geheugenadres, het getal in het geheugenadres gerepresenteerd kan worden met  $w$  bits (Rekenvoorbeeld: stel we kiezen  $w = 16$  bytes, dan heeft mijn computer maximaal 1.000.000.000 geheugenadressen beschikbaar. Ik zou in dit geval dus maximaal 333.333.333 cirkels op mijn computer kunnen opslaan). Als je tijdens je berekening een getal nodig hebt dat meer precisie vereist dan dat aanwezig is op één geheugenadres heb je vervolgens twee keuzes: of je gebruikt meerdere geheugenadressen, of je gaat afronden.

In de computationele geometrie werken we vaak met reële getallen die voortkomen uit de geometrische setting  $(\pi, \sqrt{2}, \phi)$ . Reële getallen zijn getallen waarvoor je oneindig veel geheugen nodig hebt om ze op te schrijven ( $\pi = 3.14159265358979 \dots$ ). Om onze theoretische analyse te vereenvoudigen werken we daarom in de computationele geometrie in een bijzondere variant van de RAM: de real RAM. Bij de real RAM nemen we aan dat elk geheugenadres in staat is om één reëel getal op te slaan, met oneindig veel precisie. Dus het getal  $\pi$  kan terecht op exact één geheugenadres.

De collectie aan instructies die we loslaten op dit geheugen blijft (nagenoeg) gelijk aan die van de klassieke RAM, waarbij we aannemen dat elke operatie wordt uitgevoerd met oneindige precisie (bijvoorbeeld: we nemen aan dat optellen van  $\sqrt{2}$  bij  $\sqrt{3}$  gewoon mogelijk is). Deze theoretische aanname is gebruikelijk, maar niet vanzelfsprekend. We weten immers uit te praktijk dat een daadwerkelijke computer zeer beperkt geheugen heeft! In Deel 2 van het proefschrift onderzoeken we deze ‘real RAM’ aanname en de gevolgen daarvan. We tonen aan dat de aanname van oneindige precisie zo gek nog niet is: we bewijzen dat met grote kans (lees: meestal) een berekening met oneindige precisie dezelfde uitvoer geeft als een berekening op een begrensde RAM.

**Het structureren van geometrische informatie: datastructuren.** Nadat we hebben gekeken naar de werking van geometrische berekeningen, onderzoeken we de efficiëntie van bepaalde berekeningen. In het bijzonder kijken we naar mogelijkheden om geometrische informatie te structureren voor hergebruik. Bekijk het volgende computationele probleem: gegeven is een collectie van GPS-locaties  $S$  (een voorbeeld van zo’n collectie  $S$  is de kaart waarop Google alle supermarkten in Utrecht laat zien) en een punt  $p$  (jouw locatie in Utrecht), vindt de dichtstbijzijnde locatie in  $S$ . Het beste wat een computer kan doen om dit probleem op te lossen is het volgende: bekijk alle locaties in  $S$  en onthoudt degene met de kleinste afstand tot  $p$ . Omdat we elke locatie in  $S$  bekijken, noemen we de tijdsduur van deze oplossing *lineair*. Maar stel nu dat ik niet alleen voor mijzelf de dichtstbijzijnde supermarkt wil weten, maar ook voor al mijn vrienden. Ik zou per vriend  $v$  hetzelfde lineaire programma kunnen draaien: bekijk alle supermarkten, en onthoudt de dichtstbijzijnde bij  $v$ . Echter, als ik weet dat ik dezelfde vraag meerdere keren ga stellen dan kan ik wat slimmer doen door informatie beter te hergebruiken: als  $v$  bijvoorbeeld dicht bij mij woont, dan hoef ik voor  $v$  alleen maar de supermarkten dicht bij mijzelf te overwegen. Door de geometrische data eerst slim te structureren, wordt het mogelijk om per vriend  $v$ , de uitvoer sneller (sublineair) te produceren. In Deel 3 van het proefschrift kijken we naar drie concrete computationele problemen, en hoe we geometrische informatie kunnen structureren voor efficiënt hergebruik:

- **Wederzijds zicht berekenen.** Gegeven is een simpele veelhoek (de lay-out van een kamer of huis). We tonen aan dat we de veelhoek kunnen structureren voor efficiënte gezichtsveld-vragen. We structureren de veelhoek zodanig dat, gegeven een paar mensen die door de veelhoek (kamer) loopt, we efficiënt kunnen bepalen of er een tijdstip is waarop de twee mensen elkaar kunnen zien.
- **Fréchet afstand tussen verschillende trajecten.** Gegeven is een traject (een serie aan punten, locaties, die door een bewegende entiteit worden doorlopen). We tonen aan dat we het traject kunnen structureren voor efficiënte afstandsvergelijkingen. We structureren het traject zodanig dat, gegeven een lijnsegment dat wordt doorlopen door een bewegende entiteit, we efficiënt de Fréchet afstand tussen het traject en het lijnsegment kunnen berekenen.

- Dynamische gladde gecomprimeerde quadbomen. Elk vierkant kan worden gesplitst in vier vierkanten van gelijke grootte. Elk van deze vierkanten kan vervolgens weer worden gesplitst in vier kleinere vierkanten van gelijke grootte. De structuur die ontstaat door een reeks van zulke splitsingen heet een quadboom. Wanneer vierkanten op deze wijze recursief worden gesplitst kan het voorkomen dat er hele kleine vierkanten aangrenzend aan hele grote vierkanten ontstaan. Gladde quadbomen zijn quadbomen met de extra eis dat er geen kleine vierkanten naast grote vierkanten mogen grenzen. Wanneer een splitsing deze situatie veroorzaakt, eist de gladde quadboom dat het grote vierkant ook wordt gesplitst. We tonen aan hoe men in een quadboom op slimme wijze kan splitsen, zodat men te allen tijde een gladde quadboom heeft *en* men per splitsing maximaal een constante factor aan extra werk moet verrichten.

