Data Imprecision in Computational Geometry

Typeset in LaTeX.
Cover and figures designed using the Ipe extensible drawing editor.

Printed by BOX Press, Oisterwijk, the Netherlands.
www.proefschriftmaken.nl

Copyright © 2009 Maarten Löffler.
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 Maarten Löffler.

ISBN 978-90-8891-121-7

Data Imprecision in Computational Geometry

Data-Imprecisie in de Computationele Meetkunde
(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. J.C. Stoof,
ingevolge het besluit van het college voor promoties

in het openbaar te verdedigen op
maandag 19 oktober 2009 des middags te 4.15 uur

door

Maarten Löffler

geboren op 14 augustus 1982 te Amsterdam

Promotor: Prof.dr. R.C. Veltkamp
Co-promotor: Dr. M.J. van Kreveld

Preface

It seems that my PhD thesis is finished. It feels strange to realise this fact: I actually put together a more or less coherent book of more than 200 pages, most of which even have content that makes sense. Before starting with those, though, let me briefly look back at how this came to be.

When I was in primary school, arithmetic was easily my least favourite subject. I remember that each year there was a new book, its pages filled with countless additions, multiplications, divisions, and similar operations, neatly aligned in rows and columns. Each equation would look something like "$108 - 29 =$" or "$7 \times 31 =$". I believe the idea was that I had to copy these equations to my exercise book, and then write down a number behind each "$=$" sign, to make the equation true. I could never see the point in doing this, and have spent a lot of time staring at these pages, wondering why we couldn't be spending that time doing something fun like handenarbeid[1] instead. I remember that in the eighth year, the children who finished their arithmetic work within the assigned time were allowed to play a game on the school computer. However, this was never an option for me because I was still working in the book of year six. When I was twelve years old, though, everything changed. I advanced to secondary school, and there we did not learn arithmetic, but *mathematics*. Suddenly a door was opened for me into the wondrous world of logic, of abstract thoughts, relations and riddles, of algebra and number theory, and, not unimportantly, of geometry. I was instantly hooked.

I first learnt to program about four years later. We did have a computer at home as well, and when my sister and I were small children, my father had been experimenting with computer programming, and made a few games for us to play. Of course, at the time I never guessed that he had made them himself. Then, one day he decided to

---

[1] Literally, "handenarbeid" translates to "hand labour" or "manual labour", but the kind taught at Dutch elementary schools is more about being creative with carton and glue than about, say, cutting down trees.

teach me the basics of a computer language called "BASIC". Computers were already fairly common in those days, and I had been working (or rather, playing) with them all my life, but the idea that I could make a computer do virtually anything I wanted it to, by simply typing in a few words, had never really crossed my mind. After this introduction, though, I soon had balls flying around the screen, and in no time at all I had written elaborate interactive stories, and implemented several intricate strategic games. Looking back at them now, I must admit that these products of my creativity were not all that impressive, but I clearly remember the excitement with which I laboured on those projects at the time. It was the discovery of computer programming that made me doubt my implicit decision to study mathematics after school for the first time.

Fortunately, by the time I had to choose the Direction of my Future, a programme called "TWINFO" (TWIN WIskunde and INFOrmatica) was being offered by the university of Utrecht, which promised to combine both studies in a single curriculum. This turned out to be the case only during the first year, after which it was left to my own creativity and organisational skills to complete the remaining four years of both studies; but, it was enough to lure me in. Though I encountered some very different branches of mathematics during my studies that I liked, somehow the courses that involved *geometry* in some form always specifically interested me. Visual drawings could directly transfer an idea to my mind, and I found them more appealing than formulae involving endless strings of these funny $\int \int \int$ symbols, that require significant staring at before they start making anything resembling sense. The computer science programme, though, did not offer many courses that were supplied with nice pictures. Instead, I was led astray by the intricacies of compiler construction, and at the end of my third year, I wrote what was the equivalent of my Bachelor thesis about generating error messages in a compiler for the functional programming language "Haskell".

However, in my fourth year, the university was just in the process of introducing the Bachelor/Master system, and I had to choose a Master programme, even though technically I was still a student in the old Propedeuse/Doctoraal system. Although I had enjoyed my two months of work in the software technology lab, I was intrigued by a programme called "GIVE" (Geometry, Imaging and Virtual Environments). I decided to take this direction, and as the name suggested it included several computer science courses that were rather more visual in nature than any I had seen so far. I particularly enjoyed a course on "geometric algorithms", which nicely combined the two main directions that my interest had taken. I decided that I wanted to write my Master thesis on some topic in this field. Probably due to the fashion of the moment, I ended up studying the effects of data imprecision on the computation of the convex hull, a topic which turned out to be conceiling many intriguing puzzles for me to unveil and solve. In fact, one of the results from that thesis is repeated in this one, in Chapter 4. After I finished my studies, I was offered the option to continue for a PhD on the same topic, on the "GOGO" (Geometric Optimisation with Geometric cOnstrants) project. During this time, I was able to get to know the subject better, and it has always provided me with more than enough nice, challenging, and perhaps most importantly, *unsolved* problems.

And that is how I ended up writing a PhD thesis about data imprecision in computational geometry. For those who do not immediately appreciate the implications of the last sentence, I would like to point ahead to Chapter 1, where the topic is discussed in some detail. When I had finished my Master thesis about imprecision in convex hulls, of course I proudly showed it to many of my friends, family, and other acquaintances. Most of them would, after the obligatory praising words, invariably ask the same question in the end: “What is it about?” Since they would not be satisfied with the obvious answer, “imprecision in convex hulls”, I thought it was nice to make things a little more concrete by explaining the elastic band analogy: the convex hull of a set of points is what you get when releasing an elastic band around a bunch of nails sticking out of a flat surface. I then proceeded to describe the problems that arise when the locations of the nails are unknown, but this is where I usually lost my audience. Since then, there have been persistent rumours in certain circles that everything I do all day long at my work is put nails into every available flat surface, and wrap elastic bands around them. I want to use this opportunity to stress that this is really not the case, and that there are still several undamaged flat surfaces left in my office.

I hope you enjoy reading this thesis.

Maarten Löffler
Wageningen, September 2, 2009

Contents

Preface v

Contents ix

PART I Introduction

1 Introduction 3

1.1 Computational Geometry 3

1.2 Imprecision 9

1.3 Imprecision and Computational Geometry 14

1.4 Contribution of this Thesis 15

2 Modelling Imprecise Data 17

2.1 Imprecision in Input and Output 17

2.2 Imprecise Points Modelled as Regions 22

2.3 Other Models for Imprecise Points 26

2.4 Imprecision in Other Types of Input 29

2.5 Closing Remarks 33

PART II Bounds on Output Imprecision

3 Bounds on Shape Fitting 37

3.1 Axis-Aligned Bounding Box 38

3.2 Smallest Enclosing Circle 43

3.3 Width 45

3.4 Closing Remarks 50

4 Bounds on the Convex Hull Area 51

4.1 Largest Convex Hull of Parallel Line Segments 52

4.2 Largest Convex Hull of Squares 53

4.3 Smallest Convex Hull 57

4.4 Disks 64

4.5 Closing Remarks 65

5 Approximate Largest Convex Hull 67

5.1 Preliminaries 68

5.2 Approximating the Largest Convex Hull 69

5.3 Squares 70

5.4 Disks 73

5.5 Closing Remarks 77

6 Bounds on the Diameter 79

6.1 Largest Possible Diameter 80

6.2 Smallest Possible Diameter 82

6.3 Smallest Diameter for Squares 84

6.4 Smallest Diameter for Disks 89

6.5 Closing Remarks 91

PART III Preprocessing Imprecise Points

7 Preprocessing for Triangulation 95
7.1 Triangulations 97
7.2 Splitting a Triangulation 98
7.3 Triangulating Imprecise Points 104
7.4 Extensions 106
7.5 Closing Remarks 107

8 Preprocessing for Delaunay Triangulation 109
8.1 Expanded Gabriel Circles 110
8.2 Delaunay Triangulation of Imprecise Points 116
8.3 Extensions 119
8.4 Closing Remarks 120

9 Preprocessing More General Regions 121
9.1 Disks of Different Sizes 122
9.2 Overlapping Disks 126
9.3 Thick Regions 129
9.4 Closing Remarks 129

PART IV Beyond Imprecise Points

10 Imprecise Lines 133
10.1 Linear Programming 135
10.2 Vertical Extent 141
10.3 Closing Remarks 147

11 Bounds on the Perimeter of a Polygon 149
11.1 Longest Polygon 150
11.2 Shortest Polygon 151
11.3 Solving the Transformed Problem 154
11.4 Closing Remarks 158

12 Simpleness of a Polygon 159
12.1 Simpleness and Degenerate Simpleness 160
12.2 Simple Straight Tours through Vertical Line Segments 161
12.3 Simple Straight Tours through General Regions 169
12.4 Shortest Simple Tours through Line Segments 171
12.5 Closing Remarks 173

Conclusions 175
Upper & Lower Bounds 175
Preprocessing for Triangulations 177
Imprecise Lines and Polygons 178
Final Remarks 179

Samenvatting 181
Computationele Meetkunde 181
Imprecisie 184
Grenzen 186
Voorberekenen 188
Meer dan Punten 190
Conclusie 192

Bibliography 193

Curriculum Vitae 203

Acknowledgements 205

PART I

Introduction

Chapter One

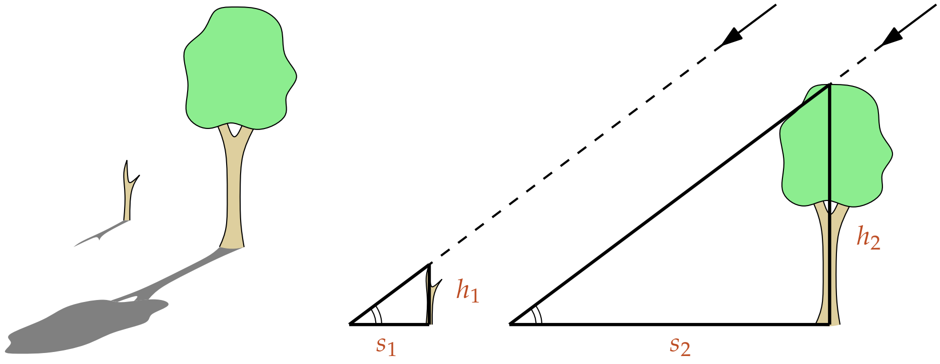
Introduction

This thesis is about the issue of *imprecision* in the field of *computational geometry*. In a single sentence, this field deals with the development of provably correct and efficient solutions to geometric problems, or the construction of mathematical proofs that no such solutions exist. Geometric problems are all around us, and such solutions are in high demand. However, an important obstacle to the practical application of techniques from computational geometry is the presence of imprecision. In order to be able to give mathematical guarantees, these techniques assume that the data they work with is correct, with absolute certainty and infinite precision. In practice, this is often not the case, and as a result the value of these guarantees is questionable.

In this first chapter, we will look into this issue in some more detail. We will overview the history and development of computational geometry, and how imprecision is fundamentally related to it. Then we will give a broad overview of how these problems can be dealt with. In the next chapter, we give a detailed overview of how imprecision can be modelled mathematically, and how this changes the problems studied in computational geometry.

1.1 Computational Geometry

Computational geometry is the branch of theoretical computer science that is concerned with developing provably efficient algorithms for solving geometric problems. Such algorithms are also called geometric algorithms.



(a) (b)

Figure 1.1 The height of a tree can be computed by measuring its shadow length, and the height and shadow length of a (smaller) reference object.

1.1.1 Geometry

Geometry is the science of shapes, of properties and relations of spatial objects such as points, lines, circles, planes, balls, etc. It is one of the oldest sciences that exist, dating back to the ancient Egyptians and possibly earlier. The word *geometry* comes from the ancient Greek γεωμετρία, which literally means "earth measure". We live in a spatial world, and geometry came into existence out of the practical need to understand that world, and to measure and analyse it.

The first systematic scientific treatment of geometry is due to Euclid of Alexandria (∼300 BC), who in his *Elements* [44] builds a formal theory based only on the concepts of *points* and *lines*, and five axioms or *postulates* that describe their properties and relations. This theory shows a remarkable likeness to the real world, and has been the standard model for many centuries. After the Greeks, geometry became an integrated part of analytical mathematics, and was not studied as a discipline of its own until Jacob Steiner (1796-1863) revived the so-called *synthetic geometry* in Euclid's style in his *Systematic Development of the Dependencies of Geometric Objects* [123]. Though several other geometric models have been developed since then, Euclid's model is still the most common even today.

Geometry is interesting by itself as an abstract branch of mathematics, in which countless nice properties and relations have been discovered throughout the ages. However, one may argue that the main reason for its continued popularity is still its original purpose: as a tool to analyse the world we live in. But, as the Greek word already suggests, before we can analyse it, we first have to measure it. To be more precise: in order to use geometry as a tool, we have to measure the *locations* of our points of interest with respect to each other, or with respect to something else. In the

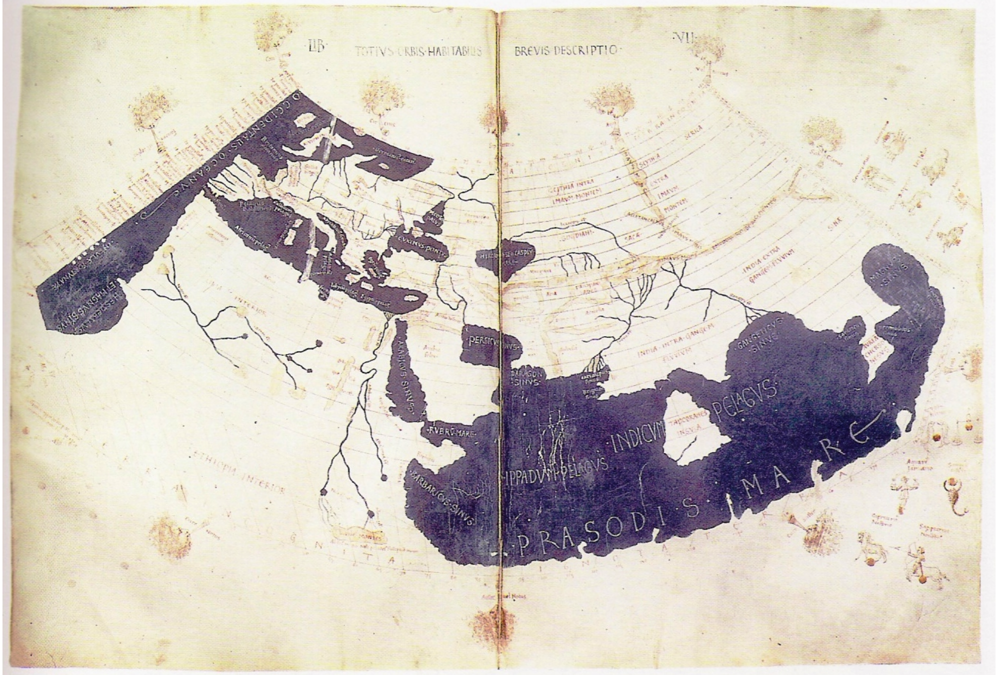


Figure 1.2 Ptolemaeus' map of the known world.

early days of geometry, such measurements took two forms: *distances* and *angles*.

As an example of a simple problem that can be solved with geometry, consider the situation in Figure 1.1(a). Suppose you want to know how high a tall tree is. You may not be able to measure the height of the tree directly, but you can compute its height instead by using the geometric law of *similarity*: if two triangles have the same angles, then the proportions of their side length are the same. In this example, the proportions between the height and shadow length of the tree and the stick are the same, see Figure 1.1(b). So by measuring the height h_1 of the stick, and the shadow lengths s_1 of the stick and s_2 of the tree, we can determine the height h_2 of the tree indirectly by computing $h_2 = s_2 \times h_1 / s_1$.

After these initial simple uses, many other methods of measuring locations have been developed. Perhaps the most literal application of geometry is the attempt to measure the location of important features of the world itself. Because the distances that are involved are very large, it is much harder to make precise measurements. One of the first systematic efforts to measure many locations of interest and map them onto a geometric space is due to Claudius Ptolemaeus ($\sim$90-168), who in his *Geographia* [110] provides one of the first world maps. Figure 1.2 shows a 15th century reproduction of this early map. The map looks quite different from what we are used to today. Over the centuries, methods for measuring locations have improved tremendously, providing us with a much more precise image of the world now. However, imprecision remains inherent in measuring anything. We will look into this more in Section 1.2.

1.1.2 Algorithms

An algorithm is a step-by-step description of how to do something, a recipe to accomplish something complex. In that sense, algorithms have existed as long as intelligent humans exist, for example as cooking recipes. The first known algorithms in a mathematical context, however, were developed by the Babylonians, who had methods to compute the square root of a positive integer (whole) number or to factorise an integer number into prime numbers. A famous early algorithm is due to Euclid of Alexandria, who devised a very simple way to compute the greatest common divisor of two integer numbers. When we are given two positive integer numbers A and B, and we perform the following steps, we are guaranteed to end up with the correct answer.

step 1: If $A = B$, then A (and B) is the greatest common divisor, and we can stop the algorithm.
step 2: If $A > B$, then subtract B from A and go back to step 1.
step 3: If $B > A$, then subtract A from B and go back to step 1.

The word *algorithm* comes from Muhammad ibn Musa al-Khwārizmī ($\sim$780-850), whose *On Calculation with Hindu Numerals* [5] is the foundation of our current decimal number system. His name evolved through several transliterations into the current “algorithm”. Algorithms have always been useful to do complicated computations in a systematic way, but have become much more important with the introduction of computers in the second half of the twentieth century. Computers can execute algorithms much faster and more reliably than humans can.

Before an algorithm to solve a problem can be designed, the problem needs to be stated formally. Such a problem statement must specify an *input* and an *output*. The input is what goes into the algorithm, and the output is what should come out of it. In the example above, the input is a pair of integer numbers, and the output is the greatest common divisor of these numbers. The essence of an algorithm is that it can be executed without “understanding” why each step is needed; for a given input it should always create the required output. This means that the user of the algorithm has to trust that the algorithm indeed does what it is supposed to do. For this purpose, an algorithm is often accompanied by a *proof of correctness*.

Since the introduction of computers, algorithms have become gradually more complicated, and proofs of correctness are also becoming much more involved. These proofs are studied by the branch of theoretical computer science called *analysis of* *algorithms*. Apart from the correctness of an algorithm, another important property is its *time complexity*. A given algorithm in principle works on any input, but the time it takes to complete usually depends on the *size* of the input, generally denoted by n. Two algorithms may solve exactly the same problem, but one could be much faster than the other. An algorithm is considered more efficient if it takes less time for large values of n. For example, an algorithm takes *linear* time if the time it takes to complete grows proportionally to n, and *quadratic* time when it grows proportionally to n^2. This can be made mathematically precise by using the so-called *big O*-notation: an algorithm has a time complexity of, for example, $O(n)$ or $O(n^2)$. In the last few years, methods for acquiring and storing large amounts of data have advanced a lot, and as a consequence, efficiency analysis of algorithms is becoming more important than ever.

1.1.3 Geometric Algorithms

A geometric algorithm is an algorithm to solve a geometric problem. In a geometric problem, the input is some spatial object, for example, a set of points in 2-dimensional space. The output could also be a spatial object, or just a simple number. Although geometric algorithms have been used since the first geometric problems were considered, their popularity increased tremendously after the introduction of computers. On the one hand, this is because they are tedious to execute, and on the other hand, because for many geometric problems on small instances humans can just "see" the solution. However, the introduction of computers changed this: computers can handle much larger problems because of their speed, but on the other hand are not very good at "seeing". In many different fields across science, some aspect of reality is studied by using a geometric model, which then has to be analysed, processed, or otherwise computed on. These fields can benefit from computers by using geometric algorithms.

As a classical example of a geometric algorithm, consider the *convex hull* of a set of points in the plane. A subset of the plane is said to be *convex* when for every two points p and q in the set, all points on the line segment between p and q are also in the set. The convex hull, then, is the smallest convex set that contains a given set of input points. Consider the set of ten points in Figure 1.3(a). For a human, it is easy to see what the convex hull of these points is, namely as in Figure 1.3(f). But if the set does not contain ten but one million points, or if we have not one but a hundred thousand sets of ten points, then we need a computer to solve it, and a computer needs an algorithm. A possible algorithm to solve the problem is outlined below.

step 1: Sort the points from left to right.[1]
step 2: Connect the points by a chain, as shown in Figure 1.3(b).
step 3: Remove any point where the chain turns left, as in Figure 1.3(c).

---

[1] We assume that there are no two points with the same x-coordinate. If there are, we can still use the algorithm if we first rotate the input point set until this assumption is met.

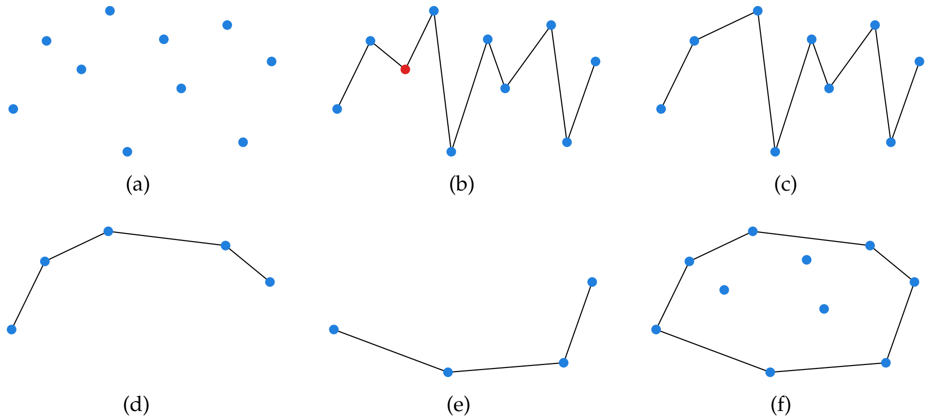

(a) (b) (c) (d) (e) (f)

Figure 1.3 (a) A set of points. (b) The chain that connects the points from left to right. The first point where the chain makes a left turn is marked. (c) The chain with one point removed. (d) The top half of the convex hull of the points. (e) The bottom half of the convex hull of the points. (f) The convex hull of the points.

step 4: Repeat step 3 until there are no such points left.
step 5: The resulting chain, shown in Figure 1.3(d), is the top half of the convex hull.
step 6: Repeat steps 1 to 4, but this time remove points where the chain makes a *right* turn instead of a left turn.
step 7: The resulting chain in Figure 1.3(e) is the bottom half of the convex hull.
step 8: Connect the two chains together to get the final result in Figure 1.3(f).

Many application fields for geometric algorithms exist. In most cases, there is an obvious link between the Euclidean space of the geometric model and the real world that we live in. For example, in Computer Aided Design, a real-world object is designed on a computer. In Computer Graphics, an artificial 3-dimensional world is created, which is then processed into images that should look like what the real world looks like as seen from a human eye. In Geographic Information Systems, a usually 2-dimensional model of the surface of the earth is stored in a computer, on which analysis is done. In Integrated Circuit Design, the layout of an electrical circuit is studied geometrically, before being printed. In Molecular Biology, complex molecules and their interaction are studied using a model in 3-dimensional space.

However, there are also application domains that use a geometric space that is not directly linked to the real world. In Databases, entries with multiple numeric attributes (such as the age and salary of employees) can be represented as points in a higher-dimensional Euclidean space. In Robotics, computations are usually done in the so-called *configuration space*, where a set of parameters that determine the position of, for example, a robot arm, defines a higher-dimensional Euclidean space.

In each of these fields, computations are done in a geometric domain, for which geometric algorithms are used. Originally, though, many of these algorithms were designed by specialists of the respective fields. As the computer applications grew more complex, so did the algorithms, and as a result they are often not as fast as possible, or even do not produce the correct result in certain rare situations. To solve these issues, the art of analysis of algorithms had to be applied to geometric algorithms.

The first systematic analysis of geometric algorithms is due to Michael Ian Shamos. In 1978, his thesis entitled *Computational Geometry* [120] opened up a field that has become ever more popular since. By explicitly studying the geometric properties of problems, it is often possible to discover algorithms that are provably correct, and often much faster than those that are used in practice. By now, many intricate and very interesting results are known. However, in order to make such theoretical guarantees, certain simplifying assumptions about reality were made.

1.2 Imprecision

In order to use geometric algorithms to study and analyse the world around us, the world must first be observed and data about it recorded, then this data must be represented in a meaningful mathematical model, and finally this model must be made accessible to a computer. Ideally, this would provide the computer with a correct description of the world, on which it can do its computations. However, in practice there are several places in this data flow where the data is distorted, resulting in an incorrect, or *uncertain*, description of the world.

There are many aspects of uncertainty in data, but the most important ones are *accuracy* and *precision*.[2] Accuracy is a measure of the “correctness” of uncertain data; it captures how close the stored data is to the “true” value that it is supposed to represent. Precision, on the other hand, is a property of uncertain data that exists independent of the “true” value: it is the extent to which the data is known or stored, for example the resolution of a bitmap, the number of digits in numerical values, or the number of points used to represent a curve as a polygonal line.

Accuracy and precision are independent concepts, that is, data can be precise but inaccurate or accurate but imprecise. However, precise but inaccurate data does not have much value, and, if this is known to be the case, is often deliberately made less precise to reduce storage space until the level of precision matches the accuracy. In this thesis, we will focus on the issue of imprecision, that is, we assume that data is accurate (at least to the degree of the precision) but imprecise.

---

[2] Other terms used in the literature to describe aspects of uncertainty include *vagueness*, *reliability*, *fuzzyness*, *consistency*, as well as a host of others, often with subtle differences in meaning. We will not consider these concepts in this thesis.

1.2.1 Sources of Imprecision

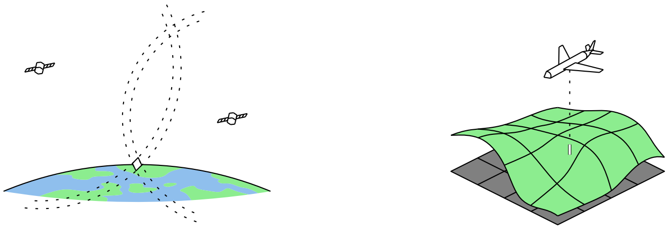
At the highest level, we can distinguish three types of imprecision, corresponding to three different phases in the data flow of a geometric application. Firstly, there is the phase of observing real-world objects and translating them into geometric objects. Secondly, there is the phase of modelling the part of the world we are interested in as objects in a geometric space. Lastly, there is the computation inside the geometric model.

The first and most direct source of imprecision is introduced when *observing* the world. Input data has to be collected using measuring equipment. Devices such as altimeters, laser scanners, or GPS receivers return the coordinates of a point, but this is only an approximation of the real location. The reason for this is that the device was not precise enough, and the amount of imprecision depends on the quality of the measuring device. Measuring equipment quality improves continually. But even modern devices used today are not always very precise, for example because a lot of data is collected at low cost, or because techniques are not yet developed far enough. In fact, the so-called *observer effect* states that it is theoretically not even possible to measure a location exactly, without altering it.

A popular method of measuring locations today is the *Global Positioning System (GPS)*. In this system, a location is computed by measuring the distance to a number of satellites orbiting the earth, see Figure 1.4(a). These distances determine spheres with certain radii, and the measuring device should be at the intersection of those spheres. However, the distance to a satellite cannot be measured exactly, but can have any value within a certain error interval. As a result, the spheres actually become shells with a certain width, and the location of the receiver could be anywhere in the region that is the intersection of those shells. The precision error of the location this results in may be up to 20 meters, although an increase in the number of satellites and the use of a technique called *differential* GPS have reduced this error significantly. Before the year 2000, it was even worse because the signal was intentionally degraded for non-military uses.