**Het vastleggen van geometrische informatie.** In Deel 2 van het proefschrift merken we op dat computers werken met eindige precisie: in plaats van een reëel getal  $r$ , moet een daadwerkelijke computer rekenen met een afronding  $r'$ . Echter, in veel gevallen is het mogelijk om, met een bepaalde computationele prijs, een preciezere representatie van  $r'$  te bemachtigen. Deze mogelijkheid roept de volgende vraag op: wanneer is het nodig om van een inaccurate waarde  $r'$ , een preciezere representatie te berekenen? Deze vraag bestuderen we in Deel 4 van het proefschrift in twee verschillende contexten:

- Het sorteren van inaccurate waarden. Gegeven is een set aan inaccurate waarden: een set aan intervallen  $\mathcal{I}$  als model voor een onbekende waardencollectie  $P$  waarbij er voor elk interval  $I_i \in \mathcal{I}$  er een onbekende exacte waarde  $p_i \in I_i$  bestaat. Voor elke inaccurate waarde  $I_i$  kunnen we de bijbehorende  $p_i$  opvragen voor een vooraf bepaalde prijs:  $C$ . We tonen aan hoe we de gesorteerde volgorde van  $P$  kunnen construeren met een minimale totaalprijs, en een minimaal aantal vergelijkingen.
- Het berekenen van een inaccurate Pareto grens. Gegeven is een set aan inaccurate tweedimensionale punten: een set aan (niet-overlappende) rechthoeken  $\mathcal{R}$  als model voor een onbekende waardencollectie  $P$  waarbij voor elke rechthoek  $R_i \in \mathcal{R}$  er een onbekende exacte waarde  $p_i \in R_i$  bestaat. We tonen aan hoe we Pareto grens (de subset van maximale punten in  $P$ ) kunnen construeren met een minimale totaalprijs, en een minimaal aantal vergelijkingen.



## Bibliography

- [1] Mohammad Abam, Mark de Berg, Sina Farahzad, Mir-Omid Haji-Mirsadeghi, and Morteza Saghafian. Preclustering Algorithms for Imprecise Points. *Scandinavian Symposium and Workshops on Algorithm Theory (SWAT)*, 2020.
- [2] Peyman Afshani, Jérémy Barbay, and Timothy Chan. Instance-optimal Geometric Algorithms. *Journal of the ACM (JACM)*, 2017.
- [3] Pankaj Agarwal. Ray Shooting and Other Applications of Spanning Trees with Low Stabbing Number. *SIAM Journal on Computing (SICOMP)*, 1992.
- [4] Pankaj Agarwal and Jiří Matoušek. Dynamic half-space range reporting and its applications. *Algorithmica*, 1995.
- [5] Pankaj Agarwal, Jiří Matoušek, and Micha Sharir. On Range Searching with Semialgebraic Sets II. *SIAM Journal on Computing (SICOMP)*, 2013.
- [6] Pankaj K Agarwal, Sariel Har-Peled, Nabil H Mustafa, and Yusu Wang. Near-linear time approximation algorithms for curve simplification. *Algorithmica*, 2005.
- [7] Alok Aggarwal, Leonidas Guibas, James Saxe, and Peter Shor. A Linear-Time Algorithm for Computing the Voronoi Diagram of a Convex Polygon. *Discrete & Computational Geometry (DCG)*, 1989.
- [8] Helmut Alt, Alon Efrat, Günter Rote, and Carola Wenk. Matching planar maps. *Journal of algorithms*, 2003.
- [9] Helmut Alt and Michael Godau. Computing the Fréchet Distance Between Two Polygonal Curves. *International Journal of Computational Geometry & Applications (IJCGA)*, 1995.

- [10] Helmut Alt, Christian Knauer, and Carola Wenk. Matching Polygonal Curves with Respect to the Fréchet Distance. *Symposium on Theoretical Aspects of Computer Science (STACS)*, 2001.
- [11] Boris Aronov, Prosenjit Bose, Erik Demaine, Joachim Gudmundsson, John Iacono, Stefan Langerman, and Michiel Smid. Data Structures for Halfplane Proximity Queries and Incremental Voronoi Diagrams. *Latin American Symposium on Theoretical Informatics (LATIN)*, 2006.
- [12] Boris Aronov, Hervé Brönnimann, Allen Y Chang, and Yi-Jen Chiang. Cost prediction for ray shooting in octrees. *Computational Geometry*, 2006.
- [13] Boris Aronov, Leonidas Guibas, Marek Teichmann, and Li Zhang. Visibility Queries and Maintenance in Simple Polygons. *Discrete and Computational Geometry (DCG)*, 2002.
- [14] Sanjeev Arora and Boaz Barak. *Computational Complexity: a Modern Approach*. Cambridge University Press, 2009.
- [15] Deepak Bandyopadhyay and Jack Snoeyink. Almost-Delaunay Simplices: Robust Neighbor Relations for Imprecise 3D Points Using CGAL. *Computational Geometry*, 2007.
- [16] Saugata Basu, Richard Pollack, and Marie-Françoise Roy. *Algorithms in Real Algebraic Geometry*. Springer, Berlin Heidelberg, 2006.
- [17] René Beier and Berthold Vöcking. Random Knapsack in Expected Polynomial Time. *ACM Symposium on Theory of Computing (STOC)*, 2003.
- [18] Michael Ben-Or. Lower Bounds for Algebraic Computation Trees. *ACM Symposium on Theory of Computing (STOC)*, 1983.
- [19] Michael Bender and Martín Farach-Colton. The LCA Problem Revisited. *Latin American Symposium on Theoretical Informatics (LATIN)*, 2000.
- [20] Marc Benkert, Joachim Gudmundsson, Florian Hübner, and Thomas Wolle. Reporting flock patterns. *Computational Geometry*, 2008.
- [21] Huck Bennett, Evanthia Papadopoulou, and Chee Yap. Planar minimization diagrams via subdivision with applications to anisotropic Voronoi diagrams. *Computer Graphics Forum*, 2016.
- [22] Huck Bennett and Chee Yap. Amortized Analysis of Smooth Quadrees in all Dimensions. *Computational Geometry*, 2017.
- [23] Marshall Bern, David Dobkin, David Eppstein, and Robert Grossman. Visibility with a Moving Point of View. *Algorithmica*, 1994.
- [24] Marshall Bern, David Eppstein, and John Gilbert. Provably Good Mesh Generation. *Journal of Computer and System Sciences (JCSS)*, 1994.