As another example, height information used to construct digital terrain models is often collected by airplanes flying over the terrain and sampling the distance to the ground, for example using a technique called *Light Detection and Ranging (LIDAR)*, see Figure 1.4(b). The data collected by these methods is also imprecise, for example because of imprecision in the plane's flying altitude, or because of artefacts on the ground. In high-resolution terrains distributed by the United States Geological Survey, it is not unusual to have vertical errors of up to 15 meters [126], although other data sets achieve a precision in the order of centimeters.

The second and perhaps the most fundamental source of imprecision lies in the actual geometric model used to describe the world. We usually model the space we live in as a 3-dimensional Euclidean space. However, an old discussion is whether the formal geometric model by Euclid is actually a valid description of the real world. Immanuel Kant (1724-1804) first argues in his *Critique of Pure Reason* [72] that Euclidean geometry



(a) (b)

Figure 1.4 (a) A location is computed by a GPS receiver by measuring its distance to a number of satellites orbiting the earth. (b) In a LIDAR system, an airplane flies over the earth surface and shoots laser rays vertically down to measure its distance to the ground.

is not based on observations, but instead is a synthetic a priori construction, and therefore there is no reason why it should be the "right" model. Shortly afterwards, various alternative geometries were introduced. Ever since Albert Einstein (1879-1955) published his *Theory of Relativity* [42], it is commonly believed that the structure of the universe does indeed not follow Euclid's model. However, what the "true" model should then be and whether it is even possible to construct one, remains a difficult and mostly philosophical discussion.

Even when one does accept Euclidean geometry as the way to model the world, it is still not always possible to translate real-world objects into geometric models. One problem is that of *interpolation*: even if we could make precise measurements of some quantity, we could not measure it at every single point in our domain, since there are infinitely many such points. As a result, data always has to be interpolated. As another example, in a GIS application we may wish to represent a shoreline by a geometric curve. However, the shoreline is not completely well-defined: it moves back and forth with the tides, and even on a smaller time scale it changes with every wave. This presents an inherent imprecision into the model. Or consider an area of land classified as "forest". What exactly is the boundary of such a forest? How many trees are needed for a group of trees to be called a forest? Such classification problems also give rise to imprecision. Jingxiong Zhang and Michael Goodchild extensively discuss imprecision of this kind in their book *Uncertainty in Geographical Information* [131].

Finally, a third source of imprecision is the way present-day computers are built. Computers store the data they work on as a sequence of *bits*, which is inherently a discrete model. However, Euclidean geometry works on a continuous space of points. In two dimensions this would be $\mathbb{R}^2$, the space of all pairs of two elements from $\mathbb{R}$, the so-called *Real* numbers. To be able to work with this, Real numbers are approximated by nearby points that can be described by the computer. On the other hand, geometric algorithms usually assume the so-called *Real RAM (Random Access Machine)* model:

they assume that a computer can perform exact operations directly on Real values. This can cause problems when executing the algorithms, since the computer has to convert the values to be able to use them, while the algorithms do not anticipate this.

1.2.2 Effects of Imprecision

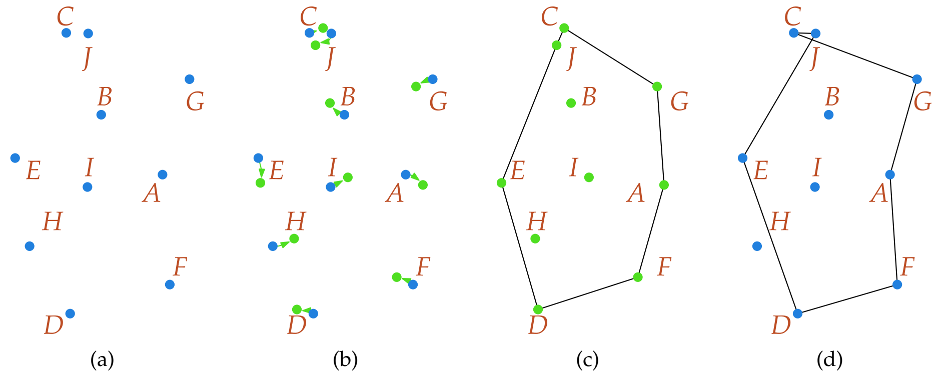
Together, the sources of imprecision described above can lead to unexpected results when executing theoretically correct algorithms in real life applications, varying from slightly different output values or longer computations times to outright wrong answers or crashing programs.

As an example, consider the following situation. Suppose we have a set of points in the plane, and we are interested in the convex hull of these points. To compute it, we use a provably correct and efficient algorithm, for example the one described in Section 1.1.3. Still, we may end up with the wrong hull.

First, assume we have named (or numbered) the points of interest. Figure 1.5(a) shows the points, named A to I, in reality. Next, we measure the points with a measuring device which has a small error. This results in another set of points, similar to the real one but still different, as shown in Figure 1.5(b). This set of points is stored in the computer memory and used as the input to the algorithm. Now, we apply an algorithm and it provides us with the correct convex hull of the measured points, see Figure 1.5(c). We can concisely describe this convex hull by outputting the order in which the points appear on the hull, in this case $D - E - J - C - G - A - F - D$. However, if we look at this sequence in the true space, before the points were perturbed due to erroneous measurements, we see in Figure 1.5(d) that it is not at all the right hull. The resulting polygon is not convex (at point A), it does not contain all points (point H is not inside the hull), and it even intersects itself (close to C and J).

In this example, we assumed that the imprecision came from the measuring equipment, but all sources of imprecision described in the previous section can have similar effects. Depending on how the output is further processed, the consequences of the imprecision can vary from a slight error in some numeric output value, to a program crash because a subsequent algorithm is assuming a consistent state of the data which is not the case.

It should be noted here, though, that the example above was especially constructed to show what can go wrong, and that in practice imprecision does not always cause serious issues. That is why people have been able to successfully use algorithms despite ignoring imprecision. However, it is possible that things do go wrong, and it does happen occasionally. It will be clear that in certain applications this is not acceptable, and that looking only at theoretical guarantees within the mathematical model where they are defined and proven may be misleading.



(a) (b) (c) (d)

Figure 1.5 (a) A set of points in the plane. (b) A slightly perturbed version of the point set. (c) The convex hull of the perturbed points. (d) The hull of the perturbed points translated back to the original points.

1.2.3 Dealing with Imprecision

One of the earliest records of an approach to handle geometric imprecision is due to Carl Friedrich Gauss (1777-1855), who was able to accurately predict the location of the dwarf planet Ceres in 1801 after it had disappeared from sight for a while. In his *Theory of Celestial Movement* [52], he describes how to extrapolate the location of a satellite from a limited number of imprecise measurements. His method, now known as the *least squares method*, fits a polynomial curve in the best possible way through a set of points. He proves that this curve has high probability of being close to the correct curve, if the errors of the points are distributed according to what we now call the *normal distribution*. The probability and accuracy increase when more points are used. This general principle of countering imprecision by making multiple measurements forms the base for statistical methods. This principle has proven useful in situations where detailed information about the distribution of errors is available, it is possible to make large numbers of measurements, and computation time is not too limited.

A modern approach to capture imprecision caused by the modelling of the world was introduced by Lotfali Asker Zadeh. In 1965, he defined in his book *Fuzzy Sets* [130] a framework where each element of a set is in the set with a certain probability. This allows for modelling spatial regions with unclear, or “fuzzy”, boundaries. Many operations for classical geometry have been defined and studied for fuzzy geometry as well [115].

Imprecision caused by the third phase, that of translating a geometric model to something a computer can work with, has been studied extensively and is quite well understood. Since this source of imprecision lives entirely within a theoretical model, it is possible to study exactly what causes errors and how these can be avoided. As one would expect, if a computer is allowed to use more bits for each point, then the approximation becomes better and the imprecision is less severe, but in some cases additional techniques are needed to prevent wrong answers. An alternative approach is to actually simulate a computer that can work with a Real RAM on an existing computer, but this is not done very often in practice because the extra translation makes it much slower. By now, there is a good understanding of these methods and their advantages and disadvantages [129], and extensive software libraries are available, such as LEDA [96] and CGAL [22], that can automatically switch between the two approaches depending on what is needed.

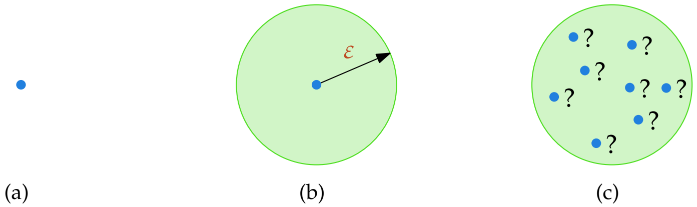
Today, geometric algorithms are used extensively in practice. Most of these algorithms, however, do not take imprecision into account at all, and if they do, they are usually heuristics that have been tested in practice, rather than algorithms with guaranteed behaviour.

1.3 Imprecision and Computational Geometry

Computational geometry is a relatively young field. In the early years virtually all attention has gone to solving the geometric problems themselves, and not much thought has been given to imprecision. In the beginning, this could be justified in several ways. Firstly, there is not much point in considering imprecision when there are not even any known algorithms to solve a problem when the input is precise. Secondly, in the early years of computing, data sets were often small (because computers were not fast enough to handle large data sets), and computing was expensive, so imprecision in the data could for a large part be removed by hand, or at least investigated by humans.

In the last twenty years, however, approaches to deal with imprecision algorithmically have slowly started to emerge. In 1989, David Salesin, Jorge Stolfi and Leo Guibas introduced *epsilon-geometry* [61] as a way to cope with computational imprecision in geometric algorithms. It defines a way to reason about the truth of geometric predicates, when each input point is certain to have an imprecision of at most ε: each point could be somewhere else, but not too far away. Figure 1.6 illustrates this model. This simple model has proven very fertile, not only for modelling computational imprecision but also imprecise input data.

After that, many other people have adopted the same model or presented new ones, and many approaches to deal with imprecision have been suggested. Still, these results are mostly scattered individual attempts, compared to the vast body of precise geometric algorithms that are available within computational geometry. By now, many different ways to model imprecision and to tackle its consequences have been introduced. In general, we can say that the more intricate models may resemble the truth more closely, but require a more detailed knowledge of reality and are



(a) (b) (c)

Figure 1.6 (a) A precise point. (b) An imprecise point, modelled as a disk of radius ε. (c) The real point could be anywhere inside the disk.

computationally harder to handle. On the other hand, simpler models often allow fast and accurate computations, but make assumptions that we may know to be inaccurate. In Chapter 2 we will study these possible models and the results that are available in detail.

1.4 Contribution of this Thesis

In the next chapter, we will give an overview of how imprecision can be modelled in the context of computational geometry. We specifically focus on data imprecision, that is, imprecision caused in the first phase of the data flow described above: observing and measuring the world. Important questions to be answered are: What can we actually assume about the input data, when we acknowledge that it is imprecise? How can this data and its imprecision be described? What kind of output do we want algorithms to produce, when there is no longer a single unquestionable true answer? These questions have been answered differently by various people, and indeed the right answers will often depend on the application at hand.

In the remaining parts of this thesis, several detailed solutions to specific problems dealing with data imprecision are presented. It is hard to provide the exact problem definitions here since they depend on modelling issues that are discussed in Chapter 2, but we will give an informal overview of the main ideas.

In Part II, we consider the problem of computing *upper* and *lower bounds* on the outcome of a geometric algorithm, when the input is an imprecise set of points. If a certain geometric problem has a single number as answer, then traditionally we are interested in algorithms that produce that number. However, when the input points are imprecise, then the value of the answer depends on where exactly the points are. In this case, we need algorithms that find the range of possible values of the answer. In particular, in Chapter 3 we consider three shape fitting problems: the smallest enclosing axis-aligned bounding box containing the points, the smallest enclosing circle containing the points, and the narrowest strip (in any direction) containing the points. In Chapters 4 and 5, we consider the problem of computing the convex hull of the points, first in an exact and then in an approximate setting. Finally, in Chapter 6 we consider the problem of computing the diameter of a set of points.

In Part III, we take a different approach to dealing with imprecision. Instead of computing bounds on the outcome of an algorithm, we now assume that even though we have an imprecise set of points, we will later obtain a (more) precise representation of the points. We also assume that once we get the precise points, we want to do some computation on them as fast as possible. The challenge, in this case, is to do as much of that computation as possible now already, while we know only approximately where the points are going to be. In Chapter 7, we show that under certain conditions it is possible to preprocess a set of imprecise points such that when we later get the real points, a triangulation of those points can be computed faster, namely in $O(n)$ time instead of best possible time of $O(n \log n)$ for direct computation. In Chapters 8 and 9, we show that the same result is possible for the Delaunay triangulation, though with more severe restrictions on the input model.

In Part IV, we move away from imprecision in point sets and discuss problems that take a set of lines or a polygon as input. For lines, the question of how to model imprecision becomes more complicated, especially since there is no good definition of a "convex set of lines". In Chapter 10, we discuss the problems of linear programming and vertical extent, both of which take a set of lines as input and produce a single value as output. We present algorithms for computing upper and lower bounds on these values. With more complicated input data such as a polygon, an interesting issue appears when modelling imprecision. It is very common to assume certain properties of composite structures such as a polygon. For example, we may know that it does not intersect itself, or that it is convex. Now, when we try to model imprecision in such a structure, the first thing to deal with is whether these properties are always observed in every possible instance of the input. In Chapter 11 we study the problem of computing bounds on the length of an imprecise polygon, and in Chapter 12 we consider the issue of avoiding self-intersection.

This thesis does not solve the problem of imprecision in computational geometry, but makes a useful contribution to this growing new field. Despite what theoretical computer scientists like to see, this problem cannot be fully captured in mathematics and we cannot hope to solve it completely; it is intrinsic in the world we live in. However, it is certainly possible to produce more meaningful output than simply ignoring imprecision.

Chapter Two

Modelling Imprecise Data

As we have seen in the previous chapter, data imprecision is a serious issue in the design and application of geometric algorithms. In this chapter we will discuss different ways to model this imprecision, and how these modelling choices affect efficiency and correctness.

In the next section, we discuss conceptually what modelling choices there are. In the sections after that, we investigate in more detail the problems that certain modelling choices lead to, and review what results are known about them.

2.1 Imprecision in Input and Output

In essence, there are two important modelling decisions that need to be made. An algorithm has an *input* and an *output*. Traditionally, these are both precise and well-defined geometric structures, or sometimes, in the case of the output, numeric or Boolean values. Now, however, we acknowledge that there is imprecision in the input. Therefore, there is also imprecision in the output. Both of these need to be modelled. That is, we need to define and describe what exactly we know or do not know about the input, and also what exactly we want to know about the output.

2.1.1 Input

In order to define imprecise input data, let us first revisit what precise input data is. We are interested in *geometric* input data. This means the data consists of geometric primitives such as points, lines, line segments, circles and curves, or higher-dimensional equivalents such as planes, balls, etc. Furthermore, there may be relations between

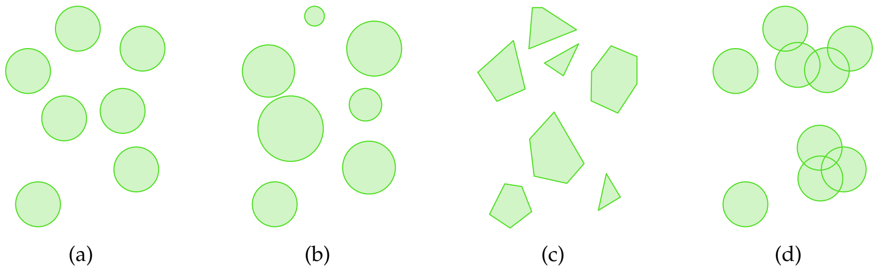

(a) (b) (c) (d)

Figure 2.1 (a) A set of disjoint disks of radius ε. (b) A set of disks of different radii. (c) A set of convex regions of different polygonal shapes. (d) A set of partially overlapping disks.

these primitives specified in the input, in order to create composite structures such as polygons, geometric graphs, arrangements, subdivisions, etc.

Now, an *imprecise* object is an object of which we do not know where it is, what shape it has, how big it is, etc. This seems to imply that we cannot do anything useful with it in any event, but fortunately, we usually do have some idea about all these things. We know approximately where the object is, or approximately how big it is, just not exactly. How can we model this?

Let's focus on the simplest possible geometric object first: a point in the plane. A point has a location, but nothing else: size and shape are not sensible properties of a point. Now, an *imprecise* point is a point of which the location is not known, but we do have some idea about it. Perhaps the simplest way to model this is to assign to the point a set of possible locations, or in other words, a region in the plane. The epsilon-geometry model mentioned in the previous chapter is a special case of this, where the regions are disks of radius ε. But it is also possible to use different shapes for the regions; for example, a square or rectangle may be more suitable when the imprecision in the x- and y-direction comes from different sources.

Usually, the input of a geometric problem consists of a set of n points, and not just a single one, since there is not much to compute about that. Therefore, restrictions on the set of regions may exist, such as that they all have the same shape and size, or that they are not allowed to overlap. These restrictions could also be parametrised. In Figure 2.1 some different sets of regions are shown. Of course, combinations of the restrictions could also occur. In general, which model to use depends on the application at hand, and whether the data can be reasonably assumed to comply with a certain restriction or not. This simple model will be used most in this thesis. In Section 2.2 we give some more insight into it.

When the input data is more complicated than a point set, some additional choices have to be made. When the input consists of a set of other primitives, such as lines, we can use the same approach and model each line as a set of possible lines; see also Section 2.4.1.

In principle, it is also possible to model composite structures in the same way as points or lines. For example, an imprecise triangulation could be represented as a subset of the "set of all possible triangulations". However, such a model is not a easy to handle since it is unclear how to concisely represent such a set. Alternatively, we may present an imprecise triangulation by defining imprecision on the primitives it is built from, and fixing the combinatorial structure. But when we do this, not all inputs may be valid triangulations. We give some more insight into this problem in Section 2.4.2.

Until now, we described imprecision only by saying that something is an element of a given set of possibilities. This imprecision model has the benefit that it is simple, and therefore easy to handle, which may result in efficient algorithms. On the other hand, it may sometimes be too restrictive, since it is not always possible to capture the nature of data imprecision by simply stating the possible values of something. Focussing on point data once again, a more elaborate possibility is to describe each point not as a single region but a set of regions, each with a different likelihood of the point being in that region. Or, one step further, to describe each point by a probability distribution over the entire plane. Then, again we can restrict the model by assuming that all distributions have the same shape, or that they do not overlap too much, etc. We might additionally require the distributions of the points to be independent. Some further discussion of this is given in Section 2.3.2.

In general, a more elaborate model of imprecision can give a more accurate description of the input data, which leads to more precise results, but when modelling imprecise data there are two things to keep in mind. First, we can only model as much as we know, and the more elaborate a model is the harder it becomes to acquire this information. Second, more elaborate means more complicated, thus algorithmically harder, to compute things on it.

2.1.2 Output

Once we have decided on the model of our imprecise input data, the question arises what we want to obtain as output. While the input to geometric algorithms is almost always geometric, the output is often not: it may be just a number, or even a Boolean answer to a decision problem. On the other hand, the output could be a full-fledged geometric data structure too. Clearly, these are quite different scenarios and what we want to obtain as output when there is imprecision in the data will vary a lot.

Generally speaking, what we really want is "the real" output, that is, the output that we would have obtained if we had used "the real" input, without any imprecision. Unfortunately, we cannot get this. So, one logical thing to compute instead is "the most likely" output. However, that alone is often not enough: we would also like to have some measure of how reliable this output is. We can compare this to other fields where imprecise values play a role, such as probability theory or statistical analysis. When we draw a random number from a distribution or a set, the most likely value is represented by the *mean* or *average*, while a measure of reliability can be supplied by the *variance* or the *standard deviation*.

One way to get a "likely" output is to use somehow the most likely input, and just compute the precise output on that. This is an approach that is often used, because it is very simple; it is the equivalent of ignoring the imprecision in the input data. It will be clear though, that this yields no guarantees to either the value or reliability of the output. On the other hand, strictly following probability theory and computing the average value of all parameters of the output based on probability distributions of all parameters on the input is often computationally infeasible, and also requires very detailed knowledge of the imprecision at hand. Still, we would like to return some reasonably likely output or small set of likely outputs.

To determine the reliability of an output, we need a way to measure the distance between the reported output and the true output. When such a measure is available we could again try to compute the equivalent of the standard deviation, that is, the expected deviation of the expected value. Alternatively, we could simply output a set of possible outputs, or explicit bounds on the possible deviations of the output.

It is hard to say much more in general about what output is interesting to compute, since there is such a large variety of things to compute, and even for one specific construction, there are often many different aspects that one could be interested in. Therefore, let us look at a concrete example.

2.1.2.1 Example: Smallest Enclosing Circle

Consider the geometric structure known as the *smallest enclosing circle (SEC)* of a set of points, as shown in Figures 2.2(a) and 2.2(b). The input to this problem is a set of points, and the output is a circle. This is a very basic geometric structure, and algorithms for it have been studied since the beginning years of computational geometry. A classical result is that the SEC can be computed in $O(n)$ time [93].

The output, as mentioned, is a circle. However, there are several ways to describe this circle. One way is by three real numbers, two defining the *location* of (the centre point of) the circle, and the third defining its *size* (radius). But, alternatively, we could output the indices of the points in the input that "define" the smallest enclosing circle, that is, the points that lie on this circle. When the input points are in general position,[1] there are always two or three defining points, and the receiver of the output can easily reconstruct the actual circle through these points if he or she so desires. We could say that the first type of output is the *numeric* type, while the second is the *combinatorial* type. When the input is precise, it does not really matter which way we choose, since

---

[1] Assuming *general position* is a common way in computational geometry to avoid special cases (such as, in this case, four points that happen to lie on one circle) that complicate algorithm descriptions, thus allowing the algorithm to focus on the fundamental part of the problem.

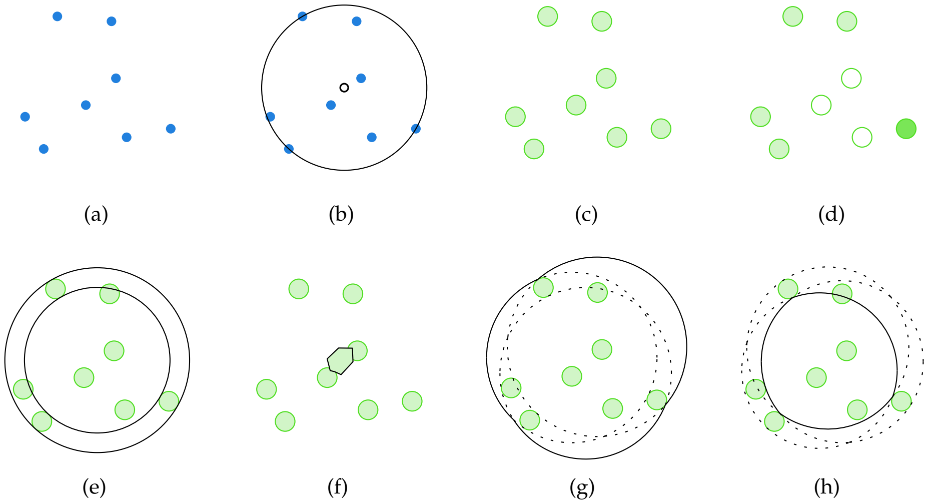


Figure 2.2 (a) A set of precise points. (b) The smallest enclosing circle (SEC) of the points, with three points on the circle and with the circle centre shown. (c) A set of imprecise points, modelled as disks. (d) A classification of the disks specifying which disks are certain (dark), possible (light) or impossible (white) to define the SEC. (e) The smallest and largest possible values for the radius of the SEC. (f) The region of possible locations of the centre point of the SEC. (g) The boundary of the union of all possible SECs. (h) The boundary of the intersection of all possible SECs.

there is no difference in computational cost and the user of the algorithm simply uses whatever information he or she is interested in. But when defining imprecise output, these different viewpoints actually lead to problems that have very different solutions and complexities.

When the input is imprecise, both types of output become imprecise as well. Consider imprecise input under the simplest model available, namely a set of disjoint disks of equal radius (say ε). Figure 2.2(c) shows such a set of regions. When we look at the combinatorial elements of the output, these can be classified as *certain*, *uncertain/possible* or *impossible* to be part of the output. In the example, the SEC is defined by three of the input points. For all points, we can indicate that they are either certain, possible, or impossible to be one of those three. Figure 2.2(d) shows this classification for the example set of regions.

Numeric values, on the other hand, have a *range* of possible values. In the example: the radius of the SEC becomes an interval of possible radii. Figure 2.2(e) shows the smallest and largest possible SEC of the set of imprecise points. Similarly, the centre of the SEC becomes a region of possible locations (you could say: an imprecise point), see Figure 2.2(f). Additionally, a combination of combinatorial and numeric information could be interesting. We could also view the output of the SEC problem as a subset of the plane: the set of all points inside the SEC of the input points. In this case, when the input is imprecise, we can compute the regions in the plane consisting of all points that are certain or impossible to be inside the SEC. Figure 2.2(g) and 2.2(h) show this for the example regions.

One might argue that the problem of computing the smallest enclosing circle of a set of imprecise points is solved only when *all* these things have been computed explicitly. However, the computational complexity for computing these properties may differ, and a user of the algorithm may not care about one or more of them. Therefore, what we need is a thorough investigation into the computational complexities of these things, so that the user can decide what he or she wants and whether he or she finds the time investment worth it or not.

This example shows a basic geometric problem in the simplest imprecision model available. It will be clear that with more elaborate input models, or more complicated geometric structures, the possibilities only increase. This makes the analysis of geometric algorithms with data imprecision such a complicated and challenging task: it may be impossible to ever provide a complete classification of all possible variants. The challenge, then, is to identify which variants are most important and useful and tackle them first. In the remainder of this chapter, we will survey what results have already been obtained in the ten or twenty years that people have studied these problems, and see how they relate to each other.

2.2 Imprecise Points Modelled as Regions

The simplest form of geometric data is a point. And the simplest way to represent an imprecise point is by a region. Not surprisingly in a relatively new direction of research, and considering the vast number of interesting problems that even a construction as simple and long-solved as the smallest enclosing circle already give, most attention has gone to this simplest setting.

Additionally, this simple model has the advantage that once the input and desired output are specified, the problem becomes a classical geometric problem again. This means that we do not have to rebuild our techniques from scratch, but can build upon the existing wealth of geometric techniques. Even more interestingly, some problems that arise from imprecision have already been solved by people who studied the resulting algorithic problems from a very different motivation.

As described in the previous section, there are several different types of output that one could be interested in. We now review them in more detail.

2.2.1 Boolean Output

The most basic type of output an algorithm can have is a single Boolean value. Such algorithms, called *decision algorithms*, answer a single question about the input data: does a certain property hold or not? This question has only two possible answers: yes or no. However, when the input is imprecise, there are three possible answers: yes, no, or maybe. This corresponds to the situations where the property is certain to be true, certain to be false, or either is possible depending on the precise locations of the points.

Problems of this kind have not been studied extensively in the presence of data imprecision, but they are often so fundamental that they have been solved as abstract problems already. For example, consider the question whether the points in a given set are collinear or not. When the points are imprecise, this can be formulated as whether there exists a line that intersects all input regions. This is known in the literature as the *transversal* or *stabber* problem, on which a lot of results are available, see e.g. Edelsbrunner [38].

As another example, consider the question whether a set of points is in convex position. Goodrich and Snoeyink [55] study a problem where they are given a set of parallel line segments, and must choose a point on each segment such that the resulting point set is in convex position. This corresponds to the question whether a set of points that have 1-dimensional imprecision is in convex position. They present an algorithm that determines whether solutions exist, and finds one if they do, in $O(n \log n)$ time.