- [25] Marshall Bern, David Eppstein, and Shang-Hua Teng. Parallel construction of quadtrees and quality triangulations. *International Journal of Computational Geometry & Applications (IJCGA)*, 1999.
- [26] Alberto Bertoni, Giancarlo Mauri, and Nicoletta Sabadini. Simulations Among Classes of Random Access Machines and Equivalence Among Numbers Succinctly Represented. *Annals of Discrete Mathematics*, 1985.
- [27] Markus Bläser, Bodo Manthey, and Raghavendra Rao. Smoothed Analysis of Partitioning Algorithms for Euclidean Functionals. *Algorithmica*, 2013.
- [28] Lenore Blum, Felipe Cucker, Michael Shub, and Steve Smale. *Complexity and Real Computation*. Springer, 1998.
- [29] Lenore Blum, Mike Shub, and Steve Smale. On a Theory of Computation and Complexity over the Real Numbers: *NP*-completeness, Recursive Functions and Universal Machines. *Bulletin of the American Mathematical Society*, 1989.
- [30] Jean-Daniel Boissonnat, Olivier Devillers, Sylvain Pion, Monique Teillaud, and Mariette Yvinec. Triangulations in CGAL. *Computational Geometry*, 2002.
- [31] Daniel Bovet and Simon Benhamou. Spatial analysis of animals' movements using a correlated random walk model. *Journal of Theoretical Biology*, 1988.
- [32] Graham Brightwell and Peter Winkler. Counting Linear Extensions. *Order*, 1991.
- [33] Karl Bringmann and Bhaskar Chaudhury. Polyline Simplification has Cubic Complexity. *Journal of Computational Geometry (JoCG)*, 2021.
- [34] Gerth Stølting Brodal and Riko Jacob. Dynamic planar convex hull. *IEEE Symposium on Foundations of Computer Science (FOCS)*, 2002.
- [35] Richard Bruce, Michael Hoffmann, Danny Krizanc, and Rajeew Raman. Efficient Update Strategies for Geometric Computing with Uncertainty. *Theory of Computing Systems (TOCS)*, 2005.
- [36] Kevin Buchin, Maike Buchin, Joachim Gudmundsson, Maarten Löffler, and Jun Luo. Detecting commuting patterns by clustering subtrajectories. *International Journal of Computational Geometry & Applications (IJCGA)*, 2011.
- [37] Kevin Buchin, Maike Buchin, Marc van Kreveld, Maarten Löffler, Rodrigo I Silveira, Carola Wenk, and Lionov Wiratma. Median trajectories. *Algorithmica*, 2013.
- [38] Kevin Buchin, Maarten Löffler, Pat Morin, and Wolfgang Mulzer. Preprocessing imprecise points for Delaunay triangulation: Simplified and extended. *Algorithmica*, 2011.

- [39] Kevin Buchin, Maarten Löffler, Pat Morin, and Wolfgang Mulzer. Preprocessing Imprecise Points for Delaunay Triangulation: Simplified and Extended. *Algorithmica*, 2011.
- [40] Kevin Buchin and Wolfgang Mulzer. Delaunay Triangulations in  $O(\text{sort}(n))$  Time and More. *Journal of the ACM (JACM)*, 2011.
- [41] Kevin Buchin, Stef Sijben, Emiel van Loon, Nir Sapir, Stéphanie Mercier, Jean Arseneau, and Erik Willems. Deriving movement properties and the effect of the environment from the Brownian bridge movement model in monkeys and birds. *Movement ecology*, 2015.
- [42] Christoph Burnikel, Rudolf Fleischer, Kurt Mehlhorn, and Stefan Schirra. Efficient exact geometric computation made easy. *International Symposium on Computational Geometry (SoCG)*, 1999.
- [43] Daniel Busto, William Evans, and David Kirkpatrick. Minimizing Interference Potential Among Moving Entities. *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2019.
- [44] Paul Callahan and Rao Kosaraju. A decomposition of multidimensional point sets with applications to  $k$ -nearest-neighbors and  $n$ -body potential fields. *Journal of the ACM (JACM)*, 1995.
- [45] Jean Cardinal, Timothy Chan, John Iacono, Stefan Langerman, and Aurélien Ooms. Subquadratic Encodings for Point Configurations. *International Symposium on Computational Geometry (SoCG)*, 2018.
- [46] Jean Cardinal, Ruy Fabila-Monroy, and Carlos Hidalgo-Toscano. Chirotopes of Random Points in Space are Realizable on a Small Integer Grid. *Canadian Conference on Computational Geometry (CCCG)*, 2019.
- [47] Jean Cardinal, Samuel Fiorini, Gwenaél Joret, Raphaël Jungers, and Ian Munro. Sorting under Partial Information (without the Ellipsoid Algorithm). *Combinatorica*, 2013.
- [48] Jean Cardinal and Udo Hoffmann. Recognition and Complexity of Point Visibility Graphs. *Discrete & Computational Geometry (DCG)*, 2017.
- [49] Timothy Chan. Comparison-based Time-space Lower Bounds For Selection. *ACM Transactions on Algorithms (TALG)*, 2010.
- [50] Timothy Chan. Optimal partition trees. *Discrete and Computational Geometry (DCG)*, 2012.
- [51] Bernard Chazelle. Cutting hyperplanes for divide-and-conquer. *Discrete and Computational Geometry (DCG)*, 1993.

- [52] Bernard Chazelle, Herbert Edelsbrunner, Michelangelo Grigni, Leonidas Guibas, John Hersberger, Micha Sharir, and Jack Snoeyink. Ray shooting in polygons using geodesic triangulations. *Algorithmica*, 1994.
- [53] Bernard Chazelle and Leonidas Guibas. Visibility and Intersection Problems in Plane Geometry. *Discrete and Computational Geometry (DCG)*, 1989.
- [54] Hue-Ling Chen and Ye-In Chang. All-nearest-neighbors finding based on the Hilbert curve. *Expert Systems with Applications*, 2011.
- [55] Vasek Chvátal. A combinatorial theorem in plane geometry. *Journal of Combinatorial Theory (JCTB)*, 1975.
- [56] David Cox, John Little, and Donal O'Shea. *Using Algebraic Geometry*. Springer Science & Business Media, 2006.
- [57] David Cox, John Little, and Donal O'Shea. *Ideals, Varieties and Algorithms*. Springer, 2007.
- [58] Daniel Dadush and Sophie Huiberts. A Friendly Smoothed Analysis of the Simplex Method. *ACM Symposium on Theory of Computing (STOC)*, 2018.
- [59] Arun Kumar Das, Sandip Das, and Joydeep Mukherjee. Largest triangle inside a terrain. *Theoretical Computer Science (TCS)*, 2021.
- [60] Mark de Berg. *Ray shooting, depth orders and hidden surface removal*. Springer Science & Business Media, 1993.
- [61] Mark de Berg, Otfried Cheong, Marc Van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications, third edition*. Springer, 2008.
- [62] Mark de Berg, Atlas Cook IV, and Joachim Gudmundsson. Fast Fréchet queries. *Computational Geometry*, 2013.
- [63] Mark de Berg, Ali Mehrabi, and Tim Ophelders. Data Structures for Fréchet Queries in Trajectory Data. *Canadian Conference on Computational Geometry (CCCG)*, 2017.
- [64] Mark de Berg, Marcel Roeloffzen, and Bettina Speckmann. Kinetic compressed quadrees in the black-box model with applications to collision detection for low-density scenes. *European Symposium on Algorithms (ESA)*, 2012.
- [65] Mark de Berg, Farnaz Sheikhi, Ali Mohades, and Ali Mehrabi. Separability of Imprecise Points. *Computational Geometry*, 2017.
- [66] Leila De Floriani and Paola Magillo. Algorithms for visibility computation on terrains: a survey. *Environment and Planning B: Urban Analytics and City Science*, 2003.