2.2.2 Combinatorial Output

Often, the output of a geometric algorithm can be specified in terms of relations between the input points alone, without specifying any actual numerical information. In this case, when the input is imprecise, we may answer a decision question for each possible combinatorial feature, or equivalently, enumerate all features that could have a certain property.

Not much work has been done in this direction. One notable exception is formed by Bandyopadhyay and Snoeyink [10, 11], who compute for a given set of points all triplets (in the planar case) that could form a Delaunay triangle when the points are moved by at most ε. They call these sets the *almost-Delaunay simplices* of the point set, which can be seen as the possible triangles in the Delaunay triangulation of a set of imprecise points modelled as disks of radius ε. They show applications to the problem of folding proteins.

2.2.3 Numeric Output

Another type of output that a geometric algorithm can have is a single numeric value (or possibly multiple numeric values). If the input is a set of imprecise points modelled as regions, such a value is no longer precise, but becomes a set of possible values. This set is usually a continuous interval of values, although this is not necessarily the case. In this situation, we would like to compute the bounds of this interval, that is, the smallest and largest possible values that the function under study can attain.

There are several measures that aim to capture the extent of a point set, including the diameter, the width, the radius of the smallest enclosing circle, and the perimeter of the smallest enclosing bounding box. For each of these, it would be interesting to compute the smallest and largest possible value. For some of these problems, existing results are already available. For example, the smallest possible radius of the smallest enclosing circle of a set of regions, is known as the *intersection radius* of these regions. Jadhav *et al.* [70] show how to compute this for a set of polygonal regions in $O(n)$ time. Similarly, Colley *et al.* [32] compute the smallest area axis-aligned rectangle that intersects a set of convex polygons in $O(n \log n)$ time. Robert and Toussaint [113] develop an algorithm for computing the smallest strip that intersects a set of convex regions, while surveying several facility location problems. Such stabbing and facility location problems can also be translated to minimisation problems of functions on imprecise points. Van Kreveld and Löffler [90] study these extent measures from the viewpoint of imprecision, and provide several additional algorithms, as well as hardness results. Kruger and Löffler [78, 79] study the same problems in higher dimensions.

Another basic geometric measure is the closest pair in a point set, that is, the smallest distance between any pair of points among the input. Fiala *et al.* [45] consider the problem of finding distant representatives in a collection of subsets of a given space. In particular, they prove that maximising the smallest distance in a set of n imprecise points, modelled as disks or squares, is NP-hard. Cabello [20] gives approximation algorithms for this case. When two distinct point sets are given, a popular way of computing the “difference” between them is the *Hausdorff* distance. Knauer *et al.* [77] compute upper and lower bounds on the directed Hausdorff distance between two point sets, one or both of which can be imprecise.

An important geometric structure is the convex hull. This is not something that can be fully described by a single number, but its size can be measured by computing, for example, its area or perimeter. Van Kreveld and Löffler [87, 89] study the problem of computing upper and lower bounds on these measures. Ju and Luo [71] improve one of these results, and also consider some variations in the model of imprecision. Some of the resulting problems here have also been studied in a different context. Mukhopadhyay *et al.* [98, 99] study the largest area convex polygonal stabber of a set of parallel or isothetic line segments, and similarly, Hassanzadeh and Rappaport [64, 112] study the shortest perimeter convex polygonal stabber of a set of line segments. Boissonnat and Lazard [17] study the problem of finding the shortest convex hull of

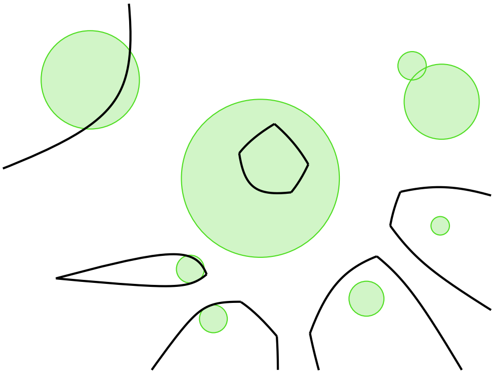


Figure 2.3 The regions are the sets of points that are certain to be in the Voronoi cells of the given imprecise points. Note that two of the disks intersect, and their corresponding cells are empty.

bounded curvature that contains a set of points, and they show that this is equivalent to finding the shortest convex hull of a set of imprecise points modelled as circles that have the specified curvature. They give a polynomial-time approximation algorithm.

In Part II of this thesis, we present detailed solutions to the problem of computing upper and lower bounds on several geometric measures.

2.2.4 Combination of Numeric and Combinatorial Output

Many geometric algorithms return a certain region in the plane, or a subdivision of the plane into regions. In these cases, a natural way of representing imprecise output is by providing two boundaries of such a region: an inner boundary that encloses all points that are certain to be in the region, and an outer boundary that encloses all points that could be in the region. Such a pair of boundaries is sometimes referred to as a *fuzzy* boundary.

Nagai and Tokura [103] follow this approach for the convex hull by computing the union and intersection of all possible convex hulls. As imprecision regions they use disks and convex polygons, and they give an $O(n \log n)$ time algorithm for computing both boundaries.

When the output is a subdivision of the plane rather than a single region, the most interesting task is probably to compute the inner boundaries of all regions. Figure 2.3 shows an example for the Voronoi Diagram. Khanban [73] develops a theory for returning such partial subdivisions, for Delaunay triangulations and Voronoi diagrams. This is based on a redefinition of the *in-circle test*, introduced by Khanban and Edalat [74], for imprecise points modelled as rectangles. Ely and Leclerc [43] study similar problems for circular imprecision regions, and Sember and Evans [119] also define partial Voronoi diagrams, similar to Khanban.

2.3 Other Models for Imprecise Points

Sometimes, a single region of possible locations for a point is too restrictive as a model. In some applications, more information is known about the imprecision, such as dependence on a limited number of parameters, or (estimated) probability distributions over the locations of the individual points.

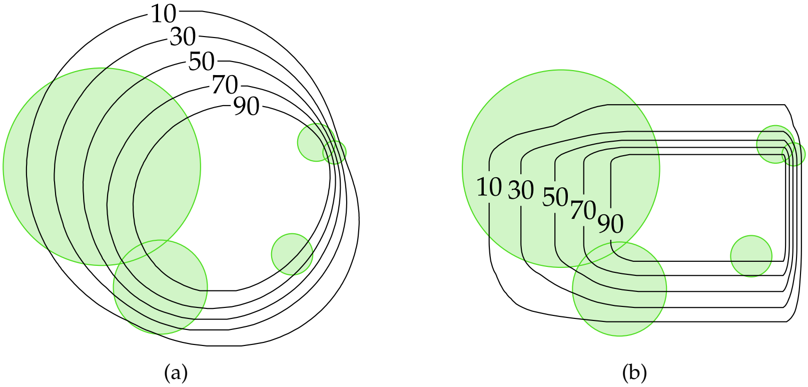
In these cases, the methods for models based on regions could still be used by choosing an error distance beyond which the probability is below a certain threshold, and using the remaining part of the plane as region. However, the disadvantage of this is that the solutions produced by those methods will depend on the choice of this threshold. Furthermore, the solutions depend heavily on the boundary cases of the error model, while it is reasonable to expect the points are more likely to appear near the “centre” of the regions.

Working directly with more involved models of imprecision can provide more accurate answers to geometric questions about such sets of points. Of course, the downside is that algorithms become more complicated, and probably take longer to run.

2.3.1 Dependence

One assumption that we have made so far about imprecise points is that the imprecision of all points is independent. This may be true in some situations, or we may not know anything better than that. In certain situations, though, we do know that the positions of points are correlated, for example when they have been measured with a consistently erroneous measuring device. In such cases, it is good if this dependence can be incorporated in the imprecision model, since this should yield much more accurate results.

Myers and Joskowicz introduce the *linear-parametric geometric uncertainty model* [101, 102], where the positions of imprecise points within their regions depend on a small set of common parameters. Within this model, they study the maximum and minimum values of geometric measures, such as the diameter and closest pair of a set of points. In an earlier work, Ostrovsky-Berman and Joskowicz [107] study the union of all possible convex hulls when the imprecision of the points depends linearly on a limited number of parameters.



(a) (b)

Figure 2.4 (a) The shape inclusion probability for the smallest enclosing ball, for points uniformly distributed inside the circles. (b) The same, but showing the smallest enclosing axis-aligned bounding box.

2.3.2 Probability Distributions

Instead of modelling imprecise points by a small region of possible locations, it is also possible to describe such a point by a probability distribution over the whole plane, which specifies for every point in the plane the exact chance that the real point is at that location. If such information is available, it is possible to compute more precise information about the output than with the simple region-based models.

One approach, inspired by the statistical law of large numbers, is to solve the problem by just sampling many input point sets, run a deterministic algorithm on these, and do a statistical analysis on the results, for example, computing the average and standard deviation of some measure. This is of course a randomised approach, but can be expected to work well. The drawback is that it may take a lot of computation time.

If the output is a numeric value, though, we can even go one step further and provide a complete probability distribution of this value based on the distributions of the input points. To compute this exactly is infeasible for almost all interesting measures, but it is possible to compute approximations of these distributions. Löffler and Phillips [85] provide simple randomised algorithms to compute such distributions. Furthermore, if the output is a region in the plane, they compute *shape inclusion probability functions*: a density function over the plane that assigns to every point in the plane the probability that that point is inside the region. This corresponds to the partial diagrams discussed in Section 2.2.4. Figure 2.4 shows two examples of such shape inclusion functions, depicted using isolines of the distributions.

2.3.3 Preprocessing

Often, the outputs produced by repeatedly generating an input according to the same probability distributions will be very similar. Sometimes it is possible to remember some information about earlier outputs, and use this to speed up the precise computation on the next inputs. Clarkson and Seshadri introduce the notion of *entropy* [31] as a way to measure the similarity between various Delaunay triangulations of point sets drawn from the same distribution. They show how information about the triangulations can be stored and used to eventually speed up the computation time.

If the distributions are restricted to regions in the plane, it is also possible to preprocess these regions explicitly, such that when a point set drawn from the distributions is given, some structure can be computed faster than when redoing the whole computation each time, independent of the distribution. Held and Mitchell [65] study a problem of this type, which they call *input-constrained* geometry. Given a set of disjoint unit disks in the plane, they show how to preprocess them in $O(n \log n)$ time into a data structure, such that a point set drawn from the disks can be triangulated in linear time given this data structure. Van Kreveld *et al.* [127] generalise this result to any set of disjoint regions in the plane.

Löffler and Snoeyink [86] study the same problem for the Delaunay triangulation. They show how to preprocess a set of disjoint unit disks in $O(n \log n)$ time so that the Delaunay triangulation can be computed in linear time, and they also remark that this result cannot be generalised to general disjoint regions. However, Buchin *et al.* [19] show that for several realistic input models, such as partially overlapping or "thick" regions, the same time bounds can still be obtained using randomised algorithms, with optimal dependency on the realism parameters. In Part III of this thesis, we provide the details of some of these algorithms.

These results are also related to the *update complexity* model. In this model, a set of imprecise points is given, but it is assumed that the points can be obtained during the execution of the algorithm with higher precision (that is, a smaller region), at a certain cost. The goal is then to compute a certain combinatorial structure while minimising the cost of updates [18, 47].

2.3.4 Outliers

A somewhat different way to model imprecision in point sets is to represent each point by an exact point, but accompanied by a probability p that it really is where we think it is. Then, with probability $1 - p$ it is elsewhere, and we make no assumption on where. This is useful, for example, for modelling measuring equipment that can make very big errors, but does so only rarely. This results in the theory of *outliers*. When the set of input points is large, we expect a fraction p of them to be measured correctly, while the remaining $k = n - pn$ points could be anywhere.

When describing the shape of a point set with traditional geometric objects such as a bounding box, a minimum enclosing circle, or the convex hull, such outliers can greatly disturb the shape. This leads to problems of this type: find the smallest possible box/circle/convex polygon such that at most k of the input points are *not* in this region. Har-Peled and Wang [63] study this problem for many different geometric shapes, and provide a general framework for solving such problems. The theory of outliers in computational geometry is extensive, and actively being developed [4, 8].

While the model of a point being really correct with probability p may not be the most suitable error model in most applications, even if we allow less strict distributions of the points, such as, for example, normal distributions, it may still be a good idea to develop algorithms that take outliers into account, since this may help to make, for example, the computation of the average more stable.

2.4 Imprecision in Other Types of Input

So far, all models discussed are for problems where the input is a set of points (either in the plane or in higher dimensions). However, many real-world applications provide more structured input, or work on other geometric primitives. Even though most attention by far has gone to the imprecision in points, there are also some results available in other areas.

2.4.1 Lines

Apart from points, perhaps the next most basic geometric object is a line. To model an imprecise line, we could follow the same approach as with points, and represent each line as a set of possible lines. However, to make this a practical model, it would be good to put some restrictions on the shapes of these sets of lines. In the case of points, one often assumes that these sets are *connected* and perhaps *convex*. Additional properties to make the regions easier to handle may include *constant description size*, or *piecewise linear* (polygonal) boundaries. We would like to have similar properties for imprecise lines.

We will distinguish between lines and *directed* lines. Problems that take a set of lines as input usually treat them as either directed or undirected lines. There is an important topological difference between the set of all lines in $\mathbb{R}^2$ and the set of all directed lines in $\mathbb{R}^2$: the former is isomorphic to the Möbius-strip, while the latter is isomorphic to a cylinder.

Connectedness is natural to define for a set of lines: if two lines can be transformed into each other by a continuous movement inside the set, then they are connected.

Convexity of sets of lines is harder to define. This subject has been studied by various people, and several different definitions have been proposed. A first approach is to

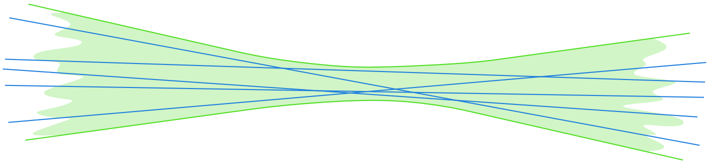


Figure 2.5 A “convex” set of lines. The set contains all lines that stay completely within the grey area.

define convexity based on point-line duality [39] and the concept of convexity for point sets. The straightforward way to do this has the drawback that vertical lines cannot be represented. Rosenfeld [114] proposes a definition that gets around this problem and has nice properties, but which is not translation-invariant. Goodman [54] argues that no natural definition can exist, and gives a definition that drops the property of connectedness of convex sets. Gates [51] defines convexity for sets of *directed* lines in a natural way. Bhattacharya and Rosenfeld [16] give an extensive comparison between the various definitions.

With the idea in mind that we want to represent imprecise lines with our convex sets, it seems reasonable to assume that for a given imprecise line, there is at least one direction in which the line certainly does *not* lie. Van Kreveld and Löffler [88] suggest a definition that takes this into account, which corresponds to the duality-based definition except that the “vertical” direction can be different for each line. They also show that such a set can be represented in a natural way as the set of all lines that lie completely within the portion of the plane bounded by two curves. Figure 2.5 shows a convex set of lines under this definition.

The other properties we mentioned, constant description size and piecewise linear boundaries, are now also easy and natural to define for convex sets of lines: if the defining curves are of constant description size or polygonal, then so is the set of lines. Van Kreveld and Löffler [88] study the problems of linear programming and vertical extent under this model. These results are presented in detail in Chapter 10.

There are of course also different possibilities to naturally describe imprecise lines. Another natural model would be the set of lines stabbing two given regions (imprecise points) in order. Imprecise linear programming also occurs in other fields, for example in biology [13, 69] These uses suggest the *axis-model*: an imprecise line is the collection of lines that intersect the x- and y-axes in certain fixed intervals.

2.4.2 Composite Structures

When dealing with composite geometric structures, it is less clear how to model an imprecise instance. Perhaps the simplest composite geometric structure is the polygon:

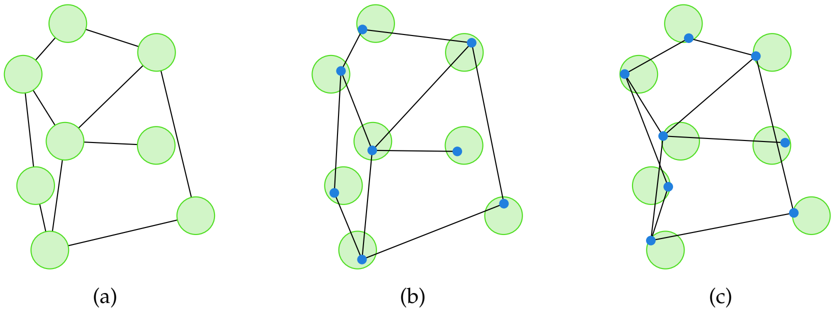


Figure 2.6 (a) A set of imprecise points, with an additional graph structure on them. (b) A possible instance of the graph. (c) Another possible instance. Note that this one intersects itself.

a sequence of points connected by line segments. Other examples are geometric graphs or triangulations, planar subdivisions, or higher-dimensional constructions.

Analogously to points, we could model an imprecise polygon by the set of all possible polygons. However, the question then becomes what such a set looks like, and whether it can be represented and stored concisely. Alternatively, we could model such a polygon by defining imprecise versions of the components of the structure: the points and the line segments. If we do that, though, it may be hard to ensure that all possible resulting composites are consistent. For example, a geometric graph may be required to have no self-intersections.

Somehow, both of these viewpoints come down to the same thing, if we describe the set of “possible graphs” as the subset of the set of graphs that arises from making the vertices imprecise and that satisfies a set of additional properties that we require the graph to have. Figure 2.6 shows an example of such an imprecise geometric graph where the vertices are modelled as imprecise points. Two possible instances are shown, one of which has self-intersections.

One fundamental geometric graph is the polygon. Finding an instance of an imprecise polygon corresponds to finding a path that visits a number of regions in the correct order. Dror *et al.* [37] study the problem of computing the shortest path that visits a number of regions in order. They give an $O(n^2 \log n)$ time algorithm if the regions are disjoint convex polygons, and prove that it is NP-hard for non-convex regions. Polishchuk and Mitchell [109] extend this result to higher dimensions. Arkin *et al.* [6] study the same problem for line segments and the L_1 metric. In these problems, a fixed starting point for the path or tour is required. These problems are also related to the Safari Keepers problem [105, 125], where all regions are required to be inside a polygonal domain, and adjacent to its boundary.

In these results, the tour is often seen as a path that is traversed by something or somebody. Conversely, when the tour is used to model an imprecise polygon, we often assume that this polygon should be *simple*: there should be no intersections between its edges. A fundamental question then becomes, when the vertices are imprecise, whether the corresponding imprecise polygon is still simple. Löffler [84] studies this question and shows that deciding whether it is possible to place the vertices such that the resulting polygon is simple, is NP-hard. In Part IV of this thesis, some results regarding this are given in detail.

When the imprecise polygon is indeed simple, we can start solving problems on simple polygons. Cai and Keil [21] study visibility in an imprecise simple polygon, where each vertex lies in a disk of radius ε. They define the *visibility skeleton* as the graph that has an edge between two vertices if these two vertices can see each other in any instance of the polygon, and show that this can be used to compute constant-factor approximations of shortest paths. Stewart [124] studies how to do robust point location in imprecise polygons, although he is mainly concerned with computational imprecision rather than data imprecision.

Another fundamental geometric graph is the triangulation. When the input is a triangulation but the vertices are imprecise, we need to ensure that the resulting graph is indeed a valid triangulation. For the Delaunay triangulation, the combinatorial structure of the triangulation may change as the points move, even without causing intersections. Abellanas *et al.* [1] and Weller [128] define the *tolerance* of a geometric structure as the largest perturbation of the vertices such that the topology of the structure is guaranteed to stay the same. They focus mainly on the planar Delaunay triangulation, and show that its tolerance can be computed in linear time. They also study several subgraphs of the Delaunay triangulation.

2.4.3 Imprecise Attributes

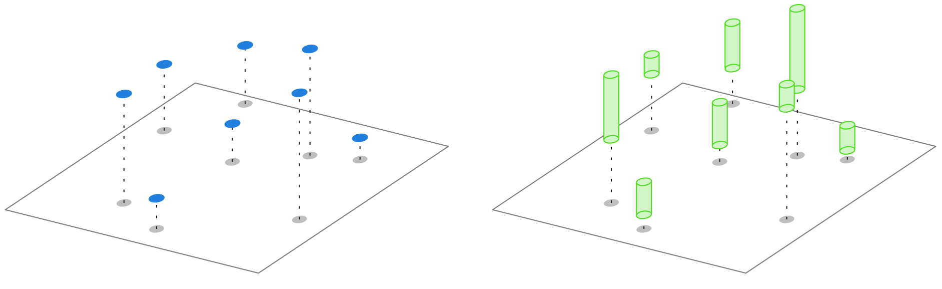
Often, a geometric data set consists not only of a set of points, but a set of points with certain *attributes* on them. For example, we may measure the soil humidity on several locations in the plane, or the elevation above sea level. In this case, not only the position of a point may be imprecise, but also the value of the attribute. In such a case, we may view the attribute value as an extra dimension, so a point set in $\mathbb{R}^d$ with imprecise attributes becomes an imprecise point set in $\mathbb{R}^{d+1}$.

In fact the imprecision of the attribute is often far greater than that of the locations.[2] For example, in high-resolution terrains distributed by the United States Geological Survey, it is not unusual to have vertical errors of up to 15 meters [126]. This suggests that we could assume that the locations are actually precise, and only the attribute values are imprecise. The regions of the imprecise points in $\mathbb{R}^{d+1}$ then becomes a set of vertical line segments. Figure 2.7 shows an example.

In a way this is no different from the models in the previous section, since we are still dealing only with points sets. But, the imprecision is of a very specific kind.

---

[2] In a relative sense, since the error of attribute values cannot be directly compared to the error in space.


(a)

(b)

Figure 2.7 (a) A set of points in the plane with a numeric attribute can be interpreted as a set of points in $\mathbb{R}^3$. (b) When the attribute value is imprecise, the point set becomes a set of vertical line segments.

Furthermore, often not only a point set is given but also some structure on it, such as a triangulation. In this case we get what we may call an *imprecise polyhedral terrain*.

Gray and Evans [58] propose a model where an interval of possible heights is associated with every vertex of a triangulation. They study the problem of computing the shortest non-ascending path between two points on the terrain, where the path has to stay inside the height interval of every vertex it visits. Kholondyrev and Evans [75] also study this model. Silveira and Van Oostrum [121] study the problem of removing local minima in a terrain by moving the vertices up and down, but this model is slightly different since they assume no bounds on the imprecision, but rather minimise the total displacement of the vertices. Gray *et al.* [59, 60] study the problem of finding the “smoothest” instance of an imprecise 1.5-dimensional terrain.

2.5 Closing Remarks

Many different ways of making data imprecision in geometric problems mathematically precise have been studied. The two main modelling questions are how to describe the imprecision in the input, and what we want to know about the resulting imprecision in the output. The different answers to these questions lead to an array of algorithmic problems, most of which have never been considered yet. Nonetheless, the number of results, especially concerning imprecise points, is growing steadily.

This finishes the introductory part of this thesis. In the remainder, detailed algorithms and hardness proofs for several of the problems mentioned in this chapter are presented. In Part II, we consider computing upper and lower bounds on geometric problems that have a numeric value as output, when the input is a set of imprecise points. In Part III, we investigate whether it is possible to preprocess a set of imprecise points such that when the real points become available later, certain computations can be speeded up. Finally, in Part IV, we study problems that take a set of imprecise lines or an imprecise polygon as input.

PART II

Bounds on Output Imprecision

Chapter Three

Bounds on Shape Fitting

In this part of this thesis, we consider the problem of computing lower and upper bounds on the possible output values of geometric problems that take a point set in the plane as input and produce a single number as output, when the points are imprecise. This was also described in Section 2.2.3.

We assume that the "real" input point set $P = \{p_1, \ldots, p_n\}; p_i \in \mathbb{R}^2$ is unknown, and that instead we are given a set of regions $\mathcal{R} = \{R_1, \ldots, R_n\}; R_i \subset \mathbb{R}^2$ and the guarantee that for every i we have $p_i \in R_i$. We are interested in the value of a certain function μ that takes a point set as input and produces a single number as output, that is, we want to know $\mu(P)$. Since P is unknown, what we want to compute instead is the smallest and largest possible value that μ can attain on any point set P that complies with $\mathcal{R}$, that is, we want to find point sets $P^{\min}$ and $P^{\max}$ that each consist of one point from every region in $\mathcal{R}$, such that $\mu(P^{\min})$ is the smallest value that μ attains on any such set and $\mu(P^{\max})$ is the largest such value. Note that such point sets are generally not unique: often the value of μ is only defined by a small number of points, and is not influenced by the precise location of the rest of the points.

We restrict the regions of the points to have the same shape, and in particular study the case where these shapes are squares or disks[1]. These shapes are arguably the most natural choices that occur in practice. The circular model occurs when the points are accompanied by a single error parameter ε, and there is no reason to assume that one direction of error is more likely than another. The square model could occur when points have been stored as floating point numbers, where both the x and y coordinates have an independent uncertainty interval, or with raster to vector conversion. Aside from this practical motivation, these shapes are easier to handle and already provide sufficient challenge, as will become clear later.

---

[1] Or, if the reader prefers, balls in the L_1 and L_2 metric.

problem	model	largest	smallest
smallest bounding box	squares disks	$O(n)$ $O(n)$	$O(n)$ $O(n^2)$
smallest enclosing circle	squares disks	$O(n)$ $O(n)$	$O(n)$ [70] $O(n)$
minimum width strip	squares disks line segments	— — NP-hard	$O(n \log n)$ [113] $O(n \log n)$ $O(n \log n)$ [113]

Table 3.1 New and known results on shape fitting problems.

One common operation in computational geometry is to capture the "shape" of a set of points by fitting a simple predefined geometric object around it. In this chapter, we study three basic shape fitting problems on point sets in the plane: the smallest enclosing axis-aligned bounding box, the smallest enclosing circle, and the minimum width enclosing strip in any direction. Each of these definitions implies a notion of *size* to define the "smallest" or "minimum" shape. We measure the size of a bounding box by its area, the size of a circle by its radius, and size of a strip by its width. Now, the size of the smallest shape that fits around a set of points is an example of a function μ as described above.

For each of the three classes of shapes, we study the algorithmic questions of maximising and minimising the size of the smallest shape. Furthermore, we consider two different models for the imprecise points: squares and disks. This leads to a total of 12 different problems, which are shown in Table 3.1, together with the efficiency of our solutions. Some of those problems have already been studied with a different motivation. For most of the other cases we were able to construct efficient algorithms. The exception is the problem of computing the largest possible width. For this, we have not found any satisfying result for the square or disk model, although we can prove that the problem is NP-hard when the imprecise points are modelled as line segments.

3.1 Axis-Aligned Bounding Box

We start with a relatively simple problem. Given a set of points P, the *axis-aligned bounding box (AABB)* is the smallest axis-parallel rectangle that contains P, see Figure 3.1(a). In an imprecise context, we are given a set $\mathcal{R}$ of regions, and we want to place a point in each region such that the bounding box of the resulting point set is as large or as small as possible, see Figures 3.1(b) and 3.1(c). We will measure the size of a rectangle by its area.