- [67] Erik Demaine, Adam Hesterberg, and Jason Ku. Finding Closed Quasigeodesics on Convex Polyhedra. *Symposium on Computational Geometry (SoCG)*, 2020.
- [68] Olivier Devillers. Delaunay Triangulation of Imprecise Points: Preprocess and Actually get a Fast Query Time. *Journal of Computational Geometry (JoCG)*, 2011.
- [69] Olivier Devillers, Philippe Duchon, Marc Glisse, and Xavier Goaoc. On Order Types of Random Point Sets. *ArXiv preprint:1812.08525*, 2018.
- [70] Yago Diez, Matias Korman, André van Renssen, Marcel Roeloffzen, and Frank Staals. Kinetic All-Pairs Shortest Path in a Simple Polygon. *European Workshop on Computational Geometry (EuroCG)*, 2017.
- [71] David Dobkin and David Kirkpatrick. Fast detection of polyhedral intersection. *International Colloquium on Automata, Languages, and Programming (ICALP)*, 1983.
- [72] David Dobkin and Lawrence Snyder. On a general method for maximizing and minimizing among certain geometric problems. In *IEEE Symposium on Foundations of Computer Science (SFCS)*, 1979.
- [73] Somayeh Dodge, Robert Weibel, and Ehsan Forootan. Revealing the physics of movement: Comparing the similarity of movement characteristics of different types of moving objects. *Computers, Environment and Urban Systems*, 2009.
- [74] Anne Driemel and Sarel Har-Peled. Jaywalking Your Dog: Computing the Fréchet Distance with Shortcuts. *SIAM Journal on Computing (SICOMP)*, 2013.
- [75] Anne Driemel and Ioannis Psarros.  $(2 + \epsilon)$ -ANN for Time Series under the Fréchet Distance. *Workshop on Algorithms and Data Structures (WADS)*, 2021.
- [76] James Driscoll, Neil Sarnak, Daniel Sleator, and Robert Tarjan. Making data structures persistent. *Journal of Computer and System Sciences (JCSS)*, 1989.
- [77] Frédo Durand. A Multidisciplinary Survey of Visibility. *ACM Special Interest Group on Computer Graphics and Interactive Techniques (SIGGRAPH)*, 2000.
- [78] Herbert Edelsbrunner, Leo Guibas, and Jorge Stolfi. Optimal Point Location in a Monotone Subdivision. *SIAM Journal on Computing (SICOMP)*, 1986.
- [79] Matthias Englert, Heiko Röglin, and Berthold Vöcking. Worst Case and Probabilistic Analysis of the 2-Opt Algorithm for the TSP. *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2007.
- [80] David Eppstein, Michael Goodrich, and Jonathan Sun. The skip quadtree: a simple dynamic data structure for multidimensional data. *International Symposium on Computational Geometry (SoCG)*, 2008.
- [81] Jeff Erickson. Lower Bounds for Linear Satisfiability Problems. *Chicago Journal of Theoretical Computer Science*, 1997.

- [82] Jeff Erickson. Algorithms. 1999.
- [83] Jeff Erickson, Ivor van der Hoog, and Tillmann Miltzow. Smoothing the Gap between  $NP$  and  $ER$ . *IEEE Symposium on Foundations of Computer Science (FOCS)*, 2020.
- [84] Michael Etscheid and Heiko Röglin. Smoothed Analysis of Local Search for the Maximum-cut Problem. *ACM Transactions on Algorithms (TALG)*, 2017.
- [85] William Evans, David Kirkpatrick, Maarten Löffler, and Frank Staals. Competitive Query Strategies for Minimising the Ply of the Potential Locations of Moving Points. *International Symposium on Computational Geometry (SoCG)*, 2013.
- [86] William Evans, David Kirkpatrick, Maarten Löffler, and Frank Staals. Minimizing Co-location Potential of Moving Entities. *SIAM Journal on Computing (SICOMP)*, 2016.
- [87] William Evans and Jeff Sember. The Possible Hull of Imprecise Points. *Canadian Conference on Computational Geometry (CCCG)*, 2011.
- [88] Esther Ezra and Wolfgang Mulzer. Convex Hull of Points Lying on Lines in  $o(n \log n)$  Time after Preprocessing. *Computational Geometry*, 2013.
- [89] Arnold Filtser, Omrit Filtser, and Matthew Katz. Approximate Nearest Neighbor for Curves – Simple, Efficient, and Deterministic. *International Colloquium on Automata, Languages, and Programming (ICALP)*, 2020.
- [90] Raphael Finkel and Jon Bentley. Quad Trees: A Data Structure for Retrieval on Composite Keys. *Acta Informatica*, 1974.
- [91] Peter Fishburn and William Trotter. Geometric Containment Orders: a Survey. *Order*, 1998.
- [92] Steve Fisk. A short proof of Chvátal’s watchman theorem. *Journal of Combinatorial Theory (JCTB)*, 1978.
- [93] Rudolf Fleischer. A Simple Balanced Search Tree with  $O(1)$  Worst-case Update Time. *International Journal of Foundations of Computer Science*, 1996.
- [94] Steven Fortune. Stable Maintenance of Point Set Triangulations in Two Dimensions. *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1989.
- [95] Steven Fortune and Christopher Van Wyk. Efficient Exact Arithmetic for Computational Geometry. *International Symposium on Computational Geometry (SoCG)*, 1993.
- [96] Michael Fredman. How Good is the Information Theory Bound in Sorting? *Theoretical Computer Science (TCS)*, 1976.
- [97] Michael Fredman and Dan Willard. Surpassing the Information Theoretic Bound with Fusion Trees. *Journal of Computer and System Sciences*, 1993.