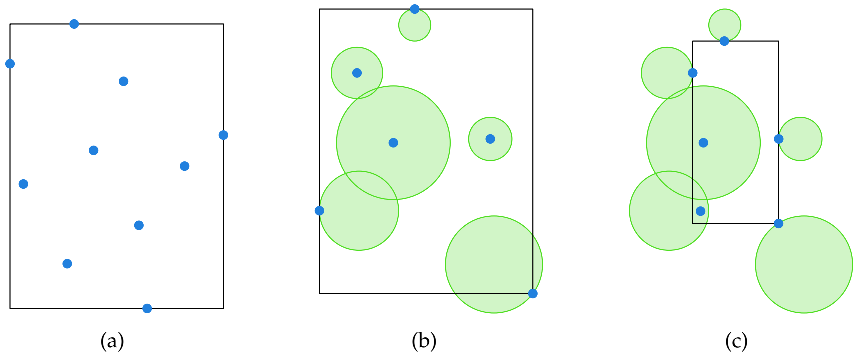

(a) (b) (c)

Figure 3.1 (a) The axis-aligned bounding box of a set of points in the plane. (b) The largest possible AABB of a set of imprecise points. (c) The smallest possible AABB of the same set of imprecise points.

3.1.1 Largest Possible AABB

In this section, we consider the following problem:

Problem 3.1 *Given a set of disks or squares in the plane, place a point in each region such that the area of the axis-aligned bounding box of these points is maximised.*

The largest possible AABB can be computed in linear time for both the square and disk model (or, in fact, any other constant-complexity model). Let $\mathcal{R}' \subset \mathcal{R}$ be the set of the four farthest regions from $\mathcal{R}$ in each of the four axis-parallel directions, making sixteen elements in total.

Lemma 3.1 *The largest possible AABB of $\mathcal{R}'$ is equal to the largest possible AABB of $\mathcal{R}$.*

Proof Suppose this is not the case. Then there is a region l in $\mathcal{R} \setminus \mathcal{R}'$ that contributes to the AABB of $\mathcal{R}$, say to the top boundary. However, there are at least four regions in $\mathcal{R}$ that extend higher than l, of which only three can contribute to another boundary. That means we can place the point of the fourth at its topmost position, and we have a larger AABB. This contradicts the assumption. ■

The AABB of $\mathcal{R}'$ can be determined by four points each lying on one of the sides of the bounding box, or by only three or two points when one or two points lie on corners. Since $\mathcal{R}$ has only constant size, we can try all possibilities and report the largest one. For a set of at squares with the extreme points of the bounding box assigned to them, it is trivial to compute the largest box; for a set of discs this involves solving a polynomial of degree at most 4.

Theorem 3.1 *Let $\mathcal{R}$ be a set of n disks or squares in the plane. We can compute in $O(n)$ time a point set P containing one point from each region in $\mathcal{R}$, such that the area of the axis-aligned bounding box of P is maximised.*

3.1.2 Smallest Possible AABB

In this section, we consider the following problem:

Problem 3.2 *Given a set of disks or squares in the plane, place a point in each region such that the area of the axis-aligned bounding box of these points is minimised.*

We will first define a rectangle R that has to be in the interior of any solution. Let the *left extreme line* be the leftmost of the lines through the rightmost points of all regions. Similarly we define the *right*, *top* and *bottom* extreme lines. When the left extreme line is to the right of the right extreme line, or the top extreme line is below the bottom extreme line, there exists a zero area solution. We can check this easily in linear time, so from now on we assume that this is not the case. Then, the for extreme lines define a rectangle R.

The smallest possible AABB is the smallest rectangle that contains at least one point of each region, so it is actually the smallest rectangle that intersects all regions. Note that this rectangle must indeed contain R.

3.1.2.1 Squares

When the points are modelled as squares, we know that every square must intersect R. Therefore, R is already the smallest AABB.

Theorem 3.2 *Let $\mathcal{R}$ be a set of n squares in the plane. We can compute in $O(n)$ time a point set P containing one point from each region in $\mathcal{R}$, such that the area of the axis-aligned bounding box of P is minimised.*

3.1.2.2 Disks

When the imprecise points are modelled as disks, R does not necessarily intersect all disks, see Figure 3.2(a). Colley *et al.* [32] study the same problem for a set of convex polygons, and they obtain an $O(n \log n)$ time algorithm. We can use similar techniques to get a quadratic-time algorithm for the disk case.

We do know that R needs to be contained in the smallest AABB, and therefore we no longer need to consider the disks that intersect R. The centre points of the remaining disks lie outside the two strips between the extreme lines; this partitions them into four groups. Consider the top right group; the other groups are treated symmetrically. For each disk in that group, we define the region where the top right corner of the AABB must lie so that the AABB intersects the disk. Take the bottom left quarter-disk boundary, and extend it to an infinite curve using a half-line vertically upwards from the topmost point and a half-line horizontally rightwards from the rightmost point. This results in the boundary of a rounded quadrant; the curve divides the locations of the top right corner of the AABB into the interior, where the AABB intersects this

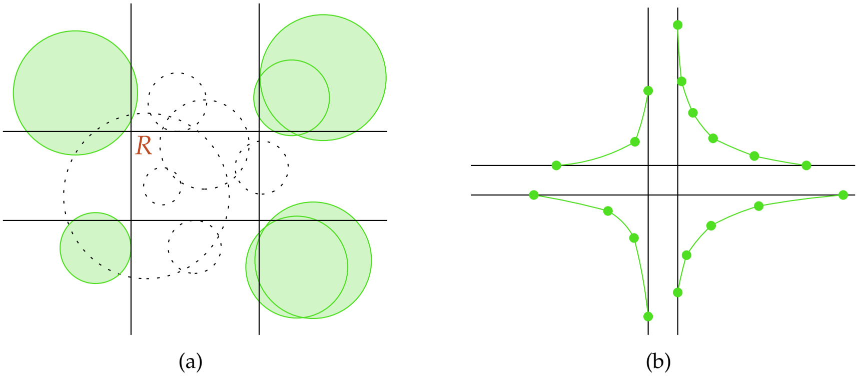


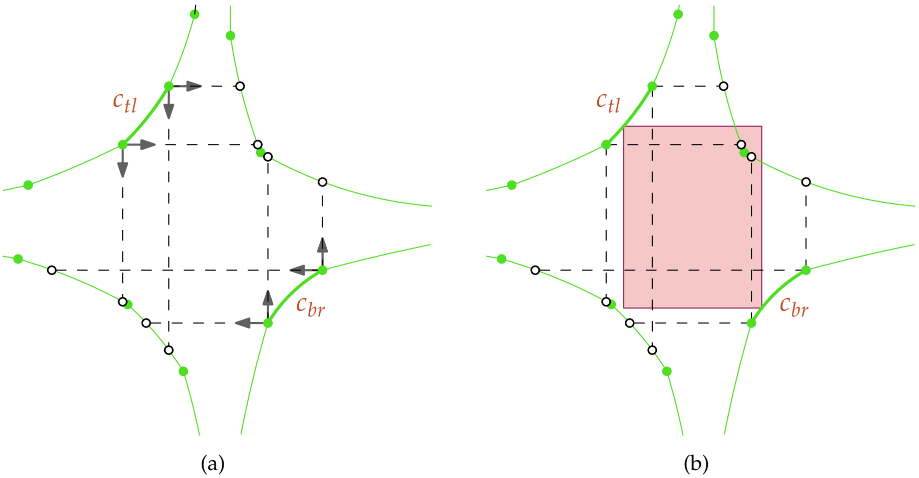
Figure 3.2 (a) The disks that intersect the rectangle R (bounded by the extreme lines) are always accounted for. (b) The remaining disks form four chains of circular arcs.

disk, and the exterior, where it does not. We compute this curve for each disk in the top right group, and then compute the boundary of the common interior, giving the top right chain. We perform symmetric steps for the other three groups. Each of the four chains is a convex chain of $O(n)$ circular arcs and two half-lines, see Figure 3.2(b). We can compute these chains in $O(n \log n)$ time by first computing the common intersection of the disks in each group.

The corners of the smallest AABB must lie on or behind those four chains. This means that either the top left and bottom right corners lie on their respective chains, or the bottom left and top right corners lie on their respective chains, otherwise we could shrink the solution. We can try both of these options, so suppose the top left and bottom right corners lie on the chains.

For any pair of a circle segment on the top left chain and a circle segment on the bottom right chain, we can compute the smallest rectangle with a corner on both segments in constant time. However, we need to ensure that the resulting bottom left and top right corners are on or beyond their chains as well. We will maintain the horizontal projection of the endpoints of the circle segment of the top left chain on the top right chain, and the vertical projection of these endpoints on the bottom left chain. In the same way, we keep track of the projections of the endpoints of the circle segment of the bottom right chain on the top right and bottom left chains, see Figure 3.3(a).

For every circle segment c_{tl} on the top left chain, we go over every circle segment c_{br} on the bottom right chain from left to right, and treat this pair. We first determine the intersections of the projections of c_{tl} and c_{br} on the bottom left and top right chains. Observe that the projection of c_{br} moves only from left to right on the top right chain and only from right to left on the bottom left chain. If the intersection on a chain is empty, say, on the top right chain, then there are two possibilities: either there



(a) (b)

Figure 3.3 (a) The endpoints of one arc of the top left chain and one arc of the bottom right chain are projected on the top right and bottom left chains. (b) The unrestricted smallest AABB with corners on c_{tl} and on c_{br}.

is no rectangle with its top left corner on c_{tl} and its bottom right corner on c_{br} that intersects all disks of the top right group, or every rectangle with its top left corner on c_{tl} and its bottom right corner on c_{br} intersects all disks of the top right group (as in Figure 3.3(a)). Both cases are easy to handle, so assume that the intersection of the projects of c_{tl} and c_{br} is not empty on the bottom left chain (as in Figure 3.3(a)). Assume it consists of m circle segments. We compute the unrestricted optimal solution for the two circle segments, see Figure 3.3(b), and test whether it is valid in $O(m)$ time. If it is not valid, then the optimal rectangle must have a corner on the bottom left chain as well. In this case, we have three chains with a point on them, and a corner on the bottom left chain defines a rectangle immediately which we must check against the top right chain. Hence, we can find the optimal solution by walking over the $O(m)$ parts of the bottom left chain from right to left and maintaining how the top right corner of the rectangle moves with respect to the top right chain. In the process, we determine the smallest valid rectangle.

Because we go over the circle segments of the bottom right chain from left to right, the projections move in only one direction on the other two chains. Hence, the total complexity of the circle segments we encounter among all cases cannot be more than $O(n)$ for one circle segment of the top left chain.

Theorem 3.3 *Let $\mathcal{R}$ be a set of n disks in the plane. We can compute in $O(n^2)$ time a point set P containing one point from each region in $\mathcal{R}$, such that the area of the axis-aligned bounding box of P is minimised.*

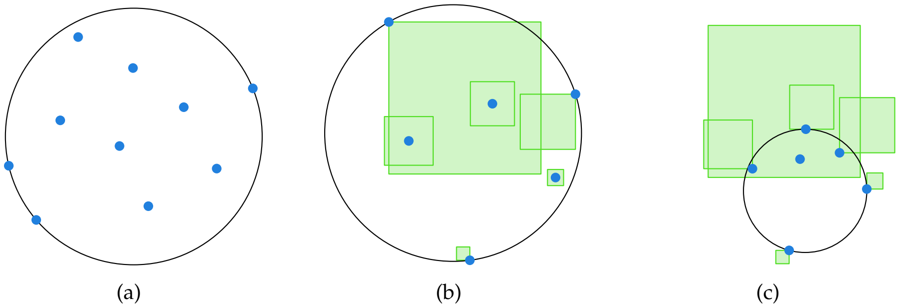

(a) (b) (c)

Figure 3.4 (a) The smallest enclosing circle of a set of points in the plane. (b) The largest SEC of a set of imprecise points. (c) The smallest SEC of the same set of imprecise points.

3.2 Smallest Enclosing Circle

We move on to another problem. Given a set of points P, the smallest enclosing circle (SEC) is the smallest circle that contains P, see Figure 3.4(a). The SEC of a set of points was already briefly discussed in Section 2.1.2.1. In this section, when we are given a set $\mathcal{R}$ of imprecise points, we want to place a point in each region such that the radius of the SEC of the resulting point set is as large or as small as possible, see Figures 3.4(b) and 3.4(c).

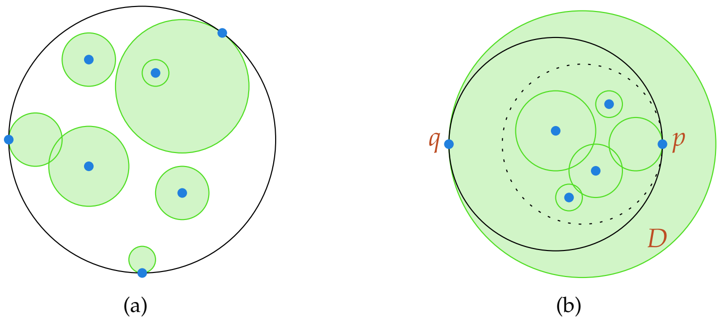
3.2.1 Largest Possible SEC

In this section, we consider the following problem:

Problem 3.3 *Given a set of disks or squares in the plane, place a point in each region such that the radius of the smallest enclosing circle of these points is maximised.*

3.2.1.1 Squares

The largest smallest enclosing circle of a set of squares (or constant size convex polygons) can be computed by first computing the smallest enclosing circle of the set of corners of all squares, using an existing SEC algorithm in $O(n)$ time (e.g. [93]). If the three points that determine this circle belong to different squares, we are done. Otherwise, there is one square of which multiple corners contribute to the smallest enclosing circle, and we know that this square has to contribute to the optimal solution. So we just try all corners of this square and compute the smallest enclosing circle of this single point and all other corners of the other squares.



(a) (b)

Figure 3.5 The LSEC (black) of a set of disks. (a) All disks are completely within the LSEC. (b) There is one disk containing all others.

If the point p we chose does not lie on the resulting circle, then we clearly chose the wrong point. If p does lie on the circle, we do not yet know whether we chose the right point, but we proceed by computing the largest smallest circle through p of the remaining squares in the same way. Since a square has four corners, and at most three points can define a circle, we inspect at most 4^3 possible solutions. We report the largest among these.

Theorem 3.4 *Let $\mathcal{R}$ be a set of n squares in the plane. We can compute in $O(n)$ time a point set P containing one point from each region in $\mathcal{R}$, such that the radius of the smallest enclosing circle of P is maximised.*

3.2.1.2 Disks

To compute the largest possible smallest enclosing circle of a set of disks, we observe that there are only two possibilities. Either the largest SEC contains all disks, or it does not, see Figure 3.5. If it does, then the largest SEC is just the smallest circle containing a set of disks, which can be computed in $O(n)$ time [95]. If it does not, this means that there must be one disk D among the input disks that contains all other disks. In this case, the largest SEC is determined by the point p in the union of all other disks closest to the boundary of D, and the point q on D furthest away from p. This case can clearly also be solved in linear time.

Theorem 3.5 *Let $\mathcal{R}$ be a set of n disks in the plane. We can compute in $O(n)$ time a point set P containing one point from each region in $\mathcal{R}$, such that the radius of the smallest enclosing circle of P is maximised.*

3.2.2 Smallest Possible SEC

In this section, we consider the following problem:

Problem 3.4 *Given a set of disks or squares in the plane, place a point in each region such that the radius of the smallest enclosing circle of these points is minimised.*

The smallest possible SEC for a set of imprecise points is the smallest circle that intersects all regions. This is also called the *intersection radius* of a set of regions. When the regions are squares (or other convex polygons), it can be computed in linear time [70].

3.2.2.1 Disks

When the points are modelled as disks, the problem of finding the smallest SEC becomes a so-called *LP-type* problem [28]. An LP-type problem is defined on a set of objects H and a function $w : 2^H \to W$, where W is some totally ordered set of possible values. The goal is to compute $w(H)$. To be LP-type, two axioms must hold, namely *monotonicity*:

$$\forall_{F \subseteq G \subseteq H} \quad : \quad w(F) \leq w(G)$$

and *locality*:

$$\forall_{F \subseteq G \subseteq H,\, h \in H} \quad : \quad w(G) = w(F) < w(F \cup \{h\}) \quad \longrightarrow \quad w(G) < w(G \cup \{h\})$$

LP-type problems can be solved in linear expected time using a generic algorithm [92].

Lemma 3.2 *Problem 3.4 is LP-type, when the regions are disks.*

Proof In our case, $H = \mathcal{R}$ is the set of disks, and w gives the radius of the smallest possible SEC, that is, of the smallest circle that intersects all disks. The monotonicity axiom clearly holds. For the locality axiom, let F and G be two collections of disks with the same radius of the smallest SEC, such that $F \subseteq G$. Let C be the smallest SEC of F: C will be tangent to two or three disks of F, and intersect all others. Any other circle of the same radius as C cannot intersect all of these two or three defining disks. Therefore, the smallest SEC of F must be the same circle C. Then, let h be disk of H outside G, such that $w(F) < w(F \cup \{h\})$. Clearly h must lie fully outside C. But now, $w(G) < w(G \cup \{h\})$ as well. ■

Theorem 3.6 *Let $\mathcal{R}$ be a set of n disks in the plane. We can compute in $O(n)$ expected time a point set P containing one point from each region in $\mathcal{R}$, such that the radius of the smallest enclosing circle of P is minimised.*

3.3 Width

Given a set of points P, the width of P is the smallest distance between any pair of parallel lines that contains P, see Figure 3.6(a). Examples of the imprecise case are given in Figures 3.6(b) and 3.6(c).

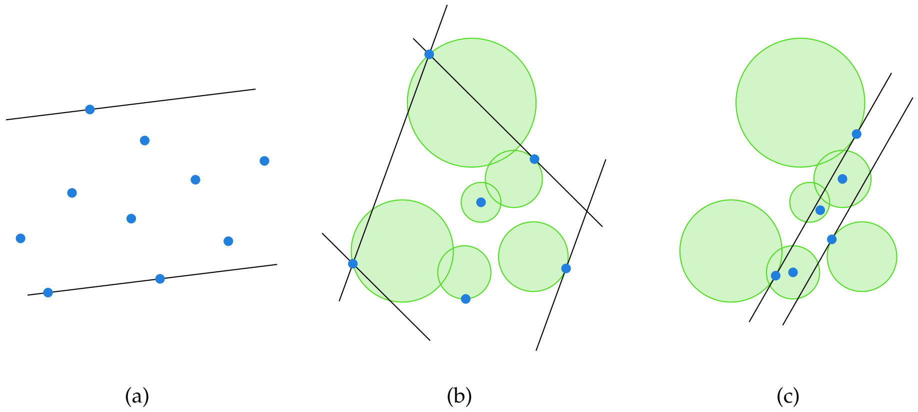

(a) (b) (c)

Figure 3.6 (a) The width of a set of points in the plane. (b) The largest possible width of a set of imprecise points. In this example, it is reached at two different locations. (c) The smallest possible width of the same set of imprecise points.

3.3.1 Largest Possible Width

In this section, we consider the following problem:

Problem 3.5 *Given a set of disks or squares in the plane, place a point in each region such that the width of these points is maximised.*

This problem seems to be hard, although we do not have any result on this problem. However, when the points are modelled as line segments, we can show the problem is in fact NP-hard. Therefore, we deviate from the structure of the chapter so far and give a result for line segments.

3.3.1.1 Line Segments

Computing the largest possible width of a set of imprecise points modelled as arbitrarily oriented line segments is NP-hard. We prove this by reduction from SAT. The construction is similar to the one used to prove NP-hardness of computing the largest possible convex hull of a set of line segments [87], but the nature of the width measure requires some new ideas.

Given a SAT instance, let k be odd and larger than the number of clauses and variables in the instance together. We base the construction on a regular $2k$-gon. We place k precise points distributed evenly on the vertices of the $2k$-gon, see Figure 3.7(a). Let the furthest distance between two of these precise points be d. The imprecise points that we will place later will all be completely within the $2k$-gon. This ensures that the largest width can at most be d, because the width of the $2k$-gon itself is d. We will

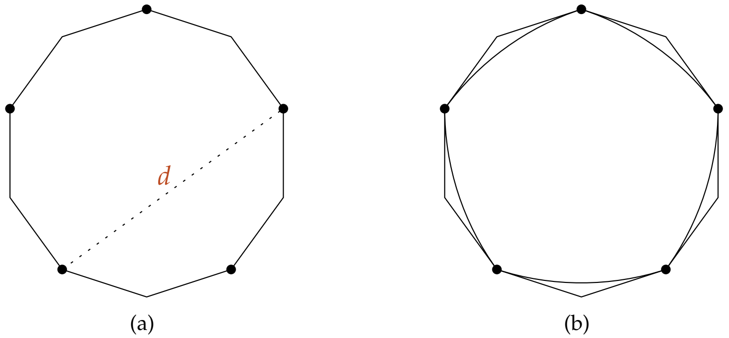


Figure 3.7 (a) A regular $2k$-gon with width d. (b) A curve of constant width, formed by k arcs.

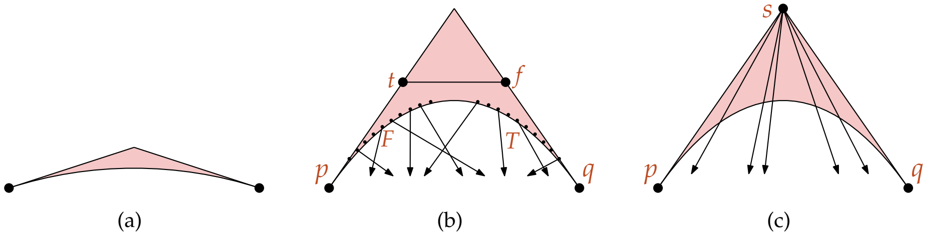


Figure 3.8 (a) A part between the $2k$-gon and the curve of constant width. (b) A variable inside a part. (c) A clause inside a part.

make a construction such that a width of d arises if and only if the SAT instance can be satisfied.

Now, for each of the k precise points we draw an arc with that point as centre between its two opposite points. These arcs together form a curve of constant width, see Figure 3.7(b). This is the smallest possible region within the $2k$-gon that contains the k precise points and has width d. As a consequence, a solution of width d is possible if and only if the imprecise points can be placed in such a way that the convex hull of the points completely contains this curve of constant width. The area between the $2k$-gon and the curve of constant width is divided into k *parts*, as shown in Figure 3.8(a), which we will use to construct variables and clauses.

Lemma 3.3 *A set of points P inside the $2k$-gon has width d if and only if the convex hull of P completely contains the curve of constant width.*

Proof If the convex hull of P completely contains the curve of constant width, its width must be at least d. On the other hand, it is still completely contained in the

$2k$-gon, which also has width d, so its width is at most d. Therefore, it is d.

If the convex hull of P does not completely contain the curve of constant width, there is some point p in the interior of the curve of constant width that is not inside the convex hull. This means that there is a line l through p that does not intersect the convex hull, and therefore does not intersect the k-gon formed by the k precise points. Because k is odd, there is a line parallel to l through one of the k precise points that does not intersect the $2k$-gon anywhere else, and since this point is part of the curve of constant width the distance between these two lines is less than d. Since the convex hull is completely between them, the width of the placed points is also less than d. ■

For each Boolean variable in the SAT instance, we will build a *variable gadget* by taking an empty arc and adding the configuration of Figure 3.8(b) inside. This configuration consists of two precise points p and q that were already added, a segment parallel to $\overline{pq}$ with endpoints t and f, and two sets of points T and F. The points of T are placed in such a way that the convex hull of $\{p, q, t\} \cup T$ contains the circular arc. Furthermore, removing any point from that set makes the convex hull intersect the arc, even when adding one or more points from F instead. Symmetrically, the points of F are placed such that the convex hull of $\{p, q, f\} \cup F$ contains the circular arc. The whole configuration is symmetric by design. By Lemma 3.3, in order to make the width larger than d, we need to use either t and all points in T, or f and all points in F. The first situation will represent the value `true` for this variable, and the second situation represents the value `false`. The points in T and F are endpoints of segments that have their other endpoint in a clause gadget.

For each clause we add a single point s at the top of an empty part, see Figure 3.8(c). We include a line segment (an imprecise point) between s and every variable in this clause. If the variable occurs normally, the other endpoint is in the T-group, and if it is negated the other endpoint is in the F-group of that variable gadget. Since we can use only one endpoint of every segment, we can place a point at s only if at least one of the variables of the clause is in the right state. Therefore, we can make a point set whose convex hull contains the curve of constant width if and only if the SAT instance is satisfiable, and together with Lemma 3.3 this implies NP-hardness of the width problem.

Theorem 3.7 *Let $\mathcal{R}$ be a set of n line segments in the plane. Computing a point set P containing one point from each region in $\mathcal{R}$, such that the width of P is maximised, is NP-hard.*

3.3.1.2 Squares or Disks

There can be many points that contribute to the width of a set of points in the plane. Figure 3.9(a) depicts an example where removing any point would result in a smaller width. Note that this is not an issue of degeneracy. In an imprecise context, this means we cannot look only at constant size subsets of the regions. Furthermore, it can happen that many points define in the optimal width, see Figure 3.9(b). This makes

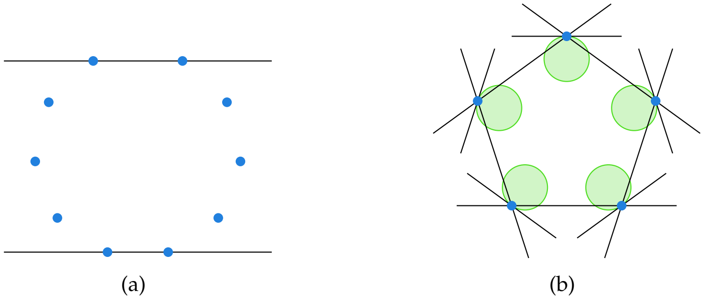

(a) (b)

Figure 3.9 (a) A set of n points, such that the width changes when any of them is removed. (b) Many triples of points define the width.

maximising the width of a set of imprecise points a hard problem. However, the hardness proof of the previous section does not work when the imprecision regions are squares or disks, thus the status of the problem remains open.

3.3.2 Smallest Possible Width

In this section, we consider the following problem:

Problem 3.6 *Given a set of disks or squares in the plane, place a point in each region such that the width of these points is minimised.*

The smallest width of a set of imprecise points can be computed efficiently. For squares (or any polygonal regions), the problem can be solved in $O(n \log n)$ time [113].

For disks, we can solve the problem by computing the *critical sequence* of the regions. Recall that a region $R \in \mathcal{R}$ is called *extreme* in a direction if, when we sort the regions by their points least far in that direction, R is (one of) the farthest. In almost every direction, there is exactly one extreme region, except for the critical directions where a line perpendicular to the direction is tangent to two regions simultaneously. If we rotate through all directions, we find the *critical sequence* of the regions, as introduced by Rappaport for a set of line segments [112]. Figure 3.10 shows an example of a critical sequence. When the regions are circles or squares, this sequence can be computed in $O(n \log n)$ time [104]. The smallest width must have a critical line on one side, so we can find it by using rotating callipers.

Theorem 3.8 *Let $\mathcal{R}$ be a set of n disks in the plane. We can compute in $O(n \log n)$ time a point set P containing one point from each region in $\mathcal{R}$, such that the width of P is minimised.*

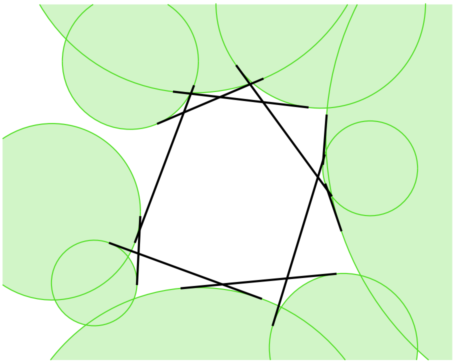


Figure 3.10 The critical sequence of a set of disks.

3.4 Closing Remarks

In this chapter, we studied three different shape fitting problems, namely the smallest enclosing axis-aligned bounding box, the smallest enclosing circle, and the narrowest strip containing a set of points. We presented algorithms for computing the upper and lower bound on their size when the points are imprecise, modelled as either squares or disks. We can compute most of these bounds efficiently, with the exception of the upper bound on the width, which seems to be a much harder problem. The exact results are summarised in Table 3.1.

The results in this chapter can also be found in [90], along with the results in Chapter 6. These results all apply only to the planar case. However, most of the results in this chapter have been extended to higher dimensions by Kruger [78], see also Kruger and Löffler [79]. Most polynomial-time solutions extend in the sense that they stay polynomial in n when studied in $\mathbb{R}^d$, though the dependency on d varies significantly between the different shapes. A notable exception is the smallest axis-aligned bounding box of a set of hyperspheres; this can be done in polynomial time in $\mathbb{R}^3$ but remains an open problem for $d > 3$.

Chapter Four

Bounds on the Convex Hull Area

Consider the same model as in the previous chapter: we are given a set of regions $\mathcal{R}$, and we want to place one point in each region to maximise or minimise some function μ. In this chapter, we let $\mu(P)$ denote the area of the convex hull of P. The convex hull is defined as the smallest convex set that contains a given input set. In our case, the input set is a finite point set in $\mathbb{R}^2$. Computing the convex hull can also be seen as a shape fitting problem, but there is one notable difference with the problems studied in the previous chapter. The shapes there were simple, in the sense that the description complexity of the output shape was always constant. The convex hull, on the other hand, can have a linear output description size.

The convex hull is one of the oldest geometric constructions studied, and can be computed in $O(n \log n)$ time, for example by sorting the points on their x-coordinate and then applying the algorithm described in Section 1.1.3, which was originally designed by Graham [57]. Apart from some direct applications, the convex hull is most often used as a subroutine in other algorithms. One example of a basic algorithm that uses the convex hull as a subroutine is an algorithm for computing the *diameter* of a set of points; more details on this can be found in Chapter 6.

As before, we consider both the square and disk model for imprecise points. However, the increased complexity of the convex hull causes algebraic issues when the regions are modelled as disks. We discuss this is some more detail in Section 4.4; for the rest of the chapter we assume that the regions are squares. For the square model, it appears that computing the lower bound of the convex hull area is much easier than computing the upper bound. We give a $O(n^2)$ algorithm that can compute the lower bound on any set of squares. For computing the upper bound, however, we have only an $O(n^7)$ algorithm, which furthermore works only when the input squares are disjoint.

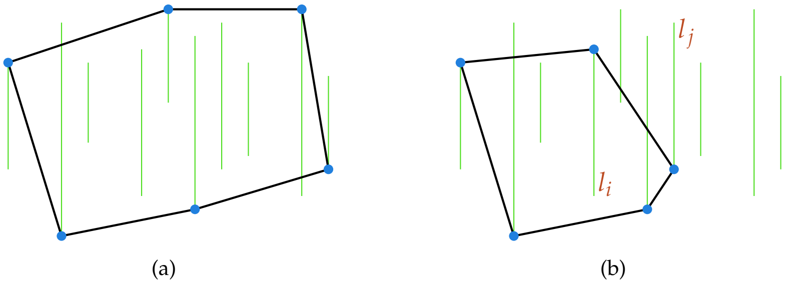


Figure 4.1 (a) The largest convex hull for a set of line segments. (b) The polygon P_{ij}.

4.1 Largest Convex Hull of Parallel Line Segments

Even though we consider the regions in $\mathcal{R}$ to be squares or disks, in this section we will first discuss the case where the regions are parallel line segments. Without loss of generality we will assume that they are vertical. This algorithm will later be reused in the other cases.

Problem 4.1 *Given a set of parallel line segments in the plane, place a point in each region such that the area of the convex hull of these points is maximised.*

Figure 4.1(a) shows an example. First we will show that we can ignore the interiors of the segments in this problem, that is, we have to consider only the endpoints.

Observation 4.1 *There is an optimal solution to Problem 4.1 such that all points are chosen at endpoints of the line segments.*

Proof Let H be an optimal solution. If p is a vertex of H, and we move it over its segment while maintaining the combinatorial structure of H, the area of the polygon changes as a linear function. The maximum of this function occurs when p is at one of the endpoints, so we can move it there. It is possible that the polygon is no longer convex or that some points of P no longer lie within the polygon, but correcting this can only increase the area of the convex hull. We repeat this for all points in P. ■

Let $\mathcal{R} = \{l_1, l_2, \ldots, l_n\}$ be a set of n vertical line segments, where l_i lies to the left of l_j if $i < j$. Let l_i^+ denote the upper endpoint of l_i, and l_i^- denote the lower endpoint of l_i. Now we need to pick one of each pair of endpoints to determine the largest area convex hull. We use a dynamic programming algorithm that runs in $O(n^3)$ time and $O(n^2)$ space. The key element of this algorithm is a polygon that is defined for each pair of line segments.

For $i \neq j$, define the polygon P_{ij} as the largest possible polygon that is the convex hull of some choice of endpoints to the left of l_i and l_j, and uses the top of l_i and the bottom of l_j, see Figure 4.1(b). In other words, it is the convex hull of a set of endpoints l_k^+ for some values $k \leq i$, and endpoints l_k^- for some values $k \leq j$, where not both l_k^+ and l_k^- can occur for the same k, such that the area of this convex hull is maximised. Note that P_{ij} is defined both for the case $i < j$ and $i > j$.

Now, we will show how to compute all P_{ij} using dynamic programming. The solution to the original problem will be either of the form P_{kn} or P_{nk} for some $0 < k < n$, and can thus be computed in linear time once all P_{ij} are known.

With some abuse of notation, we use $+$ to denote the addition of a triangle to a polygon, and we maximise over the area of the result. When $1 < i < j$, then we can write

$$P_{ij} = \max_{k<j;k\neq i} \left(P_{ik} + \triangle l_i^+ l_j^- l_k^- \right)$$

Of course we maximize over the area of the polygons. In words, we choose one of the lower points to the left of l_j, and add the new point l_j^- to the polygon P_{ik} that optimally solves everything to the left of the chosen point l_k^-. Analogously, when $1 < j < i$, we can write

$$P_{ij} = \max_{k<i;k\neq j} \left(P_{kj} + \triangle l_i^+ l_j^- l_k^+ \right)$$

When $i = 1$ or $j = 1$, we can use the same formulas to compute P_{ij}, but we need the additional option to just choose the line segment $\overline{l_i^+ l_j^-}$ with area 0, in case there are no more points to the left of the new one.

The algorithm runs in $O(n^3)$ time and requires $O(n^2)$ space. This is because we do not have to actually store the entire polygon P_{ij} for each i and j, but only the next point on the upper chain when $i > j$ or the lower chain when $i < j$, and the area of the polygon. When we scan the known polygons while determining a new one, we just add the area of a triangle to the stored area, and take the maximum of those numbers. We do not need to enforce convexity, because a non-convex solution can never be optimal.

Theorem 4.1 *Let $\mathcal{R}$ be a set of n parallel line segments in the plane. We can compute in $O(n^3)$ time a point set P containing one point from each region in $\mathcal{R}$, such that the area of the convex hull of P is maximised.*

4.2 Largest Convex Hull of Squares

The problem we discuss in this section is the following:

Problem 4.2 *Given a set of squares in the plane, place a point in each region such that the area of the convex hull of these points is maximised.*

Figure 4.2(a) show an example output. As in the case of line segments, we again observe that it suffices to consider only the corners of the squares as possible placements. The proof is analogous to that of Observation 4.1.

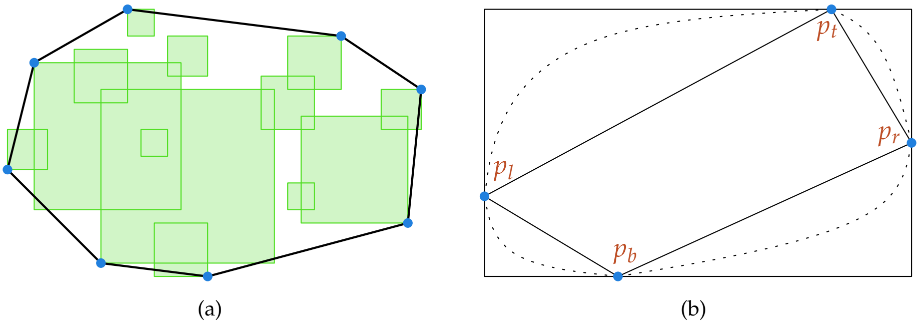


Figure 4.2 (a) The largest area convex hull for a set of squares. (b) The four extreme points narrow down the possible structure of the optimal convex hull.

Observation 4.2 *There is an optimal solution to Problem 4.2 where all points lie at a corner of their square.*

Now, let P^* be the (unknown) set of optimal placements of the points in $\mathcal{R}$. We define the four *extreme* points $p_r, p_t, p_l, p_b \in P^*$ to be the rightmost, topmost, leftmost and bottommost points of P^*. These will be present on the convex hull of P^*, and divide the hull into four chains. If we know the extreme points, the rest of the hull is restricted to stay within four triangular regions, as shown in Figure 4.2(b). We can make another observation about the nature of the points on the four chains.

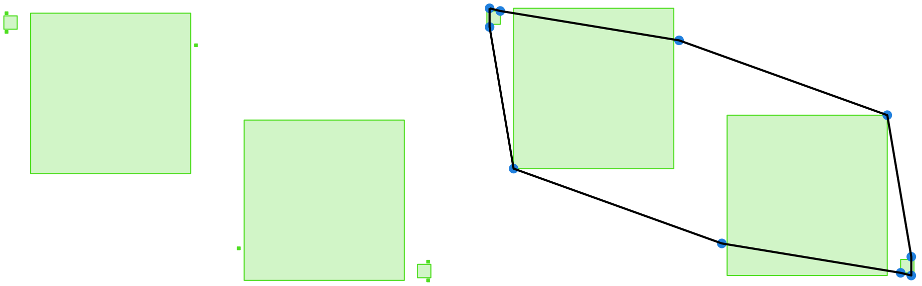
Observation 4.3 *All vertices on the top left chain are points chosen at top left corners of their squares. The same holds for the other three chains/directions.*

The general strategy will be to first find the four extreme points of the optimal solution, and then using the additional structure to place the rest of the points. Unfortunately, it is not easy to find the extreme points. For example, there is no direct relation between the extreme squares in the input and the extreme points in the optimal solution, see for example Figure 4.3.

4.2.1 Algorithm for Disjoint Squares

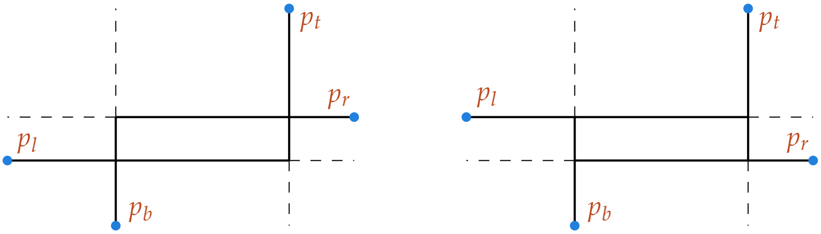
When we restrict the problem to disjoint squares, we can solve the problem in $O(n^7)$ time. The idea behind the solution is to divide the squares into groups of squares of which we know that only two of their corners are feasible for an optimal solution, and then to use the algorithm for parallel line segments (Problem 4.1) on these groups.

When the four extreme points are known, we can use this information to solve the problem in $O(n^3)$ time. However, how to find those points still remains a difficult problem, so we try all possible combinations, hence the total of $O(n^7)$.



(a) (b)

Figure 4.3 A situation where none of the four extreme points appear on an extreme input square. (a) Ten input squares. The big ones are top and bottom extreme, while the medium-sized squares are left and right extreme. (b) The optimal (largest area) solution. All the extreme points are corners of the small squares.



(a) (b)

Figure 4.4 The four extreme points can divide the plane in two different ways.

The four extreme points p_l, p_t, p_r and p_b divide the plane as shown in Figure 4.4. From p_l we draw a line to the right, from p_b one upwards, from p_r one to the left and from p_t one downwards. These four lines intersect at four intersection points. For a square to be able to have its point on the top left chain, its upper left corner needs to be in the rectangle between p_l and p_t (actually even in the upper left triangle). An analogous property holds for the other chains. If a square has the potential to be included on more than two chains, this means that it must have at least one of the four intersection points in its interior. Since the squares do not overlap, there can be at most four such squares. Of these squares we simply try every possible combination of corners, of which there are only constantly many, so we can assume from now on that every square has at most two potential corners.

Now that all squares have only two potential corners, we can represent them by line segments. We see that a segment can be of six possible kinds, as there are six ways

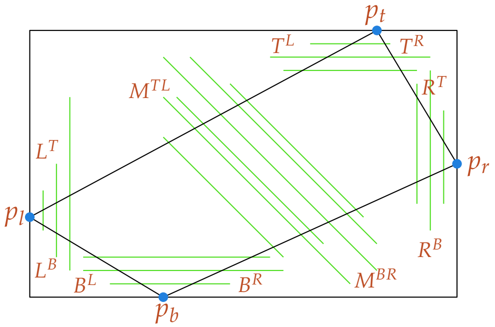


Figure 4.5 The squares can be divided into five groups of parallel line segments.

of picking two of four points. However, only five groups can appear together, since diagonal segments cannot occur in both directions without intersecting each other, see Figure 4.5. Furthermore, all line segments have to reach over the quadrilateral $\diamond p_l p_t p_r p_b$, and endpoints of line segments of the same kind must be on the same chain. Therefore, all line segments of the same kind must be close to each other, that is, their intersection intervals have to be consecutive.

We will now solve the situation of Figure 4.5 in $O(n^3)$ time. The bases L, R, T, B, and M stand for the left, right, top, bottom, and middle (diagonal) sets of line segments. The superscripts denote the endpoints of these segments.

Note that any convex hull of a choice of points in this situation must follow these sets of endpoints in the correct order. That is, it starts at the left extreme point, then goes to a number of points of L^B, then to a number of points of B^L, then to the bottom extreme point, and so on. It cannot, for example, go to a point of L^B, then to a point of B^L, and then back to a point of L^B.

The algorithm will repeatedly take two of the ten sets of endpoints, and for each combination of a point in one and a point in the other set, compute the optimal subsolution connecting those points in linear time, based on earlier results. The subsolutions are computed in the following order:

- For each pair of points in L^T and L^B, compute the optimal solution connecting them around the left side, using the algorithm for parallel line segments.
- For each pair of points in B^L and B^R, compute the optimal solution connecting them around the lower side, using the algorithm for parallel line segments.
- For each pair of points $p \in M^{TL}$ and $q \in L^B$, compute the optimal chain connecting them that does not use any other point of M^{TL}. This can be done by trying a linear number of points $r \in L^T$ as the point to connect p to, and using the known optimal chain between r and q.
- For each pair of points $p \in M^{TL}$ and $q \in B^L$, compute the optimal chain connecting them around the left side that does not use any other points of M^{TL}

and B^L. We do this by trying a linear number of points $r \in L^B$ as the point to connect q to, and combining this with the known optimal chain between p and r, computed in the previous step.
- For each pair of points $p \in M^{BR}$ and $q \in B^L$, compute the optimal chain connecting them that does not use any other point of M^{BR}. This can be done by trying a linear number of points $r \in B^R$ as the point to connect p to, and using the known optimal chain between r and q.
- For each pair of points $p \in M^{TL}$ and $q \in M^{BR}$, compute the optimal chain connecting them around the lower left side that does not use any other points of M^{TL} and M^{BR}. We can do this by trying a linear number of points $r \in B^L$ as the leftmost point of B^L that is used, and then combining the chains between p and r and between q and r that we computed in the two previous steps.
- For each pair of points $p \in M^{TL}$ and $q \in M^{BR}$, compute the optimal chain connecting them around the lower left side, which is allowed to use other points of M^{TL} and M^{BR}. We do this by using an adjusted version of the algorithm for parallel line segments. The optimal chain connecting p to q either uses another point from M^{TL} or M^{BR}, or it does not and uses the chain computed in the previous step. This means we must take the maximum of the formula given in Section 4.1, and the optimal chain of the previous step.
- In a symmetrical way, for each pair of points in M^{TL} and M^{BR}, compute the optimal chain connecting them around the upper right side that does not use any other points of M^{TL} and M^{BR}.
- Finally, check a quadratic number of pairs of a point from M^{TL} and a point from M^{BR}, and for each pair combine the chains of the previous two steps. The optimal solution is the maximum of these pairs.

All steps can be carried out in $O(n^3)$ time. The algorithm given above works when we assume that each set of endpoints is used at least once by the optimal solution. Of course, that need not be the case. But if from a certain group no point is used, then we also know that all points of the opposite group may be used, and we are left with a smaller problem that can be solved in a similar way as described above. This means we can just try solving the problem under the assumption that one or more of the groups do not appear in the optimal solution, and then pick the best solution without increasing the time bound.

Theorem 4.2 *Let $\mathcal{R}$ be a set of n disjoint squares in the plane. We can compute in $O(n^7)$ time a point set P containing one point from each region in $\mathcal{R}$, such that the area of the convex hull of P is maximised.*

4.3 Smallest Convex Hull

The problem we discuss in this section is the following:

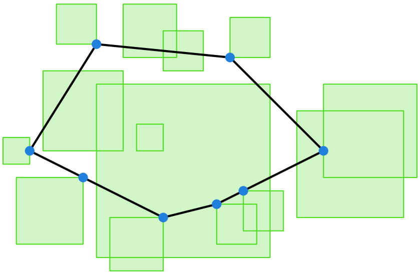


Figure 4.6 The smallest convex hull for a set of squares.

Problem 4.3 *Given a set of squares in the plane, place a point in each region such that the area of the convex hull of these points is minimised.*

Figure 4.6 shows an example. In contrast to the problem of maximising the convex hull area, we now do not have the property that all points have to be chosen at corners of their squares. An example where four vertices of the smallest hull are not at corners is shown in Figure 4.7(a). However, these four points are in fact the four extreme points of the convex hull. We can show that this is true in general.

Observation 4.4 *In the optimal solution, every vertex of the convex hull that is not an extreme point must be at the corner of its square.*

Proof Suppose some other vertex of the convex hull is not at the corner of its square. Suppose for example that this vertex lies between the leftmost and the topmost vertices. This means that moving the vertex either down or to the right will decrease the area of the convex hull. Since the vertex is not at a corner, it can move in at least one of those directions. Thus the convex hull cannot be optimal. ■

Note that it is possible that other points have to be placed in the interior of their squares, for example when one very big square contains all others. The lemma describes only those points that become a vertex of the solution.

Now, again different from the previous section, the squares that provide the extreme points are known. They are the square with the rightmost left edge, topmost bottom edge, etc. These four squares will be called the *axis-extreme squares*, and we call them S_l, S_r, S_t and S_b. We define the four chains that connect the corners of the squares. The *top left chain*, for example, will be the chain connecting the bottom right corner of S_l to the bottom right corner of S_t, via other bottom right corners of squares, such that the region to the lower right of the chain is convex and contains a point of every square. In the same way we can define the *top right chain*, the *bottom right chain*, and the *bottom left chain*. An example is shown in Figure 4.7(b).

For every location of the point p_l, there is a *tangent point* $a_{lt}(p_l)$ on the top left chain such that the line through p_l and $a_{lt}(p_l)$ does not go through the region to the lower right of the top left chain. When there are more than one such points we choose the

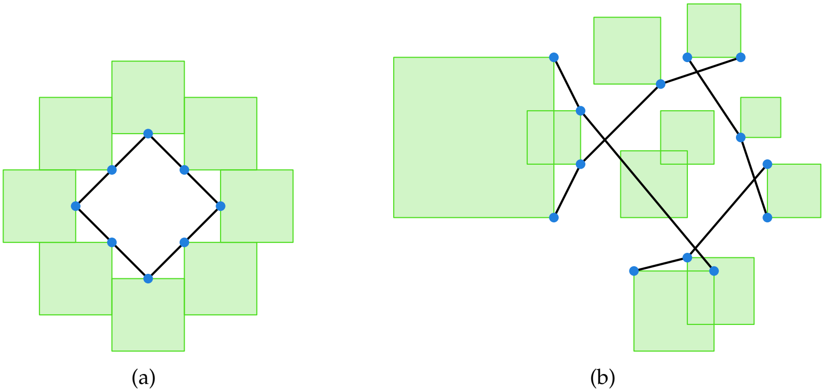


Figure 4.7 (a) Up to four vertices of the smallest convex hull may not be corners of any of the squares. (b) The top left, bottom left, top right, and bottom right chains.

one that lies most to the upper right. Similarly, we define $a_{lb}(p_l)$ as the tangent point on the bottom left chain. For every point p_t we define tangent points $a_{tl}(p_t)$ and $a_{tr}(p_t)$ on the top left and top right chains, for every p_r we define two tangent points $a_{rt}(p_r)$ and $a_{rb}(p_r)$ on the top right and bottom right chains, and finally we define for every point p_b two tangent points $a_{bl}(p_b)$ and $a_{br}(p_b)$ on the bottom left and bottom right chains. All those tangent points are vertices of the chains. Note that they may also be corners of the extreme squares.

Lemma 4.1 *If the points p_l, p_t, p_r and p_b are known, in the optimal solution the point p_l is connected to p_t by a straight line segment if this segment does not intersect the top left chain, and otherwise via the piece of the top left chain between $a_{lt}(p_l)$ and $a_{tl}(p_t)$. Similarly p_t is connected to p_r by a straight line segment or via the piece of the top right chain between $a_{tr}(p_t)$ and $a_{rt}(p_r)$, p_r is connected to p_b by a straight line segment or via the piece of the bottom right chain between $a_{rb}(p_r)$ and $a_{br}(p_b)$, and p_b is connected to p_l by a straight line segment or via the piece of the bottom left chain between $a_{bl}(p_b)$ and $a_{lb}(p_l)$.*

Proof The optimal solution H contains the convex hull of p_l, p_t, p_t and p_b. If there is a vertex q of the top left chain that is to the top left of the segment $\overline{p_l p_t}$, but is not contained in the optimal solution, then the solution is invalid. This is because q is a bottom right corner of its square, so the whole square is disjoint from H. Similar reasoning applies to the other three chains. ■

In the degenerate case where the squares S_l, S_t, S_r and S_b are just points, this implies that the four chains together form the optimal solution.

Finally, we observe that we can find out if a zero area solution exists in linear time, since a solution of zero area corresponds to a stabber of a set of squares, which can be computed in $O(n)$ time if it exists [38]. Therefore, we assume for the remainder of this section that the optimal solution has a positive area.

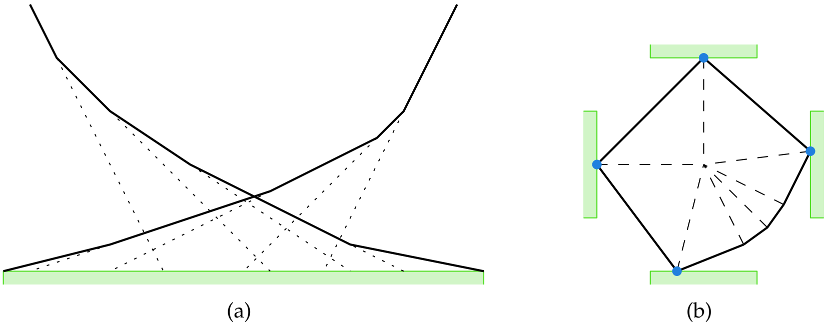

(a) (b)

Figure 4.8 (a) The lower left and the lower right chains divide the upper edge of S_b into a linear number of intervals. (b) The area of a solution can be decomposed into triangles that depend on at most two variables.

4.3.1 Naive Algorithm

Each of the four extreme points can move over an edge of its square. We divide this edge into a linear number of intervals. Within each interval, if the point would be chosen there, the vertices on the chains the point would be connected to are the same. This means the endpoints of the intervals are exactly the points that lie on a line through two consecutive vertices of one of the chains. The resulting intervals are shown in Figure 4.8(a).

Assume we know for each of the four extreme points the interval on which they must be in the optimal solution. We then know the tangent points on the chains they must be connected to if we do not look at the other extreme points. Since we know this for each extreme point, we can see whether they will be connected to each other or to a chain. For example, the left point p_l has a point $a_{lt}(p_l)$ on the top left chain it could be connected to, and the upper point p_t has a point $a_{tl}(p_t)$ on the top left chain it could be connected to. If $a_{lt}(p_l)$ lies to the lower left of $a_{tl}(p_t)$, then p_l and p_t will be connected to their tangent points, and not to each other. If $a_{lt}(p_l)$ lies to the upper right of $a_{tl}(p_t)$, then p_l and p_t will be connected by a straight line segment. If $a_{lt}(p_l)$ is equal to $a_{tl}(p_t)$, then we do not know yet.

We can now write the area of the convex hull as a polynomial in four variables that specify the exact locations of the extreme points within their interval. This polynomial will have degree at most 2, because we can decompose the area into triangles that depend on at most two of the variables, as shown in Figure 4.8(b). We can find the minimum of this polynomial within the bounds given by the intervals on which the points can move, in constant time. In the case where we do not know whether two points will be connected by a straight line segment or via a point on a chain, we simply try both and add an extra restriction to the variables in the one case.

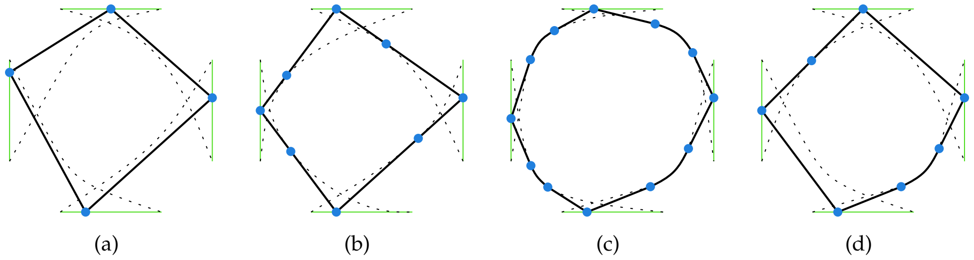

(a) (b) (c) (d)

Figure 4.9 The points can be connected either (a) directly, (b) just touching a point on the chain, (c) via multiple points of the chain. (d) Several cases can appear together.

We can now easily solve the problem in $O(n^4)$ time. In the optimal solution, each of the four extreme points needs to be on one of the intervals of its segment. This means a total of $O(n^4)$ possible combinations of intervals, and for each combination the solution requires solving a polynomial that does not depend on n. However, many of the combinations of intervals seem to be redundant, because a solution using them can clearly be seen not to be optimal. Indeed we can show that we do not need to spend $O(n^4)$ time to solve this problem, and improve it to $O(n^2)$.

4.3.2 Improved Algorithm

We observe that each connection between two of the extreme points must be one of three types. We call the connection of *type 0* if the points are connected by a single line segment that does not touch the respective chain in the optimal solution. We call the connection of *type 1* if the points are connected by a single line segment that touches the respective chain. We call the connection of *type 2* if they are not connected by a single line segment. In Figure 4.9 examples are shown of only type 0 connections, only type 1 connections, only type 2 connections, and an example with two connections of type 0, one of type 1, and one of type 2.

There are only a constant number of possible combinations of connections between the four extreme points. If we can compute the optimal solution for each type (that is, the best solutions among all solutions that have that type) in less than $O(n^4)$ time in total, then we can just pick the best one and we have a faster algorithm. To do this, we define three patterns, each of which allows us to solve the problem faster for a different reason.

- Each connection of type 1 reduces the number of possible combinations of intervals by a linear factor.