- [98] Michael Fredman and Dan Willard. Trans-dichotomous Algorithms for Minimum Spanning Trees and Shortest Paths. *Journal of Computer and System Sciences (JCSS)*, 1994.
- [99] Scott Gaffney and Padhraic Smyth. Trajectory Clustering with Mixtures of Regression Models. *Knowledge Discovery and Data Mining (KDD)*, 1999.
- [100] Subir Ghosh and Partha Goswami. Unsolved Problems in Visibility Graphs of Points, Segments, and Polygons. *ACM Computing Surveys (CSUR)*, 2013.
- [101] Michael Godau. A Natural Metric for Curves — Computing the Distance for Polygonal Chains and Approximation Algorithms. *Symposium on Theoretical Aspects of Computer Science (STACS)*, 1991.
- [102] Jacob Goodman, Richard Pollack, and Bernd Sturmfels. Coordinate Representation of Order Types Requires Exponential Storage. *ACM Symposium on Theory of Computing (STOC)*, 1989.
- [103] Joachim Gudmundsson, Patrick Laube, and Thomas Wolle. Computational Movement Analysis. In *Springer Handbook of Geographic Information*. Springer, 2011.
- [104] Joachim Gudmundsson, Patrick Laube, and Thomas Wolle. Movement Patterns in Spatio-Temporal Data. *Encyclopedia of GIS.*, 2017.
- [105] Joachim Gudmundsson, Majid Mirzanezhad, Ali Mohades, and Carola Wenk. Fast Fréchet Distance Between Curves with Long Edges. *International Journal of Computational Geometry & Applications (IJCGA)*, 2019.
- [106] Joachim Gudmundsson, André van Renssen, Zeinab Saeidi, and Sampson Wong. Fréchet distance queries in trajectory data. *The Third Iranian Conference on Computational Geometry (ICCG 2020)*, 2020.
- [107] Joachim Gudmundsson, André van Renssen, Zeinab Saeidi, and Sampson Wong. Translation Invariant Fréchet Distance Queries. *ArXiv preprint:2102.05844*, 2021.
- [108] Leonidas Guibas and John Hershberger. Optimal Shortest Path Queries in a Simple Polygon. *Journal of Computer and System Sciences (JCSS)*, 1989.
- [109] Leonidas Guibas, John Hershberger, Daniel Leven, Micha Sharir, and Robert Tarjan. Linear-Time Algorithms for Visibility and Shortest Path Problems Inside Triangulated Simple Polygons. *Algorithmica*, 1987.
- [110] Leonidas Guibas, David Salesin, and Jorge Stolfi. Constructing strongly convex approximate hulls with inaccurate primitives. *Algorithmica*, 1993.
- [111] Eliezer Gurarie, Russel Andrews, and Kristin Laidre. A novel method for identifying behavioural changes in animal movement data. *Ecology Letters*, 2009.

- [112] Torben Hagerup. Sorting and Searching on the Word RAM. *Symposium on Theoretical Aspects of Computer Science (STACS)*, 1998.
- [113] Dan Halperin. Robust Geometric Computing in Motion. *The International Journal of Robotics Research*, 2002.
- [114] Sarel Har-Peled. *Geometric approximation algorithms*. American mathematical society Boston, 2011.
- [115] Idit Haran and Dan Halperin. An Experimental Study of Point Location in Planar Arrangements in CGAL. *Journal of Experimental Algorithmics (JEA)*, 2009.
- [116] Juris Hartmanis and Janos Simon. On the Power of Multiplication in Random-access Machines. *IEEE Symposium on Switching and Automata Theory*, 1974.
- [117] Martin Held and Joseph Mitchell. Triangulating Input-constrained Planar Point Sets. *Information Processing Letters (IPL)*, 2008.
- [118] Michael Hoffmann, Thomas Erlebach, Danny Krizanc, Matúš Mihalák, and Rajeev Raman. Computing Minimum Spanning Trees with Uncertainty. *Symposium on Theoretical Aspects of Computer Science (STACS)*, 2008.
- [119] Hiroshi Imai and Masao Iri. Polygonal approximations of a curve – formulations and algorithms. *Computational Morphology*, 1988.
- [120] Minghui Jiang, Ying Xu, and Binhai Zhu. Protein structure–structure alignment with discrete Fréchet distance. *Journal of bioinformatics and computational biology*, 2008.
- [121] Kai Jin. Maximal area triangles in a convex polygon. *ArXiv preprint:1707.04071*, 2017.
- [122] Leo Joskowicz and Yonatan Myers. Topological Stability and Convex Hull with Dependent Uncertainties. *European Workshop on Computational Geometry (EuroCG)*, 2014.
- [123] Jeff Kahn and Jeong Kim. Entropy and Sorting. *Journal of Computer and System Sciences (JCSS)*, 1995.
- [124] Frank Kammer, Maarten Löffler, Paul Mutser, and Frank Staals. Practical Approaches to Partially Guarding a Polyhedral Terrain. *International Conference on Geographic Information Science (GIScience)*, 2014.
- [125] Frank Kammer, Maarten Löffler, and Rodrigo Silveira. Space-efficient Hidden Surface Removal. *ArXiv preprint:1611.06915*, 2016.
- [126] Vahideh Keikha. On Optimal  $w$ -gons in Convex Polygons. *ArXiv preprint:2103.01660*, 2021.