- Every pair of connections of type 2 divides the problem into two independent subproblems.
- Every pair of adjacent connections of type 0 removes the need to divide one of the extreme squares into intervals, reducing the number of possible combinations of intervals by a linear factor.

The first pattern reduces the time by a linear factor, because there is only a linear number of pairs of intervals for the two extreme points in question for which the point to which they can be connected on the chain is the same, and this is required for a type 1 connection.

The second pattern also reduces the time by a linear factor. Since we know by assumption that the two connections are of type 2, the optimal subproblems together have to be the optimal solution for the complete problem. The two connections could be adjacent, giving one subproblem of linear complexity and one of potentially cubic complexity, or they could be opposite to each other, giving two problems of potentially quadratic complexity.

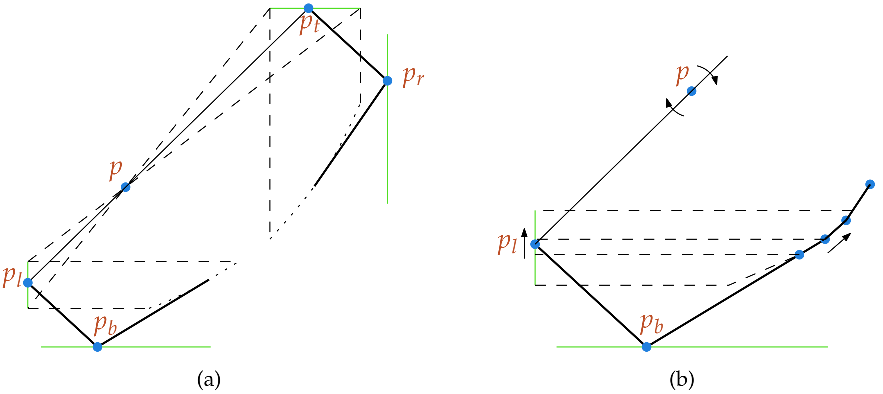
Finally, the third pattern also reduces the time by a linear factor: if one of the extreme points has only connections of type 0, there is no need to divide its edge into intervals, since the structure of the convex hull, and therefore the polynomial describing the area, will be the same.

These patterns can occur together, potentially saving multiple linear factors. As an example, take type 0-1-2-2. This type contains two type 2 connections, and also a type 1 connection. The type 2 connections divide the problem into two independent subproblems. The smallest subproblem can be solved in linear time, by just choosing the best out of a linear number of intervals. For each interval we know their tangent points on the chains, so it can be solved in constant time. The larger subproblem contains a type 1 connection. This means there is only a linear number of combinations of intervals such that the connection is really of type 1. We can find them and store them easily in quadratic time. Now we look at a quadratic number of groups of three intervals; one combination of two that we just stored, and one from the remaining extreme point (the one between the type 0 and type 2 connections). For each three intervals, we can solve the problem in constant time since all tangent points are known. We can also check in constant time whether all connections are of the correct types (in particular the type 0 connection), by looking at the tangent points. We have now solved both subproblems in $O(n^2)$ time. We finally need to check whether the subsolutions together yield a convex shape. If they do not, then the type was not 0-1-2-2. If they do, then it is a potential optimal solution.

There are 3^4 possible types, but after removing symmetries only 21 remain, see Table 4.1. Note that in all 21 types at least one of these three patterns has to occur, and thus every type can be solved in $O(n^3)$ time. Furthermore, in all but one case (and its symmetric variants), actually two of these patterns occur together, and they can be solved in $O(n^2)$ time. All these types can be solved in a similar way to the one

0-0-0-0	0-0-0-1	0-0-0-2	0-0-1-1	0-0-1-2	0-1-0-1	0-1-0-2
1-1-1-1	1-1-1-2	1-1-1-0	1-1-2-2	1-1-2-0	1-2-1-2	1-2-1-0
2-2-2-2	2-2-2-0	2-2-2-1	2-2-0-0	2-2-0-1	2-0-2-0	2-0-2-1

Table 4.1 The 21 possible types.



(a) (b)

Figure 4.10 (a) The special case 0-1-0-2. (b) Expanded view of the bottom left part.

described above. The one exception is type 0-1-0-2, of which an example is shown in Figure 4.9(d).

This type has only one of the three patterns. However, we will analyse it separately and show that we can in fact solve this type in linear time.

Lemma 4.2 *The pattern with a type 0 connection, a type 1 connection, a type 0 connection and a type 2 connection, in that order, can be solved in* $O(n)$ *time.*

Proof We assume the types of connections to be as in Figure 4.9(d); other cases are symmetric. Since the top right and bottom left connections do not touch the chains, we do not need to look at these two chains any more. There will be one point p on the top left chain that is the tangent point of the top left connection. Now the top left connection is a single line segment from the leftmost extreme point to the topmost extreme point, and is still allowed to rotate around p within some interval such that it does not intersect the top left chain, see Figure 4.10(a).

For a fixed position of the top left connection, and therefore the points p_l and p_t, the optimal solution is easy to see. Find the two consecutive points on the bottom right chain such that the leftmost extreme point has its y-coordinate between the y-coordinates of those two points. The bottommost extreme point will be on the line through these two points, because moving it in either direction from that position would increase the area. In the same way, the rightmost point must be on the line through the two points that have their x-coordinates closest to that of the topmost extreme point, as in Figure 4.10(a).

Now if we start with p as the bottom leftmost point on the top left chain, and rotate the line connecting the leftmost extreme point to the topmost extreme point around it in clockwise direction, the bottommost two points of the bottom right chain that determine the position of the bottommost extreme point will move only upwards, while the two points that determine the position of the rightmost extreme point will also move only upwards, see Figure 4.10(b). When the line rotates far enough, p will change to the next point of the top left chain, but still the points on the bottom right chain move only towards the upper right. This means that after a linear number of steps we have tried all possibilities and found the optimal solution.

Because we assumed the type is 0-1-0-2, we still have to make sure that the solutions we check are indeed of this type. This means that we have to check that the lower left and upper right connections do not intersect their respective chains. We can easily check this in linear time. Also, the points on the lower right chain have to be in the correct order, otherwise the resulting polygon is not convex and may even intersect itself. We can check this in linear time as well. ■

We conclude that every situation can be solved in $O(n^2)$ time, and therefore the whole problem can be solved in $O(n^2)$ time.

Theorem 4.3 *Let $\mathcal{R}$ be a set of n squares in the plane. We can compute in $O(n^2)$ time a point set P containing one point from each region in $\mathcal{R}$, such that the area of the convex hull of P is minimised.*

4.4 Disks

Perhaps the most natural way of modeling imprecision is by allowing every point to be inside a disk. Figure 4.11 shows an example of the resulting problem for computing bounds on the convex hull:

Problem 4.4 *Given a set of disks in the plane, place a point in each region such that the area of the convex hull of these points is minimised or maximised.*

Two difficulties are introduced by using circular regions. The first difficulty is that the combinatorial complexity of the problem increases. In the square model we can use the notion of extreme points in some directions. With disks this is not possible, since there are no special directions.

The second difficulty is of an algebraic kind. Even when we know which disks have to be chosen to obtain the largest/smallest area/perimeter, it is not easy to find out where exactly in the disks the points should be. For example, in the case of the smallest perimeter, even in a simple situation with only three disks, the coordinates

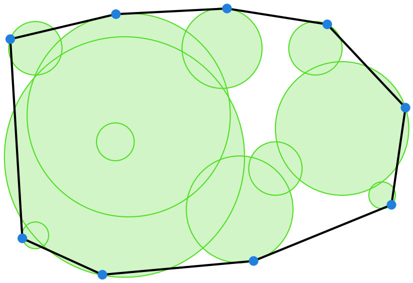


Figure 4.11 The largest area convex hull for a set of circles.

of the optimal points within the disks will generally be the roots of polynomials of degree six. These roots cannot be computed exactly, only approximated. Such algebraic difficulties are present in many geometric problems [9], and will appear more often in this thesis. Often, these issues are ignored in computational geometry, and a theoretical model of computation that can handle them is assumed, since in practice there are good numerical methods to solve such algebraic problems as a last step. However, this does mean that the result is not exact anymore, and we could say that an approximation is the best we can get in any case, and therefore a good polynomial-time approximation is a good solution.

One case Problem 4.4 has been studied before. When given a set of unit size disks, Boissonnat and Lazard [17] show that the smallest perimeter of a set of points chosed from the disks can be approximated in polynomial time. The question of whether it can be solved exactly in polynomial time is left open, and has to our knowledge not yet been answered. The same problem for smallest area is also stated as an open problem.

4.5 Closing Remarks

In this chapter, we presented algorithms for computing the upper and lower bound on the area of the convex hull of a set of imprecise points, modelled as squares. The algorithm for computing the lower bound runs in $O(n^2)$ time and works for any set of squares, while the algorithm for computing the upper bound requires the squares to be disjoint and takes $O(n^7)$ time.

These results are a selection from the results in the author's Master thesis [83], and later appeared in [89]. In those places, additional algorithms are given, in particular an algorithm that computes the upper bound for a set of overlapping unit squares in $O(n^5)$ time, and one for disjoint unit squares that runs in only $O(n^3)$ time. The setting where the squares are arbitrary remains an open problem. However, the problem of maximising the convex hull of a set of *line segments* is proven to be NP-hard, using a construction very similar to that shown in Section 3.3.1.1 of this thesis. Finally, all of these problems are also studied in the setting where the perimeter of the convex hull is measured, rather than the area, with similar results, though the exact exponents of the polynomial time bounds differ.

All these results apply only to squares; as mentioned in the previous section there is no hope for solving the problem exactly for disks. As mentioned, approximation algorithms for computing the smallest convex hull of a set of disks already exist. Approximation algorithms for computing the *largest* convex hull are the topic of the next chapter.

Chapter Five

Approximate Largest Convex Hull

In the previous chapter, we studied the computational complexity of computing the largest and smallest convex hull of a set of imprecise points modelled as a set of squares. While the result for computing the lower bound was reasonably efficient, computing the largest convex hull seems to be much harder. Firstly, the time bound of the algorithm of $O(n^7)$ is too high to be useful in practice, even though it is polynomial. Secondly, the algorithm works only when the input regions are disjoint, which may be too restrictive in practical situations. Also, for the disk model we were unable to provide any exact algorithms at all, due to algebraic difficulties.

At the same time, we are dealing with imprecise points, and it is reasonable to assume that the squares or disks in the input are themselves approximations of the "real" imprecision regions. So perhaps we do not need exact solutions to this problem, but approximate answers may be sufficient. Here we show that we can use a generic approach for approximately computing the largest convex hull, that improves the running time and restrictions for the square model and is the first algorithm in the disk model.

We present linear-time approximation schemes for computing the largest area convex hull of a set of squares or disks. For squares, the algorithm runs in $O(n + \eta^{14})$time, where n is the input size and $\frac{1}{\eta} = \varepsilon$ is the required precision of the answer. For disks, the algorithm runs in $O(n) + 2^{O(\eta^2 \log \eta)}$ time. The dependence on n is linear, provided that the ceiling operation can be performed in constant time,[1] which makes the results suitable for large data sets. On the other hand, the dependence on η is rather high, which makes the results less suitable for achieving good precision.

---

[1] Although the Real RAM model does not allow a constant time floor operation, it is fairly common in computational geometry to add this operation to the model, since in practice it can be executed very fast. However, care must be taken: when the values involved are very large such an extra operation can be too powerful. See for example [117]. Other results in this thesis that depend on the availability of the floor function are in Section 9.1.1 and in Section 10.1.2.2.

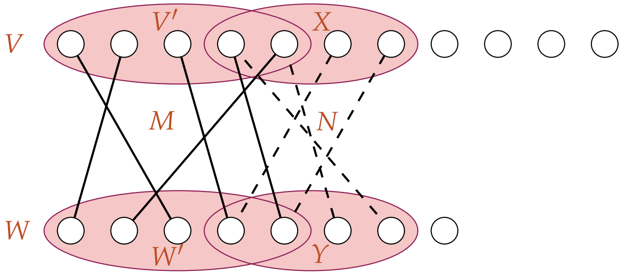


Figure 5.1 There are vertices in $X - V'$, and from these vertices there is an augmenting path that ends in either $V' - X$ or $Y - W'$.

5.1 Preliminaries

Before describing the results, we will provide a few miscellaneous results on matchings and geometric approximations that we need later.

5.1.1 Perfect Matchings in Bipartite Graphs

Let G be a bipartite graph with two sets of vertices V and W, and a set of edges $E \subset V \times W$. A *matching* $M \subset E$ between two subsets $A \subset V$ and $B \subset W$ is a subset of E such that each edge in M goes from an element of A to an element of B, and no element of A or B is used by more than one edge in M. Such a matching is called *perfect* if it consists of exactly $\min(|A|, |B|)$ edges.

Lemma 5.1 *Let G be a bipartite graph with vertex sets V and W and edge set E. Let $M \subset E$ be a maximum-cardinality matching of G, and let $V' \subset V$ and $W' \subset W$ be the two vertex sets that are used by M. For every subset $Y \subset W$, if there is a perfect matching between V and Y then there is also a perfect matching between V' and Y.*

Proof Suppose the lemma is false. Let $Y \subset W$ be a subset of W such that there exists a perfect matching between V and Y, but no perfect matching between V' and Y. Let $N \subset E$ be the matching among all perfect matchings between V and Y that uses the largest number of vertices of V'. Let $X \subset V$ be the set of vertices used by N, apart from Y. Then $X \not\subset V'$, so there is a vertex $x \in X$ with $x \notin V'$; see Figure 5.1.

Now start an augmenting path from x that uses only edges of $M \cup N$. This path takes alternating edges from N and from M, since no two from the same set can use the same vertex. Therefore, this path ends either in a vertex $v \in V' - X$ or in a vertex $w \in Y - W'$. In the first case, we have a perfect matching between $X - \{x\} \cup \{v\}$ and Y, which is in contradiction with the choice of N. In the second case, we have a perfect matching between $V' \cup \{x\}$ and $W' \cup \{w\}$, which contradicts the maximality of M. ■

5.1.2 Constant Factor Approximation of the Diameter

Let P be a set of points in the plane. The *diameter* of P is the largest distance between any two points in P. Here we briefly show that the diameter can be approximated in linear time.

Lemma 5.2 *Let P be a set of n points, and let d be its diameter. We can find two points of P whose distance is at least $\frac{d}{\sqrt{2}}$ in $O(n)$ time.*

Proof Compute the axis-parallel bounding box B of P with dimensions $w \times h$. If $w > h$, take the leftmost and rightmost points, otherwise take the topmost and bottommost points. In the worst case, B is a square and the distance between those points is the side length of the square, while the real diameter is the diagonal, which is a factor $\sqrt{2}$ larger. ■

5.2 Approximating the Largest Convex Hull

We study the problem of finding the largest possible convex hull for a set of imprecise points. We measure the size of the convex hull by its area. That is, we want to solve Problem 4.2 of the previous chapter. As before, we are given a set $\mathcal{R}$ of subsets of $\mathbb{R}^2$. We denote a *solution*, that is, a convex polygon with each vertex in a different region of $\mathcal{R}$, by S, and its area by $A(S)$. We denote the *optimal* solution by S^*.

Unlike in the previous chapter, though, here we do not ask for an exact solution to the problem, but are satisfied with an approximate solution. In particular, we look for $(1-\varepsilon)$-approximations, that is, algorithms that return for any value ε a solution with a value that is at least $(1-\varepsilon)$ times the value of the optimal solution. We also denote $\eta = \varepsilon^{-1}$.

To solve the problem we use the *core-set* paradigm, introduced by Agarwal and Har-Peled [2]. In this framework, a point set P is given, and the problem is to maximise some measure $\mu(P)$. To do this, one constructs a *core-set* $P' \subset P$, such that $\mu(P') > (1-\varepsilon)\mu(P)$. The size of the core-set must depend only on ε, and not on n (or sublinearly on n, depending on the application). Now the total running time of the algorithm is the time it takes to construct P', and the time it takes to compute $\mu(P')$, where the second step does not depend on n. If the first part can be done in linear time, one obtains a *linear-time approximation scheme* (LTAS) [23].

In our case, we want to find a core-set $\mathcal{R}' \subset \mathcal{R}$ with respect to the measure μ, where μ measures the area of the largest possible convex hull: a set of regions such that the optimal solution S'^* for $\mathcal{R}'$ has area $A(S'^*) \geq (1-\varepsilon)A(S^*)$. If necessary, this can be translated back to a solution for $\mathcal{R}$ by just taking a random point inside each region that is not part of $\mathcal{R}'$.

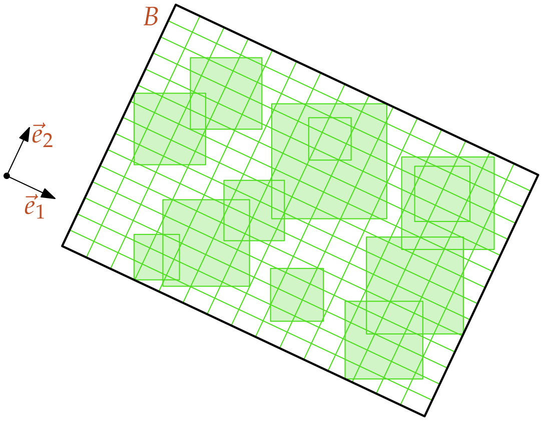


Figure 5.2 A set of squares divided according to their cells.

5.3 Squares

In this section, we consider again Problem 4.2, which we repeat here as Problem 5.1 for convenience.

Problem 5.1 *Given a set of squares in the plane, place a point in each region such that the area of the convex hull of these points is maximised.*

The status of the unrestricted version of this problem is still open. In the optimal solution, every point has to be chosen on a corner of its square. Therefore we can solve the problem in $O(4^n n \log n)$ time by computing the convex hull of every possible set of corners. However, as we showed in the previous chapter, the problem can be solved more efficiently when the squares are disjoint. By Theorem 4.2, we can solve the problem in that case exactly in $O(n^7)$ time.

5.3.1 Core-Set Construction

Let s_{max} be the largest square in $\mathcal{R}$, and s_{max2} the second largest square. We will allways make s_{max} and s_{max2} part of the core-set $\mathcal{R}'$. Now, using Lemma 5.2 we compute two points p and q that approximate the diameter d of the vertices of $\mathcal{R} - \{s_{max}, s_{max2}\}$ by a factor 2. Let $\vec{e}_1$ be unit vector in the direction from p to q, and $\vec{e}_2$ the unit vector perpendicular to this. Let B be the smallest bounding box of $\mathcal{R} - \{s_{max}, s_{max2}\}$ in the $(\vec{e}_1, \vec{e}_2)$ coordinate system, and let $w \times h$ be its dimensions. Assume that $w > h$, otherwise we swap $\vec{e}_1$ and $\vec{e}_2$.

Divide B into $2^{14}\eta$ by $2^{14}\eta$ grid cells; see Figure 5.2. The cells will be $\delta_1 = 2^{-14}\varepsilon w$ long in the $\vec{e}_1$ direction, and $\delta_2 = 2^{-14}\varepsilon h$ long in the $\vec{e}_2$ direction. Consider the bipartite

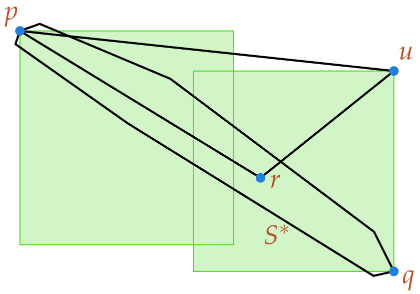


Figure 5.3 Triangle $\triangle pru$ has a larger area than S^*.

graph where one set of nodes corresponds to the set of squares $\mathcal{R} - \{s_{max}, s_{max2}\}$, and the other set of nodes corresponds to the cells of the grid. There is an edge between square s and cell c if one of the corners of s is in c. Let M be a maximum-cardinality matching of this graph. Now, let $\mathcal{R}'$ be the set of all squares that occur in M, together with s_{max} and s_{max2}. Then $\mathcal{R}'$ is a core-set for $\mathcal{R}$.

5.3.2 Proof that $\mathcal{R}'$ is a Core-Set

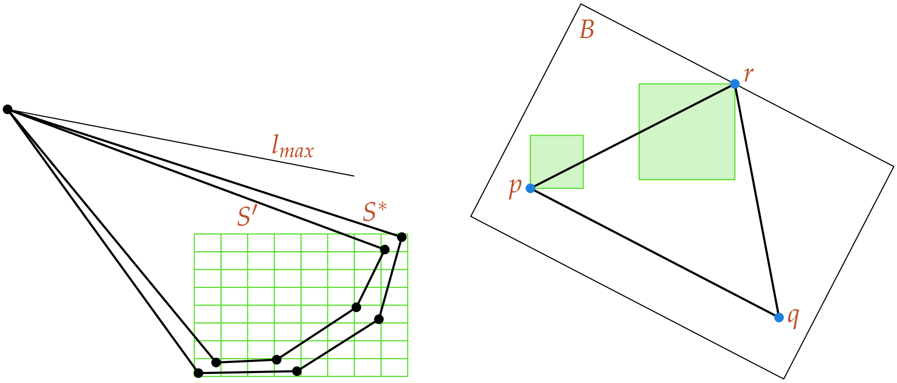
Let S^* be an optimal solution for $\mathcal{R}$, the original input, and let S'^* be an optimal solution for $\mathcal{R}'$, the core-set. First we show that the area of S^* is bounded from below by a constant factor of the area of B. Then we prove that the difference in area between S^* and S'^* is only a fraction of the area of B, depending on ε.

Lemma 5.3 *If $n \geq 3$, then the width of S^* is at least $\frac{1}{8}$ times the side length of s_{max2}.*

Proof Let b be the side length of s_{max2}, and let ω^* be the width of S^*. Assume the lemma is not true, so $\omega^* < \frac{1}{8}b$. Let p and q be the vertices of S^* that define the diameter d^* of S^*. Then we know that the area of S^* is $A^* \leq d^*\omega^* < \frac{1}{8}d^*b$. Suppose either p or q is not a corner of one of the two largest squares. Then one of the two largest squares (the one that does not provide p or q) has a corner u that is at least $\frac{1}{2}b$ away from the line extending $\overline{pq}$, and there exists a solution of area $\frac{1}{4}d^*b > \frac{1}{8}d^*b$, so in this case S^* would not be optimal, a contradiction. Now suppose that both p and q are corners of the largest two squares; see Figure 5.3. Let $r \neq p, q$ be an arbitrary vertex of S^*. Now r is at least $\frac{1}{2}d^*$ away from either p or q, say p without loss of generality. Now the square that has q as a corner has another corner u that is at least $\frac{1}{2}b$ away from the line extending $\overline{pr}$, and there exists a solution of area $\frac{1}{8}d^*b$, so in this case S^* would not be optimal either. Therefore the assumption is false, and the lemma is true. ■

This lemma implies that the area of S^* is at least 2^{-8} times the area of s_{max2}.

Lemma 5.4 *The area of S^* is at least 2^{-12} times the area of B.*



(a) (b)

Figure 5.4 (a) In one strip, the horizontal difference between the points in S^* and S' is at most δ. (b) If all squares are small, the triangle $\triangle pqr$ has a large area.

Proof Let b be the side length of s_{max2}. Recall that w and h denote the dimensions of B, and that $w > h$. If $b \geq \frac{1}{4}w$, then this square has area at least $2^{-4}wh$. The optimal solution has area at least 2^{-8} times the second largest square, so at least $2^{-12}wh$.

Next, assume that $b < \frac{1}{4}w$. If p and q, approximating the diameter of the vertices of $\mathcal{R} - \{s_{max}, s_{max2}\}$, were corners of the same square, then the width of this square would be at least $\frac{1}{2}d \geq \frac{1}{2}w$, which is larger than the diameter of s_{max2}. So p and q are corners of different squares. Let r be the point in P furthest from the line extending $\overline{pq}$; see Figure 5.4(b). If r is a corner of yet another square, then the solution $\triangle pqr$ has an area of at least $\frac{1}{8}wh$, and so the optimal solution also has at least that area.

If r is a corner of the same square as either p or q, say p, then this means that the width of this square is larger than $\frac{1}{4}\sqrt{2}h$, so $b > \frac{1}{4}h$. The optimal solution S^* uses some corner p'_l of the same square as p_l, the leftmost point in the $\vec{e}_1$ direction, and some corner p'_r of the same square as p_r, the rightmost point in the $\vec{e}_1$ direction and we know that the distance between p'_l and p'_r is at least $w - 3b > \frac{1}{4}w$. We also know that S^* has a width of at least $\frac{1}{8}b > \frac{1}{32}h$, so the area of S^* is at least $2^{-6}wh$. ■

Lemma 5.5 *There exists a solution S' for $\mathcal{R}'$ such that the difference between the areas of S' and S^* is at most $2^{-12}\varepsilon$ times the area of B.*

Proof Let Y be the set of grid cells used by the optimal solution S^*. There exists a perfect matching between $\mathcal{R}$ and Y, since, otherwise S^* would not be possible. By Lemma 5.1, we know that there is also a perfect matching between $\mathcal{R}'$ and Y. Let S' be the convex hull of the point set that realises this matching. Then for each vertex of S^* there is a point of S' in the same grid cell. Going from S^* to S', all vertices can move a distance of δ_1 in the $\vec{e}_1$ direction, and δ_2 in the $\vec{e}_2$ direction; see Figure 5.4(a).

In the worst case, the transformed solution has a complete band around it, that is, it is the Minkowski sum of the old solution and a $2\delta_1$ by $2\delta_2$ rectangle centred at $(0, 0)$. The area of such a band is smaller than $2\delta_1 h + 2\delta_2 w = 2^{-12}\varepsilon wh$. ■

Let S'^* be the optimal solution for $\mathcal{R}'$. Then we have:

$$A(S'^*) \geq A(S') \geq A(S^*) - 2^{-12}\varepsilon wh \geq A(S^*) - 2^{-12}\varepsilon 2^{12} A(S^*) = (1-\varepsilon)A(S^*)$$

5.3.3 Running Time Analysis

The computation of B takes linear time, by Lemma 5.2. Here we need to perform the ceiling operation to allocate the corners of the squares to the right cells of the grid; without the ability to execute this operation in constant time we need to spend $O(n \log n)$ time here.

To compute a maximum-cardinality matching, we can use the algorithm by Hopcroft and Karp [68], which runs in $O(\sqrt{|V|}|E|)$ time. In our case, we have $n-1$ nodes on the left side and $2^{28}\eta^2$ nodes on the right side, and every left node has degree 4. When there are more than four left nodes that are connected to the same four right nodes, we will never use more than four of them, so we can reduce the number of left nodes to at most $4 \cdot 2^{112}\eta^8$ by using radix sort. The number of edges is four times the number of left nodes. Now we can compute a maximum-cardinality matching in $O(\eta^{12})$ time. In total this takes $O(n + \eta^{12})$ time.

Theorem 5.1 *We can compute a core-set of size $O(\eta^2)$ for Problem 5.1 in $O(n + \eta^{12})$ time.*

For arbitrary squares, we can solve the problem exactly in $O(4^n n \log n)$ time; therefore we can approximate it in $O(n) + 4^{O(\eta^2)}$ time. On the other hand, we can solve the problem for disjoint squares exactly in $O(n^7)$ time so in this setting we get a strong linear-time approximation scheme that runs in $O(n + \eta^{14})$ time.