- [127] Vahideh Keikha, Maarten Löffler, Ali Mohades, and Zahed Rahmati. Width and Bounding Box of Imprecise Points. *Canadian Conference on Computational Geometry (CCCG)*, 2018.
- [128] Lutz Kettner, Kurt Mehlhorn, Sylvain Pion, Stefan Schirra, and Chee-Keng Yap. Classroom Examples of Robustness Problems in Geometric Computations. *Computational Geometry*, 2008.
- [129] Elena Khramtcova and Maarten Löffler. Dynamic stabbing queries with sub-logarithmic local updates for overlapping intervals. *Computer Science—Theory and Applications*, 2017.
- [130] David Kirkpatrick. Optimal Search in Planar Subdivisions. *SIAM Journal on Computing (SICOMP)*, 1983.
- [131] David Kirkpatrick and Stefan Reisch. Upper Bounds for Sorting Integers on Random Access Machines. *Theoretical Computer Science (TCS)*, 1984.
- [132] David Kirkpatrick and Raimund Seidel. Output-size Sensitive Algorithms for Finding Maximal Vectors. *International Symposium on Computational Geometry (SoCG)*, 1985.
- [133] Victor Klee and George Minty. How Good is the Simplex Algorithm. Technical report, Washington Univ. Seattle Dept. of Mathematics, 1970.
- [134] János Körner. Coding of an Information Source having Ambiguous Alphabet and the Entropy of Graphs. *Prague conference on information theory*, 1973.
- [135] Drago Krznaric and Christos Levkopoulos. Computing a Threaded Quadtree from the Delaunay Triangulation in Linear Time. *Nordic Journal of Computing*, 1998.
- [136] Sam Kwong, Qianhua He, K. Man, Wallace Tang, and C. Chau. Parallel genetic-based hybrid pattern matching algorithm for isolated word recognition. *International Journal of Pattern Recognition and Artificial Intelligence (IJPRAI)*, 1998.
- [137] Patrick Laube, Marc van Kreveld, and Stephan Imfeld. Finding REMO - Detecting Relative Motion Patterns in Geospatial Lifelines. *Developments in Spatial Data Handling (SDH)*, 2004.
- [138] Jae-Gil Lee, Jiawei Han, and Kiu-Young Whang. Trajectory clustering: a partition-and-group framework. *ACM International Conference on Management of Data (SIGMOD)*, 2007.
- [139] Seungjun Lee, Taekang Eom, and Hee-Kap Ahn. Largest triangles in a polygon. *Computational Geometry*, 2021.
- [140] Chen Li, Sylvain Pion, and Chee-Keng Yap. Recent Progress in Exact Geometric Computation. *The Journal of Logic and Algebraic Programming*, 2005.

- [141] Xiaojie Li, Xiang Li, Daimin Tang, and Xianrui Xu. Deriving features of traffic flow around an intersection from trajectories of vehicles. *IEEE International Conference on Geoinformatics*, 2010.
- [142] Kwei-Jay Lin, Swaminathan Natarajan, and Jane Liu. Imprecise Results: Utilizing Partial Computations in Real-time Systems. *IEEE Real-Time Systems Symposium*, 1987.
- [143] Giuseppe Liotta, Franco Preparata, and Roberto Tamassia. Robust Proximity Queries: An Illustration of Degree-driven Algorithm Design. *SIAM Journal on Computing (SICOMP)*, 1998.
- [144] Chih-Hung Liu and Sandro Montanari. Minimizing the Diameter of a Spanning Tree for Imprecise Points. *Algorithmica*, 2018.
- [145] Maarten Löffler. *Data Imprecision in Computational Geometry*. PhD thesis, Utrecht University, 2009.
- [146] Maarten Löffler and Wolfgang Mulzer. Triangulating the Square and Squaring the Triangle: Quadrees and Delaunay Triangulations are Equivalent. *SIAM Journal on Computing (SICOMP)*, 2012.
- [147] Maarten Löffler and Wolfgang Mulzer. Unions of Onions: Preprocessing Imprecise Points for Fast Onion Decomposition. *Journal of Computational Geometry (JoCG)*, 2014.
- [148] Maarten Löffler, Joseph Simons, and Darren Strash. Dynamic Planar Point Location with Sub-logarithmic Local Updates. *Workshop on Algorithms and Data Structures (WADS)*, 2013.
- [149] Maarten Löffler and Jack Snoeyink. Delaunay Triangulation of Imprecise Points in Linear Time after Preprocessing. *Computational Geometry*, 2010.
- [150] Maarten Löffler and Marc van Kreveld. Largest and Smallest Convex Hulls for Imprecise Points. *Algorithmica*, 2010.
- [151] Shachar Lovett and Raghu Meka. Constructive Discrepancy Minimization by Walking on the Edges. Technical report, SIAM Journal on Computing (SICOMP), 2015.
- [152] David MacDonald and Kellogg Booth. Heuristics for ray tracing using space subdivision. *The Visual Computer*, 1990.
- [153] Harry Mairson and Jorge Stolfi. Reporting and counting intersections between two sets of line segments. *Theoretical Foundations of Computer Graphics and CAD*, 1988.
- [154] Bodo Manthey. Smoothed Analysis of Local Search Algorithms. *Workshop on Algorithms and Data Structures*, 2015.

- [155] Bodo Manthey and Heiko Röglin. Improved Smoothed Analysis of The k-means Method. *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2009.
- [156] Jiří Matoušek. Efficient partition trees. *Discrete and Computational Geometry (DCG)*, 1992.
- [157] Jiří Matoušek. Range searching with efficient hierarchical cuttings. *Discrete and Computational Geometry (DCG)*, 1993.
- [158] Nimrod Megiddo. Applying parallel computation algorithms in the design of serial algorithms. *Journal of the ACM (JACM)*, 1983.
- [159] Kurt Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*. Springer Science & Business Media, 2013.
- [160] Kurt Mehlhorn and Athanasios Tsakalidis. An Amortized Analysis of Insertions into AVL-trees. *SIAM Journal on Computing (SICOMP)*, 1986.
- [161] Amir Mesrikhani, Mohammad Farshi, and Behnam Iranfar. Minimum Spanning Tree of Imprecise Points Under  $L_1$ -metric. *Journal of Algorithms and Computation*, 2019.
- [162] Wouter Meulemans, Bettina Speckmann, Kevin Verbeek, and Jules Wulms. A framework for Algorithm Stability and its Application to Kinetic Euclidean MSTs. *Latin American Symposium on Theoretical Informatics (LATIN)*, 2018.
- [163] Ether Moet. *Computation and Complexity of Visibility in Geometric Environments*. PhD thesis, Utrecht University, 2008.
- [164] Shlomo Moran, Marc Snir, and Udi Manber. Applications of Ramsey’s Theorem to Decision Tree Complexity. *Journal of the ACM (JACM)*, 1985.
- [165] Ketan Mulmuley. Hidden Surface Removal with Respect to a Moving View Point. *ACM Symposium on Theory of Computing (STOC)*, 1991.
- [166] Takayuki Nagai, Seigo Yasutome, and Nobuki Tokura. Convex Hull Problem with Imprecise Input and its Solution. *Systems and Computers in Japan*, 1999.
- [167] George Nemhauser and Zev Ullmann. Discrete Dynamic Programming and Capital Allocation. *Management Science*, 1969.
- [168] Jürg Nievergelt and Edward Reingold. Binary Search Trees of Bounded Balance. *SIAM Journal on Computing (SICOMP)*, 1973.
- [169] Joseph O’Rourke. *Art Gallery Theorems and Algorithms*. Oxford University Press, Inc., 1987.
- [170] Evanthia Papadopoulou and Sandeep Kumar Dey. On the Farthest Line-Segment Voronoi Diagram. *International Journal of Computational Geometry & Applications (IJCGA)*, 2013.