5.4 Disks

In this section, we consider the following problem:

Problem 5.2 *Given a set of disks in the plane, place a point in each region such that the area of the convex hull of these points is maximised.*

Our exact solution to the convex hull problem for square regions makes use of the four extreme points in the cardinal directions, which makes it impossible to extend to circular regions. And, as mentioned before, even if we know which disks have points

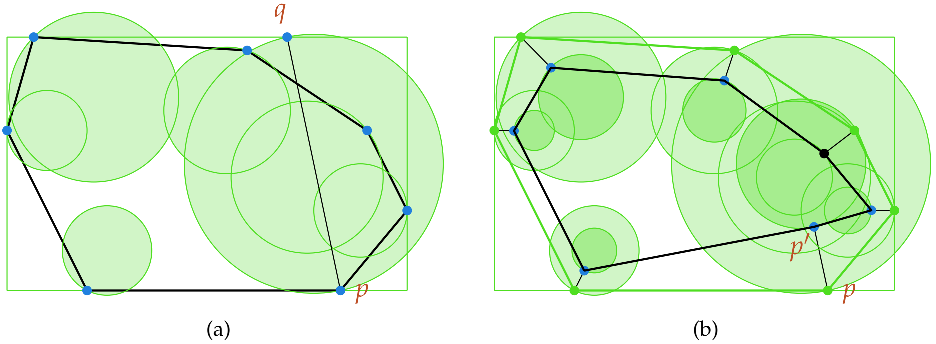


Figure 5.5 (a) The line segment from p to q cannot go too far outside B. (b) Decreasing the radii of the disks by a factor $(1-\delta)$ does not decrease the area of the convex hull by more than a factor $(1-\varepsilon)$.

that contribute to the largest area convex hull, it is not easy to determine where on the disks the points should be, due to algebraic difficulties.

However, we can solve the problem by adding another layer of approximation. If we represent the disks by regular k-gons, for suitably chosen k, we can adjust the algorithm for squares by using $k/2$ cardinal directions. Note that when we model the points as disks, we still have the property that we need to consider only the boundaries, since no vertex of an optimal solution needs to be chosen in the interior of a region.

5.4.1 Approximating Disks by k-gons

Lemma 5.6 *Let ε be given, and let $\mathcal{C}$ be a set of disks. Consider the set $\mathcal{C}'$ of disks with the same centres but radii a factor $(1-\frac{1}{8}\varepsilon)$ smaller. Then the area of an optimal solution for $\mathcal{C}'$ is at least $(1-\varepsilon)$ times the area of an optimal solution for $\mathcal{C}$.*

Proof Let $\delta = \frac{1}{8}\varepsilon$. Let S^* be the optimal solution for $\mathcal{C}$. Let B the smallest enclosing bounding box of S^* with dimensions $w \times h$, and let $\vec{e}_1$ and $\vec{e}_2$ be the unit vectors along the axes of B. Let S' be the solution for $\mathcal{C}'$ achieved by placing the vertices of S^* on the border of the new smaller disk, but at the same angle with relation to the positive x-axis (as seen from the disk centre) as they were before, and taking the convex hull of this new point set.

Let p be a vertex of S^*, and let q be the opposite point on the same disk as p; see Figure 5.5(a). The line segment $\overline{pq}$ cannot be longer than $2w$ in the $\vec{e}_1$ direction and $2h$ in the $\vec{e}_2$ direction, because otherwise choosing q instead of p would yield a better solution (q would contribute an area of more than $\frac{1}{2}wh$, while p contributes at most

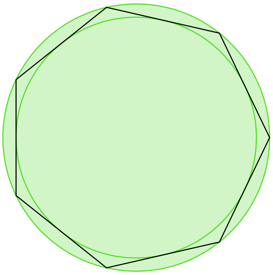


Figure 5.6 A k-gon that fits tightly between the bounding circles of two disks.

$\frac{1}{2}wh$). This means that the point p' in $\mathcal{C}'$ at the same disk as p is at most δw away from p in the $\vec{e}_1$ direction, and at most δh in the $\vec{e}_2$ direction.

Since this is true for all vertices of S^*, the area of S' is at most $4\delta wh$ smaller than the area of S^*; see Figure 5.5(b). Since the area of B is at most twice the area of S^*, this means that the area of S' is at least $(1-8\delta) = (1-\varepsilon)$ times as large as the area of S^*. Of course, the optimal solution for $\mathcal{C}'$ can only be larger. ■

We will now approximate the circular imprecise points by k-gons that lie completely within the band between the circle that bounds the original disk and the disk with a factor $(1-\delta)$ smaller radius; see Figure 5.6. A k-gon fits inside this band when $2k \arccos(1-\delta) \geq 2\pi$, and this can be estimated by $k \geq 2\pi\sqrt{\eta}$. Let $k = \lceil 2\pi\sqrt{\eta} \rceil$, and $\mathcal{G}$ the set of k-gons (with the same orientation) that have their corners on the bounding circles of the disks in $\mathcal{C}$.

Theorem 5.2 *The optimal solution for $\mathcal{G}$ is a $(1-\varepsilon)$ approximation of the optimal solution for $\mathcal{C}$.*

Proof Since all k-gons are contained in the respective disks, the optimal solution for $\mathcal{G}$ is a valid solution for $\mathcal{C}$. Since all small disks are contained in the k-gons, the optimal solution for $\mathcal{C}'$ is also a valid solution for $\mathcal{G}$, and smaller than the optimal solution for $\mathcal{G}$. Now, by Lemma 5.6, the optimal solution for $\mathcal{C}'$ is a $(1-\varepsilon)$ approximation of the optimal solution for $\mathcal{C}$, hence the same is true for the optimal solution for $\mathcal{G}$. ■

5.4.2 Exact Algorithms

Now that we have established that disks can be approximated by regular k-gons, we will study the following problem:

Problem 5.3 *Given a set of regular k-gons in the plane, place a point in each region such that the area of the convex hull of these points is maximised.*

The status of the general version of the problem for regular k-gons is open. In the optimal solution, every point has to be chosen at a corner of its k-gon. Therefore we

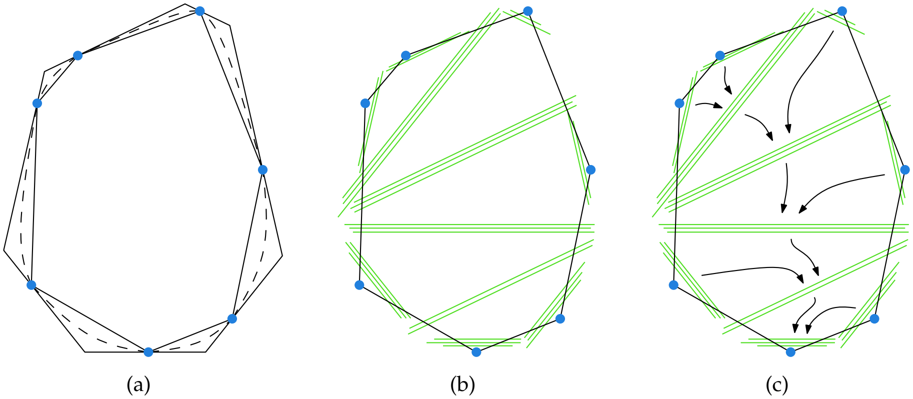


Figure 5.7 (a) The division of the plane for $k = 7$. (b) There are 11 groups of parallel line segments. (c) The arrows indicate order in which the groups can be combined.

can solve the problem in $O(k^n n \log n)$ time by computing the convex hull of every possible set of endpoints. Of course this can be improved slightly.

As in the case of squares, we can achieve a better running time when the regions satisfy certain constraints. If the k-gons are disjoint, we can solve the problem in $n^{O(k)}$ instead of $k^{O(n)}$ time. We can adapt the algorithm described in Chapter 4 in a mostly straightforward manner to the k-gon case. We will briefly discuss the main differences and new ideas that are needed to make the algorithm work.

We need to know the k extreme points of the solution. These are the vertices of the solution that lie furthest in one of the k directions that are perpendicular to the edges of a k-gon. Trying all possibilities gives a factor $O(n^k)$. Suppose the k-gons are disjoint. The k extreme points divide the plane into k triangular regions; see Figure 5.7(a). For each k-gon, we need to consider only the endpoints that are within their respective triangle. Since the k-gons are disjoint, there can be at most $k - 2$ k-gons that intersect more than two of these triangles. For these k-gons, we try every possible combination of their candidate endpoints. This adds a factor $O(k^k)$ to the running time.[2]

The remaining k-gons can now be represented as line segments. There are at most $2k - 3$ groups of line segments; see Figure 5.7(b). We can solve the problem in this situation in $O(kn^3)$ time, using the dynamic programming approach as described in Section 4.1. We start with two consecutive groups that pass over only one extreme point, for which there is no group between them. For these two groups, we compute the optimal solution for every pair of points. Then we combine them with the group that passes over both extreme points. This process is repeated until we have found the optimal solution; see Figure 5.7(c).

---

[2] In fact, a little more work shows it can even be bounded by $O(3^k)$.

5.4.3 Core-Set Construction

A core-set of a set of regular k-gons can be computed in exactly the same way as with squares, as long as $k \geq 4$. The same proof also applies.

5.4.4 Running Time Analysis

Constructing a core-set of size $O(\eta^2)$ takes $O(\sqrt{|V|}|E|)$ time. In our case, we have $O(\eta^{2k})$ nodes at the left side after removing doubles, and $O(\eta^2)$ nodes at the right side, and each left node has exactly k edges, so $|V| = O(\eta^{2k} + \eta^2)$ and $|E| = O(\eta^{2k+\frac{1}{2}})$. This means that the core-set selection algorithm runs in $O(n + \eta^{3k+\frac{1}{2}}) = O(n) + 2^{O(\sqrt{\eta}\log\eta)}$ time, again provided that the ceiling operation takes constant time.

Theorem 5.3 *We can compute a core-set of size $O(\eta^2)$ for Problem 5.3 in $O(n) + 2^{O(\sqrt{\eta}\log\eta)}$ time.*

The general problem for k-gons can be solved exactly in $O(k^n n \log n)$ time. When we choose $k = O(\sqrt{\eta})$ and combine this with Theorem 5.2, then the approximation algorithm takes $O(n) + 2^{O(\sqrt{\eta}\log\eta)} + O(k^{\eta^2}\eta^2\log\eta) = O(n) + 2^{O(\eta^2\log\eta)}$ time in total.

Under the assumption that the disks are disjoint, we also get a disjoint set of k-gons after the first approximation step. For such a set of k-gons, we have a better exact algorithm, which runs in $n^{O(k)}$ time. The approximation algorithm then takes $O(n) + 2^{O(\sqrt{\eta}\log\eta)} + \eta^{O(\sqrt{\eta})} = O(n) + 2^{O(\sqrt{\eta}\log\eta)}$ time in total.

5.5 Closing Remarks

In this chapter, we presented approximation algorithms for computing the upper bound on the area of the convex hull of a set of imprecise points, modelled as squares or disks. The algorithms are based on the core-set paradigm, and the main result is that core-sets of a size that does not depend on n can indeed be computed for these problems. However, the dependence on η mirrors the dependence on n of the exact algorithms we have available, which may, depending on the setting, be quite high.

These results also appeared in [87]. In addition to the results in this chapter, that paper also describes similar algorithms for the case where the imprecise points are modelled as line segments. Furthermore, as mentioned in the previous chapter, in certain special cases more efficient exact algorithms for computing the upper bound exist. For example, in the case where the regions are disjoint unit squares, an $O(n^3)$ algorithm is available [89]. The framework described in this chapter allows for plugging in such algorithms, reducing the dependence on η.

Chapter Six

Bounds on the Diameter

In the last chapter of this part, we will study the *diameter* of a point set. As was already briefly mentioned in Section 5.1.2, given a set of points P, the diameter is defined as the largest distance between any pair of points in P. Figure 6.1(a) shows an example in the plane. The diameter is an important measure of how “large” a set of points is.

A classical result in computational geometry is that the diameter can be computed in $O(n \log n)$ time by first computing the convex hull of the points, and then using a technique called *rotating callipers* [120]. This is an interesting result, since the diameter is defined by only two points, so the output complexity is constant, but still it is as hard to compute (at least using this approach) as the convex hull of the points, which is a much more descriptive structure. However, this dependency disappears when considering the imprecise variants of those algorithms: computing bounds on the diameter seems to be significantly easier than computing bounds on the convex hull.

When the points are imprecise, we are given a set $\mathcal{R}$ of regions, and we want to place a point in each region such that the diameter of the resulting point set is as large or as small as possible, see Figures 6.1(b) and 6.1(c). As in the previous chapter, we consider two models of imprecise points: squares and disks. We show that in the case of squares, both the upper and lower bound on the diameter can be computed in optimal $O(n \log n)$ time. However, it turns out that, as in the convex hull problem, for the disk model there are algebraic difficulties that make it impossible to exactly compute the lower bounds. In this model we can still compute the upper bound in $O(n \log n)$ time, but for the lower bound we present a $(1 + \varepsilon)$-approximation scheme that runs in $O(n^{c/\sqrt{\varepsilon}})$ time for some constant c.

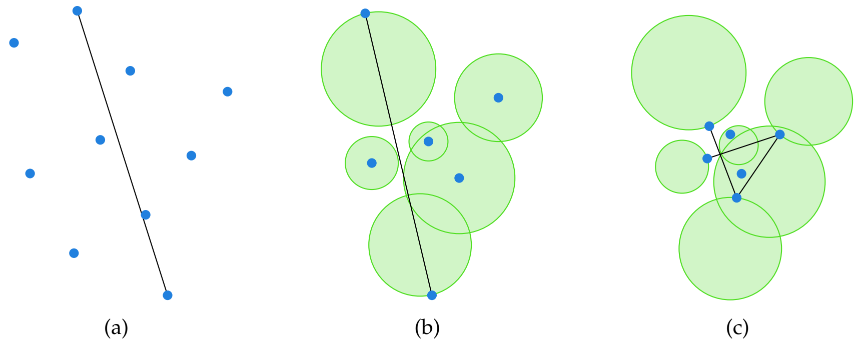

(a) (b) (c)

Figure 6.1 (a) The diameter of a set of points in the plane. (b) The largest possible diameter of a set of imprecise points. (c) The smallest possible diameter of a set of imprecise points, determined by three pairs simultaneously.

6.1 Largest Possible Diameter

In this section, we consider the following problem:

Problem 6.1 *Given a set of disks or squares in the plane, place a point in each region such that the diameter of these points is maximised.*

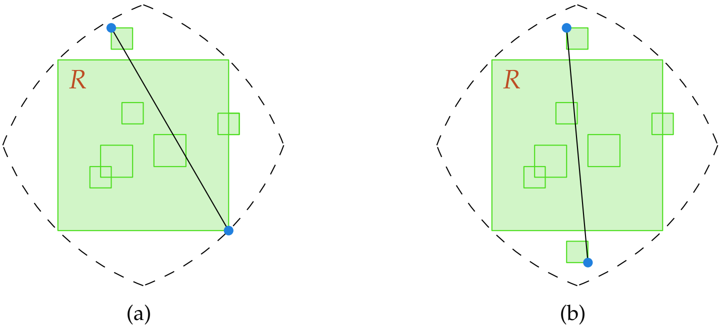
In other words, this means we have to place one point in every region, such that the largest distance between any pair of them is as large as possible. This suggests that we can simply find the two points p and q in the union of $\mathcal{R}$ that are furthest away from each other, using an existing exact algorithm for computing the diameter, and then place two points there.

The problem, however, is that p and q could belong to the same region, and we can place only one point per region. If we are in luck, though, and p and q are in different regions, then we solved the problem: clearly there cannot be any set of points P that has a larger diameter than $|pq|$. To solve the other case, we distinguish the case where the regions are squares from the case where the regions are disks.

6.1.1 Squares

When the points are modelled as squares, the two points forming the largest diameter must be among the corners of the squares. This means we can just compute the diameter of the set of all corners using a conventional diameter algorithm in $O(n \log n)$ time. If the two points found belong to different squares, we are done.

Otherwise, they are diagonally opposite points of one square R, and there are two possibilities. Either the largest diameter is formed by one corner of R and one corner



(a) (b)

Figure 6.2 The largest diameter among all corners is defined by two corners of the same square R. As a result, there are no other squares outside the dashed boundary. (a) The largest possible diameter is formed by one corner of R and one corner of another square. (b) The largest possible diameter is formed by two corners of the other squares.

of another square, as in Figure 6.2(a), or the largest diameter is formed by two points among the other squares, as in Figure 6.2(b). To handle the first case, we can simply check each corner of R with all other corners in $O(n)$ time. To handle the second case, we remove R and compute the diameter of the corners of all other squares. They must lie on different squares, which is shown by the following lemma.

Lemma 6.1 *If the largest diameter of the corners of all squares is formed by two corners of one square R, and there are two points p and q in the remaining squares that are farther away from each other than from R, then p and q must belong to two different squares.*

Proof Both points must lie outside R, because if not we could take a corner of R instead and get a larger diameter. Furthermore, they must lie in two different triangle-like regions (bounded by one side of R and two circular arcs) as in Figure 6.2(b). Now, if they belonged to the same square, this square would have to contain at least one corner of R, which means that it must have a corner outside R, and that corner with the opposite corner of R would have given a larger diameter than R alone, a contradiction. ■

Theorem 6.1 *Let $\mathcal{R}$ be a set of n squares in the plane. We can compute in $O(n \log n)$ time a point set P containing one point from each region in $\mathcal{R}$, such that the diameter of P is maximised.*

6.1.2 Disks

When the regions are disks, we have no corners to restrict the problem to. But the shape of the regions means we can make some observations. Again, we start by

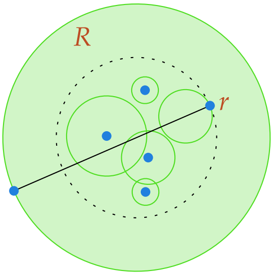


Figure 6.3 When there is one disk R that contains all others, we need to find the point in the union of the remaining disks that is closest to the boundary of R.

computing the two points p and q in the union of $\mathcal{R}$ that are furthest away from each other. This can be done by computing the convex hull of the set of disks in $O(n \log n)$ time [111], and using rotating callipers to find the diameter. As before, if these points belong to different regions, we are done.

However, if p and q belong to the same region R, then we can observe that R must contain all other regions of $\mathcal{R}$. Indeed, if it did not, then there must be one region with a point outside R, and the distance from this point to the furthest point in R would be larger than the diameter of R, contradicting the definition of p and q. In this case, as depicted in Figure 6.3, we have to find the point r among the remaining disks that is closest to the boundary of R, since that also means it is furthest from the opposite point on the boundary of R. In fact, this is the same algorithm as described in Section 3.2.1.2 for computing the largest possible SEC of a set of disks. As observed there, we can easily find r in linear time, by going over the list of disks and computing the distance to the boundary of R for each disk.

Theorem 6.2 *Let $\mathcal{R}$ be a set of n disks in the plane. We can compute in $O(n \log n)$ time a point set P containing one point from each region in $\mathcal{R}$, such that the diameter of P is maximised.*

6.2 Smallest Possible Diameter

In this section, we consider the following problem:

Problem 6.2 *Given a set of disks or squares in the plane, place a point in each region such that the diameter of these points is minimised.*

Computing the smallest possible diameter d of $\mathcal{R}$ is a difficult problem. The reason is that it can be determined by multiple pairs of points simultaneously, as could already be seen in Figure 6.1(c). Moving any of the four points involved would increase the distance between at least one pair of them. In general, all n points could be involved in such a construction, where none of the points can be moved without increasing the diameter. Note that these situations are not degenerate. In the case of disks, it means that we have to resort to approximation algorithms, but we will show that in the case of squares, an exact solution is possible.

For any subset $\mathcal{R}' \subset \mathcal{R}$, let d' be the value of the smallest possible diameter of $\mathcal{R}'$ and let P' be the set of points that achieves it. There will be some pairs of points in P' that have distance exactly d' to each other. These pairs define a graph on P'. We remove any edges in this graph that can be removed by simply moving one point of P' (alternatively, we could say we have an edge between two points only if the distance between them is exactly d' in *any* optimal placement of the points). From now on, we refer to this graph as a *star*.

We can make some observations about the nature of such a star (which also motivate its name). If $\mathcal{R}$ is in general position, the graph will be connected, and every vertex in the graph will have degree 1 or 2, which implies that the graph is actually a path or a cycle.[1] Furthermore, all edges of this graph have the same length d'. Since this is the diameter of P', no two points can be more than d' away from each other. Therefore all edges must intersect each other, and the path makes an angle of at most 60° at each degree 2 vertex. We also call such a vertex a *bend* of the star. A bend is a point that is in balance between its two neighbours, it could move closer to one neighbour but only by moving farther from the other neighbour. The bends are what make the problem hard. Some examples of stars can be seen in Figures 6.1(c), 6.5(b) and 6.9(a).

Ultimately, we want to compute the star of $\mathcal{R}$, which we call the *optimal star*. We first make some observations about this star that are true for any convex regions. Then we show how to solve the problem efficiently for square regions, and we present approximation algorithms for circular regions.

Observation 6.1 *Let p be a point on the optimal star, and let q be an adjacent point on the star. Let ℓ be the line through p perpendicular to $\overline{pq}$. Then no region is entirely on the other side of ℓ than where q is.*

The reason why this observation is true, is that such a region would be more than d away from q, which contradicts optimality of the star. With this observation, we can introduce some more definitions and facts. Let $R \in \mathcal{R}$ be a region. We call R an *extreme* region if there exists a line ℓ that has the interior of R completely on one side, but no other region of $\mathcal{R}$ has its interior completely on the same side of ℓ. We call a point $p \in R$ an *extreme placement* if such a line exists that goes through p.

Observation 6.2 *All points of the optimal star must be on extreme placements in extreme regions.*

Proof Let $R \in \mathcal{R}$ be a region and $p \in R$ a vertex of the optimal star. Consider the vertices q and r of the star that are adjacent to p (possibly there is only one), and consider the lines ℓ_q and ℓ_r through p that are perpendicular to pq and pr. Figure 6.4

---

[1] The assumption of general position is not necessary for the algorithm: if the graph is more complex, then the algorithm will simply return a subgraph of it that is a path or a cycle.

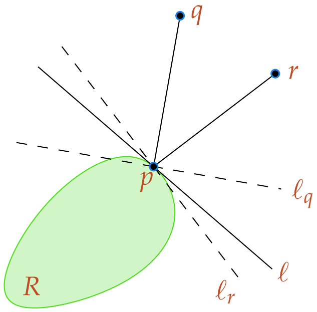


Figure 6.4 A point p on the optimal star. R cannot intersect the section of the plane bounded by ℓ_q and ℓ_r where q and r lie, otherwise p could move closer to q and r. Therefore p is extreme in R.

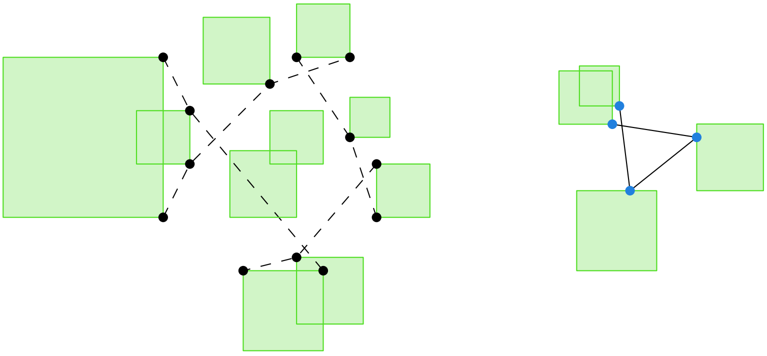
illustrates the situation. Because p is in its optimal position, there is no point in R that is closer to both q and r than p is. Since R is convex, this means there exists a line through p that has the interior of R completely on one side. By Observation 6.1, there are no other regions completely on the other side of ℓ_q than q, and no regions completely on the other side of ℓ_r than r. Therefore, there are no other regions completely on the same side of ℓ as R. ■

In Section 3.3.2, we defined the critical sequence of a set of regions. In almost every direction, there is exactly one extreme region, except for the critical directions where a line is tangent to two regions simultaneously. The order in which regions become extreme when we rotate through all directions is the *critical sequence* of the regions. The critical sequence can be computed in $O(n \log n)$ time, and plays an important role in the following sections.

6.3 Smallest Diameter for Squares

When the points are modelled as squares, we can solve the smallest diameter problem in $O(n \log n)$ time. We first investigate where the vertices of the optimal star could be. Among the extreme squares, there are only four with infinitely many extreme placements, being the squares with the topmost bottom side, the bottommost top side, the leftmost right side and the rightmost left side. We call these squares *axis-extreme*. The other extreme squares can have an extreme placement at only one corner. As already noted in Section 4.3, these placements form four *chains*: the top left chain connects all bottom right extreme placements, etc., see Figure 6.5(a). Note that these chains are convex. The extreme squares and chains can be computed in $O(n \log n)$ time by computing the critical sequence of the squares.

Assuming the squares are in general position, the optimal star cannot have bends at


(a)

(b)

Figure 6.5 (a) Four extreme squares and four chains. (b) A star with two bends.

corners of squares. If there would be a bend p at a corner, there would be at least one neighbour q of p such that p would not be able to move closer to q, even if it had no other neighbour. This means p is not in balance, making this in fact a degenerate case and not a real bend. Therefore, the only interesting bends occur at axis-extreme squares. Together with the fact that all edges of a star must intersect each other, this means that the optimal star can have at most two bends, as in Figure 6.5(b). This implies that we can find it efficiently, as we will now show.

Lemma 6.2 *We can find the optimal star by computing the star of every subset of four squares, of which two are axis-extreme, and reporting the largest among these.*

Proof The algorithm returns a star for subset $\mathcal{R}' \subset \mathcal{R}$. Let this star have diameter d'. Clearly, the diameter d of $\mathcal{R}$ must then be at least as large as d'. On the other hand, it cannot be larger, because the optimal star has only four vertices, of which two are on axis-extreme squares, so this set of squares must have been considered too. Therefore, $d = d'$. ■

As an immediate result, we can solve the problem in $O(n^2)$ time by enumerating all these sets of squares and computing their stars. However, we will now show that after precomputing the chains of possible extreme placements we can also find the optimal star in linear time, by using a careful case analysis and using the structure of the chains. Together with the computation of these chains, this yields an $O(n \log n)$ algorithm. In overview, for every placement of a point on an axis-extreme square, there is one vertex on one of the chains that is furthest away from it, and this determines the best possible diameter in this case. As the axis-extreme point moves over its edge, this furthest vertex can move only in restricted ways, which saves us a linear factor.

Assume we have computed in $O(n \log n)$ time the four axis-extreme squares and the four chains of extreme squares, as in Figure 6.5(a). Since the optimal star has at most

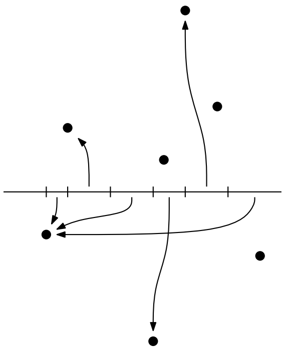


Figure 6.6 The interval division of a set of 7 points.

two bends and its edges must intersect each other, it can be of only three different types. It may be a single connection between two squares without any bends, or a star with one bend on an axis-extreme square and two endpoints on the opposite chains, or a star with two bends on consecutive axis-extreme squares and two endpoints on the same opposite chain, see Figure 6.7. We will try all cases and all symmetric possibilities within a case, and compute the largest possible valid star in each case in linear time. The largest among these must be the optimal star.

6.3.1 Interval Division

In order to solve the different cases, we will need a simple structure that divides a line into intervals. Given a set of points P, we define a function $f(x)$ as follows. Let $L_P(x) \subset P$ be the set of those points in P with an x-coordinate at most x. Then $f(x)$ is the furthest point in $L_P(x)$ from the point $(x, 0)$. We divide the x-axis into intervals where f is constant, see Figure 6.6. There will be an unbounded interval to the left of the leftmost point where f is not defined.