- [171] Vilfredo Pareto. Il massimo di utilità dato dalla libera concorrenza. *Giornale degli economisti*, 1894.
- [172] Eunhui Park and David Mount. A self-adjusting data structure for multidimensional point sets. *European Symposium on Algorithms (ESA)*, 2012.
- [173] Michel Pocchiola and Gert Vegter. The Visibility Complex. *International Journal of Computational Geometry & Applications (IJCGA)*, 1996.
- [174] Franco Preparata and Michael Shamos. *Computational Geometry: An Introduction*. Texts and Monographs in Computer Science. Springer, 1985.
- [175] Daniel Reem. The Geometric Stability of Voronoi Diagrams with Respect to Small Changes of the Sites. *International Symposium on Computational geometry (SoCG)*, 2011.
- [176] Daniel Richardson. Some Undecidable Problems Involving Elementary Functions of a Real Variable. *Journal of Symbolic Logic*, 1968.
- [177] Azriel Rosenfeld. Fuzzy Geometry: an Updated Overview. *Information Sciences*, 1998.
- [178] David Salesin, Jorge Stolfi, and Leonidas Guibas. Epsilon Geometry: Building Robust Algorithms from Imprecise Computations. *International Symposium on Computational Geometry (SoCG)*, 1989.
- [179] Hanan Samet. *The design and analysis of spatial data structures*. Addison-Wesley, 1990.
- [180] Neil Sarnak and Robert Tarjan. Planar Point Location Using Persistent Search Trees. *Communications of the ACM*, 1986.
- [181] Arnold Schönhage. On The Power of Random Access Machines. *International Colloquium on Automata, Languages, and Programming (ICALP)*, 1979.
- [182] Mark Segal. Using Tolerances to Guarantee Valid Polyhedral Modeling Results. *ACM Special Interest Group on Computer Graphics and Interactive Techniques (SIGGRAPH)*, 1990.
- [183] Mark Segal and Carlo Séquin. Consistent Calculations for Solids Modeling. *International Symposium on Computational Geometry (SoCG)*, 1985.
- [184] Michael Shamos. *Computational Geometry*. PhD thesis, Yale University, 1979.
- [185] Micha Sharir. Almost tight upper bounds for lower envelopes in higher dimensions. *Discrete & Computational Geometry (DCG)*, 1994.
- [186] Jack Snoeyink. Point Location. In *Handbook of Discrete and Computational Geometry*. Chapman and Hall/CRC, 2017.

- [187] Daniel Spielman and Shang-Hua Teng. Smoothed Analysis of Algorithms: Why the Simplex Algorithm Usually Takes Polynomial Time. *Journal of the ACM (JACM)*, 2004.
- [188] Andreas Stohl. Computation, accuracy and applications of trajectories – A review and bibliography. *Atmospheric Environment*, 1998.
- [189] Kokichi Sugihara and Masao Iri. Construction of the Voronoi diagram for one million generators in single-precision arithmetic. *Proceedings of the IEEE*, 1992.
- [190] Leonida Tonelli. Sull'integrazione per parti. *Atti della Accademia Nazionale dei Lincei*, 1909.
- [191] Csaba Toth, Joseph O'Rourke, and Jacob Goodman. *Handbook of Discrete and Computational Geometry*. Chapman and Hall/CRC, 2017.
- [192] Alan Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *London mathematical society*, 1937.
- [193] Pravin Vaidya. Minimum spanning trees in  $k$ -dimensional space. *SIAM Journal on Computing (SICOMP)*, 1988.
- [194] Mees van de Kerkhof, Irina Kostitsyna, Maarten Löffler, Majid Mirzanezhad, and Carola Wenk. Global Curve Simplification. *European Symposium on Algorithms (ESA)*, 2019.
- [195] Ivor van der Hoog, Irina Kostitsyna, Maarten Löffler, and Bettina Speckmann. Preprocessing Ambiguous Imprecise Points. *International Symposium on Computational Geometry (SoCG)*, 2019.
- [196] Peter van Emde Boas. Machine Models and Simulations. In *Algorithms and Complexity*, Handbook of Theoretical Computer Science. Elsevier, 1990.
- [197] Marc van Kreveld, Maarten Löffler, and Joseph Mitchell. Preprocessing Imprecise Points and Splitting Triangulations. *SIAM Journal on Computing (SICOMP)*, 2010.
- [198] Marc van Kreveld, Maarten Löffler, and Lionov Wiratma. On Optimal Polyline Simplification Using the Hausdorff and Fréchet Distance. *Journal of Computational Geometry (JoCG)*, 2020.
- [199] Georges Voronoi. Nouvelles applications des paramètres continus à la théorie des formes quadratiques. Premier mémoire. Sur quelques propriétés des formes quadratiques positives parfaites. *Journal für die reine und angewandte Mathematik (Crelles Journal)*, 1908.
- [200] Emo Welzl. Constructing the Visibility Graph for  $n$ -line Segments in  $O(n^2)$  Time. *Information Processing Letters (IPL)*, 1985.

- [201] Martin Werner and Dev Oliver. ACM SIGSPATIAL GIS Cup 2017: Range Queries under Fréchet Distance. *ACM International Conference on Advances in Geographic Information Systems (SIGSPATIAL)*, 2018.
- [202] Andrew Yao. A Lower Bound to Finding Convex Hulls. *Journal of the ACM (JACM)*, 1981.
- [203] Chee Yap and Thomas Dubé. The Exact Computation Paradigm. In *Computing in Euclidean Geometry*. World Scientific, 1995.
- [204] Chee-Keng Yap. Towards Exact Geometric Computation. *Computational Geometry*, 1997.
- [205] Chee-Keng Yap. In Praise of Numerical Computation. In *Efficient Algorithms*. 2009.
- [206] Chee-Keng Yap, Vikram Sharma, and Jyh-Ming Lien. Towards Exact Numerical Voronoi Diagrams. *International Symposium on Voronoi Diagrams in Science and Engineering*, 2012.
- [207] Jihun Yu, Chee Yap, Zilin Du, Sylvain Pion, and Hervé Brönnimann. The Design of Core 2: A Library for Exact Numeric Computation in Geometry and Algebra. *International Congress on Mathematical Software*, 2010.





# Curriculum Vitae

Ivor van der Hoog was born on February 3, 1995, in Nieuwegein. He started his pre-university education at the Cals College, doing a bilingual gymnasium program. He started as a student in Mathematics and Computer Science at Utrecht University in 2013. In 2017, he received his Masters degree in computer science. That same year, he started as AIO (Assistent In Opleiding) at the Department of Information and Computing Sciences of Utrecht University. The results of the scientific work done during this thesis are summarized below (papers present in this thesis are marked with ★). In addition to performing research, he functioned as a TA in various courses and as university lecturer where he taught Security and coordinated the introduction project for game technology students. He served as a committee member of various departmental, faculty and community committees. Now, he continues his academic career at the Technical University of Denmark.

## Published papers:

- ★ E. Arseneva, I. van der Hoog, M. Löffler. Dynamic Smooth Compressed Quadrees. *International Symposium on Computational Geometry (SoCG)*, 2018.
- ★ I. van der Hoog, I. Kostitsyna, M. Löffler, B. Speckmann. Preprocessing Ambiguous Imprecise Points. *International Symposium on Computational Geometry (SoCG)*, 2019.
- ★ P. Eades, I. van der Hoog, M. Löffler, F. Staals. Trajectory Visibility. *Scandinavian Workshop on Algorithm Theory (SWAT)*, 2020.
- I. van der Hoog, V. Keikha, M. Löffler, A. Mohades, J. Urhausen: Maximum-area triangle in a convex polygon, revisited. *Information Processing Letters (IPL)*, 2020.
- ★ I. van der Hoog, I. Kostitsyna, M. Löffler, B. Speckmann. Preprocessing Imprecise Points for the Pareto Front. *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2021.

- I. van der Hoog, M. van de Kerkhof, M. van Kreveld, M. Löffler, F. Staals, J. Urhausen, J. Vermeulen. Mapping Multiple Regions to the Grid with Bounded Hausdorff Distance. *Algorithms and Data Structures Symposium (WADS)*, 2021.
- I. van der Hoog, M. van Kreveld, W. Meulemans, K. Verbeek, J. Wulms. Topological Stability of Kinetic  $k$ -centers. *Theoretical Computer Science (TCS)*, 2021.
- ★ J. Erickson, I. van der Hoog, T. Miltzow. Smoothing the gap between NP and  $\exists\mathbb{R}$ . *SIAM Journal on Computing (SICOMP)*, 2021.

#### Work on ArXiv or under review:

- E. Arseneva, I. van der Hoog, M. Löffler. Dynamic Stabbing Queries with Sub-logarithmic Local Replacement.
- ★ M. Buchin, I. van der Hoog, T. Ophelders, L. Schlipf, R. Silveira and F. Staals. Efficient Fréchet distance queries for segments.
- A. Driemel, I. van der Hoog and E. Rotenberg. On the Discrete Fréchet Distance in a Graph.
- K. Buchin, B. Custers, I. van der Hoog, M. Löffler, A. Popov, M. Roeloffzen and F. Staals. Segment Visibility Counting Queries in Polygons.



# Acknowledgements

Writing a thesis is a process which involves many people. Throughout my academic and educational career I have been surrounded by people who supported me, inspired me, guided me, or aided me in countless other ways. This section is dedicated to thanking those individuals who helped shape this thesis and the person I am today.

First, I want to thank Maarten Löffler and Marc van Kreveld, who were my supervisors for these past few years and who gave me the opportunity to do a PhD in computational geometry. Throughout this process I enjoyed their support and guidance and for the many opportunities that they created for me. Almost all publications that lead to this thesis are the result of many, many research sessions we had together. I also want to thank Gerard Tel, Lennart Herlaar and Frank van der Stappen who guided me in my teaching endeavours. I want to thank them for the faith and the support that they gave me, and the many opportunities that they presented me to expand my teaching experience.

I want to thank the two high school teachers who were the first to inspire me to pursue an my academic career: dr. R. Schoen and dr. S. de Haar. Dr. de Haar jump-started my interest into seeking answers to complicated questions. She was the first who introduced me to university-level content when she gave me her first-year university chemistry textbook. Dr. Schoen's classical method of teaching (befitting to a teacher of the classics) which combines presenting structure and consequences with individual responsibility and freedom inspired the teaching ethos that I adhere to today.

I want to thank the people who throughout my studies spent considerable efforts to keeping me engaged and on track. A special thanks to: prof. J. Hogendijk, who enrolled me in his personal 'hidden talent' program after I failed the first course that I enrolled in. With great patience he taught me essential skills such as maintaining a calendar and careful reading. To prof. M. Crainic, who invited me to his office to personally establish high expectations after I for the first time definitively failed a course: Topology. And to dr. R. Abma who exposed me to other scientific fields.

I want to thank all my coauthors. Naturally this includes my supervisors, but I also want to thank all the coauthors of the papers that lead to this thesis (E. Arseneva, M. Buchin, P. Eades, J. Erickson, I. Kostityna, T. Miltzow, T. Ophelders, L. Schlipf, R. Silveira, B. Speckmann, and F. Staals) and all coauthors of papers beyond this thesis (A. Driemel, W. Evans, V. Keikha, M. van de Kerkhof, D. Kirkpatrick, W. Meulemans, A. Mohades, E. Rotenberg, J. Urhausen, K. Verbeek, J. Vermeulen, J. Wulms). A special thanks to prof. B. Speckmann whose standards I still try to uphold today.

I want to thank my direct coworkers and peers (S. de Berg, M. van der Kerkhof, T. Miltzow, F. Staals, J. Urhausen, J. Vermeulen) for the great days at Utrecht University. We had many fun discussions, games, lunches and afternoon tea breaks. I want to also thank them for their useful input for my work, and their support.

Finally, I want to thank all my friends and family. Writing a thesis is a long process and for the whole duration it was continuously supported through love, interaction and distractions. You all provided those in great quantities. A special thanks to my parents for their unconditional support and a very special thanks to Kees Blijleven, who is amazing, and on whose support I depend the most.