Lemma 6.3 *The interval division has linear complexity and can be computed in $O(n \log n)$ time.*

Proof We can compute the interval division incrementally by sorting the points by increasing x-coordinate, and inserting them in that order. When we know the interval division of the x-axis with respect to the points $\{p_1, \ldots, p_i\}$, and we insert the next point p_{i+1} with x-coordinate x_{i+1}, we observe that at most one new interval can be created and that this new interval must start at x_{i+1}. The new interval may (partially) overlap any number of existing intervals, which we can find by scanning to the right until the new interval ends. With each new point, only a single new interval is created, therefore only a linear number of intervals is created in total. This means that also only a linear number can be overwritten during the scans, and the total time spent is linear. ■

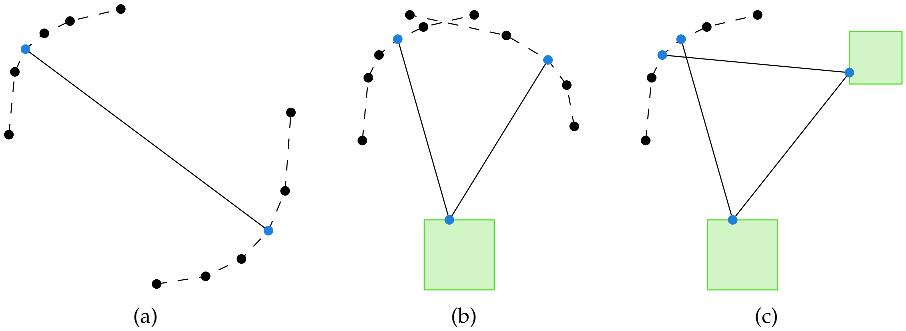


Figure 6.7 (a) The optimal star is a direct connection between two chain vertices. (b) The optimal star has one bend in an axis-extreme square. (c) The optimal star has two bends in consecutive axis-extreme squares.

We define the interval division of any directed line by transforming that line onto the x-axis.

Case 1: no bends. When the optimal star contains no bends, it is a direct connection between two regions, see Figure 6.7(a). This can either be a horizontal or vertical connection between two opposite axis-extreme squares, or a diagonal connection between two vertices of opposite chains. The former case can be computed in constant time. In the latter case we need to ensure that we consider only pairs with the right slope: positive for a connection between the bottom left and top right chains, negative for a connection between the top left and bottom right chains. This can be computed by a simple variation to the conventional diameter algorithm using rotating callipers [120] in linear time.

Case 2: one bend. When the optimal star has exactly one bend, this bend is on an axis-extreme square, say the bottommost square, see Figure 6.7(b). The start and end points of the star must be vertices of the top left and top right chains. To find the largest star of this type, we must find the point p on the bottommost square such that the distance to the furthest point on both chains is minimised. However, we must consider points of the top left chain only if they are to the left of p, and points of the top right chain only if they are to the right of p.

Let l be the horizontal line through the top side of the bottommost axis-extreme square. We compute the interval division of the directed line l from left to right with respect to the set of points that form the top left chain, and we also compute the interval division of l directed from right to left with respect to the points in the top right chain. We can now balance the furthest points on both chains in linear time.

Case 3: two bends. When the optimal star has two bends, these bends must occur at consecutive extreme squares, say the bottommost and rightmost ones, see Figure 6.7(c). The start and end points are then vertices of the top left chain. We must now find the point p on the bottom square and the point q on the right square that minimise the largest among the distance from p to the furthest vertex of the chain to the left of p, from q to the furthest vertex of the chain above q, and from p to q.

To find these optimal positions, we again compute the interval division of the line from left to right through the top side of the bottommost axis-extreme square with respect to the points on the top left chain, and we similarly partition the line from top to bottom through the left side of the rightmost axis-extreme square with respect to the same point set. Now we place p at its locally optimal position, and q too. The furthest distance must occur between p and q, otherwise the optimal star was not of this type. Now start moving the points towards each other, keeping their distances to the furthest point on the chain equal. They both move in only one direction, so we can find the optimal location again in linear time.

6.3.2 Placing the Points

We have now computed the value d of the smallest possible diameter. If needed, we can also compute a placement of the points in their regions that realises this diameter. We first compute valid placements of the four axis-extreme points, and then observe that all other points should be placed “as far inwards” as possible.

To compute a valid placement in an axis-extreme square, note the following. If we find a placement such that any star that includes this point has at most length d, then this placement is valid for a global solution of diameter d. This means we can check for all possible stars in which interval the point is allowed to lie if that star must be at most d long, and then place the point somewhere in the intersection of all these intervals.

To determine the intervals of an axis-extreme square, say the bottommost, where a point could be placed that still allows for a solution of length d, we compute for every square the interval that is at most d away from that square. The intersection of these intervals gives an interval where a point can still be placed that is at most d away from any other square, see Figure 6.8. After that, we must still place the axis-extreme points in such a way that they are at most d away from each other. However, we already know that this is possible and since there are only four axis-extreme points we can place them in constant time. Any placement within the precomputed intervals will be fine with respect to the $n-4$ other squares.

For the rest of the squares, if a point is to the left of the vertical lines through the topmost and bottommost axis-extreme points, moving it to the right can only decrease the diameter, and a similar statement holds for the other extremes. Because the regions are squares, every point will end either in a corner of its region or somewhere in the middle of the whole construction.

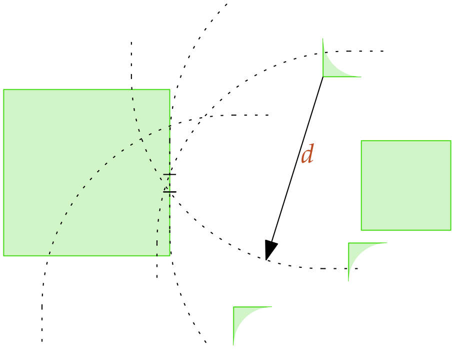


Figure 6.8 (b) The leftmost axis-extreme points must be placed within the dotted circles.

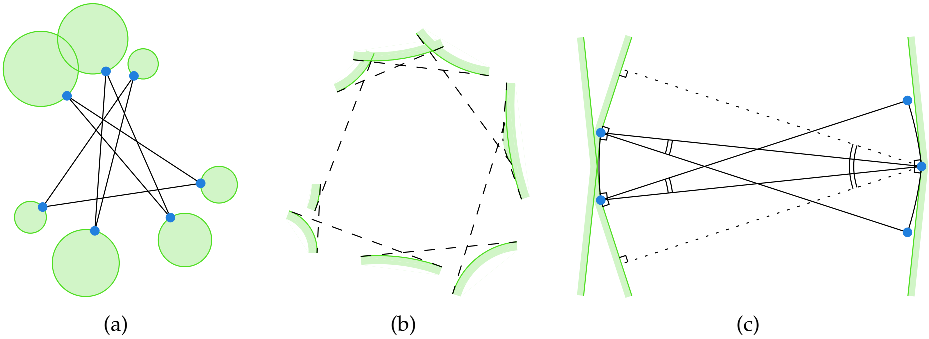

(a)
(b)
(c)

Figure 6.9 (a) A cyclic star that visits seven regions. (b) Circular arcs that are extreme in some direction form the critical sequence of the disks. (c) Three consecutive angles of at most α give a diameter of $\cos\alpha$.

Theorem 6.3 *Let $\mathcal{R}$ be a set of n squares in the plane. We can compute in $O(n \log n)$ time a point set P containing one point from each region in $\mathcal{R}$, such that the diameter of P is minimised.*

6.4 Smallest Diameter for Disks

When the points are modelled as disks, stars can have up to n bends, see Figure 6.9(a). This leads to algebraic difficulties: even if we knew the combinatorial structure of the optimal star, computing the positions where the points are in balance exactly would not be possible.

We can, however, make some observations about the combinatorial structure. We can still define the extreme disks, that is, disks that have a tangent line with no other disk completely on the same side. These disks are on the critical sequence, see Figure 6.9(b). Any bend in the star still needs to be on such an extreme arc. However, it is possible that all disks have extreme arcs.

Since we cannot compute the optimum efficiently, we provide an approximation algorithm instead. We can compute a $(1+\varepsilon)$-approximation in $O(n^{3\pi/\sqrt{\varepsilon}})$ time. The idea of this algorithm is to consider only stars of at most k bends, for k chosen suitably. We compute a subset $\mathcal{R}' \subset \mathcal{R}$ for which we can show that the optimal solution d' is at most a factor $(1-\frac{1}{2}\varepsilon)$ shorter than the real optimum d. Therefore, we get a $1/(1-\frac{1}{2}\varepsilon)$-approximation, which is a $(1+\varepsilon)$-approximation if $\varepsilon \leq 1$.

Lemma 6.4 *Suppose the optimal solution is given by a star of at least k bends. Then there exists a star with only one bend that approximates it within a factor of $1 - O(k^{-2})$.*

Proof Suppose that the optimal diameter is 1. The sum of the angles that the star makes in the bends is at most π. That means that there are three consecutive bends a, b and c somewhere that together make an angle of at most $\alpha = \frac{3\pi}{k}$. Therefore the individual angles are also at most α, see Figure 6.9(c).

The region $R \in \mathcal{R}$ that supplied the point b is convex and completely outside the wedge that is formed by the halfline from b perpendicular to $\overline{ab}$ in the direction of c, and the halfline from b perpendicular to $\overline{bc}$ in the direction of a, otherwise b would not be in its optimal position. In the same way, the regions of a and c are also convex and behind the wedges formed by their two neighbours on the star, unless one of them is the endpoint of the star and the wedge degenerates to a line.

This means that in the worst case the regions of a and c are arbitrarily close to the halflines that come nearest to b, and in that case the best possible diameter of a, b and c would be $\cos\alpha$, as denoted by the dotted lines in Figure 6.9(c). This is more than $1 - \frac{1}{2}\alpha^2 = 1 - \frac{9\pi^2}{2k^2}$. ■

To get a $(1-\frac{1}{2}\varepsilon)$-approximation we take $k = 3\pi\varepsilon^{-\frac{1}{2}}$ (and at least 3) and compute all chains of length at most k in $O(n^k)$ time. If the optimal star has at most k bends, we will find it. If not, then by Lemma 6.4, there exists a good approximate star of length 3, which we will find.

Theorem 6.4 *Let $\mathcal{R}$ be a set of n disks in the plane. We can compute in $O(n^{3\pi/\sqrt{\varepsilon}})$ time a point set P containing one point from each region in $\mathcal{R}$, such that the diameter of P is at most $(1+\varepsilon)$ times as large as the minimum.*

6.5 Closing Remarks

In this chapter, we studied the problem of computing the upper and lower bound on the diameter of a set of imprecise points, modelled as squares or disks in the plane. We provided $O(n \log n)$ time algorithms for computing the upper bound in both cases. The lower bound for a set of squares can also be computed in $O(n \log n)$ time, but for a set of disks this is not possible due to algebraic difficulties. Instead, we presented an approximation scheme.

These results can also be found in [90], together with the results in Chapter 3. In addition to the results mentioned here, we also present a simple constant factor approximation algorithm for computing the smallest diameter of disks in that paper. Furthermore, we also discuss the *closest pair* of a set of points: the pair of points that have the smallest distance, rather than the largest distance. Computing the lower bound on this measure can be shown to be NP-hard with a direct application of [45], while the upper bound can be computed in $O(n \log n)$ time rather easily, both for disks and for squares.

PART III

Preprocessing Imprecise Points

Chapter Seven

Preprocessing for Triangulation

In this part of this thesis, we consider a different approach to dealing with imprecise data. Instead of computing explicit bounds on the values of things we want to compute, we compute something that gives no information by itself, but somehow captures the imprecise data in the sense that if the data is later given without imprecision, we can process it more quickly. In a sense, instead of treating imprecise points as points of which we do not know where they are, we treat them as indications of where they are, and see how much information this gives us.

This point of view assumes that even though the data is imprecise, we will at some point have access to the precise data. Although this is usually not the case, it is often possible to obtain better approximations of the points later. This is related to the *update complexity* model [47], in which each data point is given imprecisely at the beginning but can always be found precisely at a certain price. In our setting, we assume that the points will all become available together at some point in the future, and we want to do as much of the computation as possible now.

Alternatively, we could interpret the "real points" that become available later as sampled points. If we do not have the precise points available, but we do have information about how the points are distributed and we want to do some analysis on a particular structure on the points, then we want to sample a large number of point sets and for each of these compute the true structure, as described in Section 2.3.2. Then it would be useful if as much of the computation as possible was already done on the imprecise points.

From a theoretical point of view, it is interesting to know just how much of the hardness of a certain problem comes from the exact placement of the points, and how much from the more general layout. The setting is that we have an unknown point set P and we want to compute some structure $S(P)$. We do have available a set of imprecise points $\mathcal{R}$, such that each point of P comes from one region in $\mathcal{R}$. Now

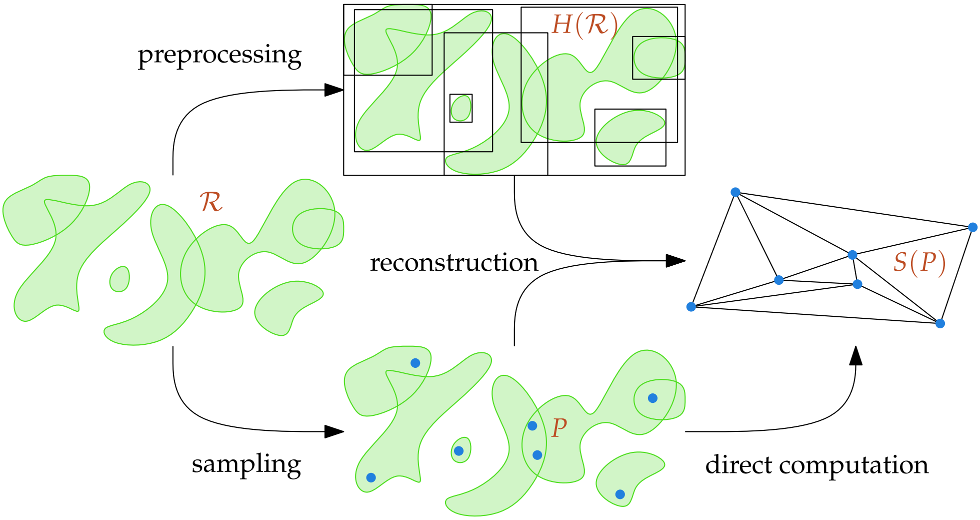


Figure 7.1 An example showing the data flow of the preprocessing and reconstruction algorithms, compared to direct computation.

we want to compute an intermediate structure $H(\mathcal{R})$ that can aid us in computing $S(P)$ once we know P. For this we need two algorithms: a *preprocessing* algorithm that computes $H(\mathcal{R})$ from $\mathcal{R}$, and a *reconstruction* algorithm that computes $S(P)$ from P and $H(\mathcal{R})$. Clearly, both algorithms together can never be faster than the fastest algorithm to compute $S(P)$ from P directly. But the question is how fast the second algorithm can become, by doing as much of the most time-consuming computation as possible in the first algorithm. Figure 7.1 illustrates this idea.

One way to achieve results of this type is by using a new technique called *scaffolding*. In this framework, to compute $S(P)$, we first construct another point set Q, which is called a *scaffold*, that somehow captures the “typical” layout of the unknown point set P. Next, we compute the structure $S(Q)$ using an existing exact algorithm. These two steps together form the preprocessing phase, and the data structure $H(\mathcal{R})$ corresponds to $S(Q)$, possibly enhanced with additional structure. Then, once they are known, we insert the points of P into the scaffold using an incremental algorithm that can update the structure S until we have $S(P \cup Q)$ available. Finally, we remove the scaffold using a *splitting* algorithm (also called a *hereditary* algorithm) until we are left with only $S(P)$. Splitting algorithms are a relatively new concept, but some results are available. Chazelle *et al.* [26] show how to split a Delaunay triangulation in linear expected time. Chazelle and Mulzer [29] show how to split a 3D convex hull, also in linear expected time.

7.1 Triangulations

In this first chapter, we will consider triangulations of P as structures to compute. We show that a set of imprecise points, modelled as disjoint polygonal regions, can be preprocessed in $O(n \log n)$ time, such that a triangulation of the real points can be computed in linear time once they are known.

A *triangulation* is a subdivision of the convex hull of a set of points into triangles, such that the vertices of the triangles correspond exactly to the input points. Triangulations are important composite geometric objects. A set of points can be triangulated in $O(n \log n)$ time. However, it is not possible to triangulate a point set faster, for example because any triangulation has to sort the vertices along the convex hull, and sorting is well-known to have an $\Omega(n \log n)$ lower bound. However, for some classes of regions it is possible to construct a preprocessing algorithm that takes $O(n \log n)$ time, and a reconstruction algorithm that runs in $O(n)$ time.

We are given a set of regions $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$. The properties (shape, size, overlap, etc.) of these regions will vary, as in the previous part. We assume that later we get a set of precise points $P = \{p_1, p_2, \ldots, p_n\}$ and that we know for each point p_i which region R_i it belongs to.

It is not always possible to preprocess a set of imprecise points to speed up the reconstruction algorithm. If all regions in $\mathcal{R}$ have a common intersection (that contains a small disk), then we could still get any point set as input, and lower bounds for the direct computation still apply to the reconstruction algorithm.

Preprocessing imprecise points for triangulation has been studied before by Held and Mitchell [65]. They consider the problem of preprocessing a set of n disjoint unit disks in $O(n \log n)$ time, such that when one point in each disk is given, the point set can be triangulated in linear time. They give a simple and practical solution. Their result can be extended to overlapping regions of different shapes, as long as the regions do not overlap more than a constant number of other regions, the regions are "fat" in some sense, and the sizes do not vary by more than a constant.

Often, regions are more complex than disjoint unit disks. They can have different sizes, shapes, or they can partially overlap. We show in this chapter that the result for triangulation can be generalised to regions of arbitrary shapes, as long as they remain disjoint. The approach can be generalised to partially overlapping regions.

We apply the scaffolding paradigm to solve the problem. The main problem in this case is to split a triangulation: given a triangulation in the plane with red and blue vertices, we want to compute a triangulation of only the blue (or red) vertices in linear time. Chan [24] shows how to compute the convex hull of a subset of the vertices of a simple polygon in linear time, and a triangulation in $O(n \log^* n)$ time. In the next section, we show this can be improved by providing a linear-time splitting algorithm for triangulations. In the section after that, we apply this result to the original problem.

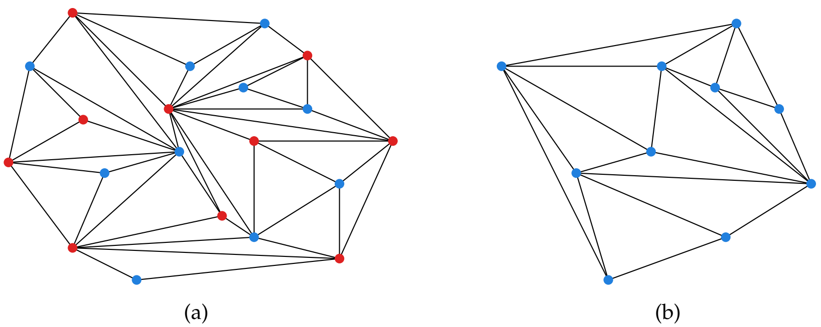


Figure 7.2 (a) Example input with red (open) and blue (solid) points. (b) Example output.

7.2 Splitting a Triangulation

Before solving the preprocessing problem, we will first study the following problem in this section:

Problem 7.1 *Given a triangulation embedded in the plane with vertices that are coloured either red or blue, compute a triangulation of only the blue vertices.*

Figure 7.2 shows an example of this problem.

To solve the problem, we will remove all of the red points one by one, until we have only blue points left. During this process, we will maintain a subdivision of the plane with certain properties, which allow us to quickly find new red points to remove and to remove them efficiently. We first describe this subdivision and some operations we can perform on it, and then give the algorithm and time analysis.

7.2.1 Structure and Operations

During the algorithm, we will maintain a subdivision of the plane that uses the blue points and remaining red points as vertices. The subdivision is a special kind of *pseudotriangulation*. A pseudotriangulation is a subdivision of a convex region into *pseudotriangles*: simple polygons with exactly three convex vertices. The three convex vertices are also called the *corners* of the pseudotriangle, and the three polygonal lines connecting each pair of corners are called the *sides* of the pseudotriangle. (Note that we do not require a pseudotriangulation to be "pointy" or "minimal".)

In the pseudotriangulation that we maintain, we allow only two types of faces: triangles and *foxes*. A fox is a pseudotriangle that has only one side which is not a straight edge, and for which all vertices along this side are blue, and the one remaining

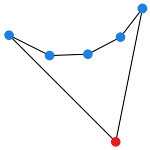


Figure 7.3 A fox is a pseudotriangle with one red vertex and one concave chain of blue vertices.

vertex is red. We call the red vertex the *chin* of the fox, and the other two convex vertices the *ears*. Figure 7.3 shows an example of a fox. For each fox, we store the chain of concave blue vertices in a balanced binary tree. If a pseudotriangulation has only triangles and foxes as faces, we call it *happy*.

Note that our input triangulation is happy, since it has only "normal" triangles and no pseudotriangles. Also note that if we manage to remove all red vertices and maintain a happy subdivision, we cannot have any foxes left, since they have a red vertex: we are left with only triangles with three blue vertices, which is the required output of the algorithm.

Whenever we have a happy subdivision, we will denote the number of blue points by n and the number of remaining red points by k.

For a given red point p, let $r(p)$ be the number of red neighbours of p and $b(p)$ the number of blue neighbours of p (i.e., $r(p) + b(p)$ is the degree of p). Observe that any face which p is incident to is either a triangle or a fox that has p as its chin. As a consequence, the union of all faces incident to p forms a star-shaped polygon. By $c(p)$ we denote the total complexity of this polygon.

In addition to the shape restriction on the pseudotriangles, we will pose one more condition that we will maintain throughout the algorithm. For all red points p, Condition ($*$) should hold:

$$b(p) \leq 2 \cdot r(p) + 3 \tag{$*$}$$

This condition is not necessarily true for the input triangulation, so we will have to do an initial pass over the input triangulation to make this condition hold.

We will define some useful operations that we can perform on this subdivision.

7.2.1.1 Simplifying a Red Point

Lemma 7.1 *Let p be a red point in a happy pseudotriangulation. We can make Condition ($*$) hold for p in $O(c(p))$ time.*

Figure 7.4 shows an example. Since p is a red point, and the pseudotriangulation is happy, the region around p (the union of its incident cells) is a star-shaped polygon of which all red and all convex vertices are connected to p. The purpose of this step is to

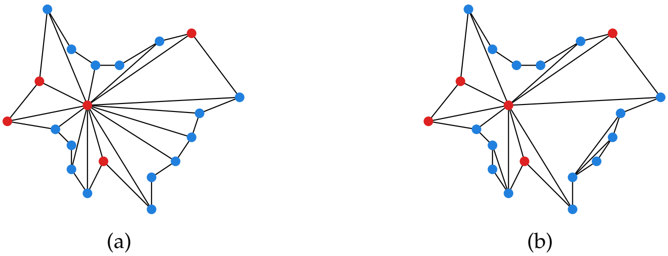


Figure 7.4 (a) The pseudotriangles incident to a given red point form a star-shaped region. (b) By adding and removing the appropriate edges, we can make Condition (∗) hold.

remove any superfluous red-blue edges that p has. We leave all red-red edges where they are, so we do not need to consider them. Between each pair of red neighbours of p, there is a sector of the star that has only blue points (we will consider the case where p has no red neighbours separately). We will first prove the following lemma for a single sector.

Lemma 7.2 *Let p be a red point, and let q_1 and q_2 be two red neighbours of p such that there are no other red neighbours of p between them. Let b be the number of blue neighbours of p between q_1 and q_2, and c the total number of blue points between q_1 and q_2. In $O(b \log c + m)$ time or in $O(c)$ time, where m is the number of blue-blue edges produced in this step, we can update the subdivision such that the number of red-blue edges from p to a point between q_1 and q_2 is at most 2 if $\measuredangle q_1 p q_2 \leq 180^\circ$, and at most 3 otherwise.*

Proof We have a sequence of blue points, some of which may be connected to p. The first and last points are always connected to p, since their neighbours are red, and any face of the subdivision with two red vertices must be a triangle. Now, if any other point s is also connected to p, we can almost always remove it. There are two cases.

If the angle at s after removing edge ps is concave, we can simply remove the edge. Both neighbours of s are blue, so the two cells ps separates must be foxes (or triangles with one red and two blue vertices, which are degenerate foxes). Since s is also concave, the combination of both cells is still a valid cell. In this case, we do need to concatenate the two binary trees that store the chains between the ears of the foxes. We postpone this concatenation until the end of the procedure.

If the angle at s after removing edge ps is convex, we can still remove the edge if the triangle formed by s and its two neighbours is empty. If this is the case, we add the edge between the neighbours of s, forming a completely blue triangle, then add edges from the two neighbours of s to p, and recurse. If the triangle is *not* empty, then we must keep ps. However, this is possible only if p itself is inside this triangle, which can happen at most once and only if the angle of the sector is at least 180°.

After all superfluous red-blue edges have been removed, we might be left with a sequence of $O(b)$ blue chains, stored as balanced binary trees, of total complexity $O(c)$, which have to be concatenated into one big balanced binary tree. We note that this can be done in $O(b \log c)$ time by merging them one by one, or in $O(c)$ time by simply building a new tree from scratch. ■

With this result, we can prove Lemma 7.1.

Proof There are two cases to consider. If p has any red neighbours, we apply Lemma 7.2 to all sectors, using the $O(c)$ time complexity algorithm. There can be at most 2 sectors with an angle of at least 180°, so the number of red-blue edges after simplifying all sectors is at most $2r(p) + 2$.

On the other hand, if p does not have any red neighbours, we can still proceed with deleting blue edges as described above, until only three neighbours remain. In fact, in this case we are just triangulating the star-shaped polygon that remains after taking p out. In both cases, Condition ($*$) follows. ■

7.2.1.2 Subdividing a Pseudotriangle

Lemma 7.3 *Let T be a pseudotriangle with the property that all concave vertices are blue. We can subdivide T into $O(1)$ non-blue triangles and foxes plus some number of triangles that are completely blue, in time $O(\log c + m)$ where c is the complexity of T and m is the number of blue-blue edges we produce in this step.*

Proof If none of the sides of T have any concave vertices, then T is already a triangle.

If only one of the sides has concave vertices, and the corner opposite to it is blue, then we can triangulate the pseudotriangle with edges from the blue corner to all of the concave vertices, see Figure 7.5(a). This creates many blue triangles, and at most two triangles that involve a red point. If the corner opposite to it is red, then depending on the colours of the other two corners we either make an edge to the neighbours or not, see Figure 7.5(b). In this case, we make one fox and at most two triangles on the sides.

If two of the sides of the pseudotriangle have a concave vertex on them, consider the corner between these two sides. We can add an edge between its two neighbours, which are both blue. We can then continue adding blue-blue edges between the two chains, until this is no longer possible. The part that is left is then either a quadrilateral, see Figure 7.5(c), which we can simply split into two triangles, or a pseudotriangle with at most one side with concave vertices, see Figure 7.5(d), which we can further split in the way described above.

If all three sides have concave vertices on them, consider one of the corners. If we can make an edge between this corner and one concave vertex of the opposite side, then this splits the pseudotriangle into two pseudotriangles with both at most two sides with concave vertices, see Figure 7.5(e), which we can then further split as described above. We can find out whether there is such an edge in $O(\log c)$ time by extending

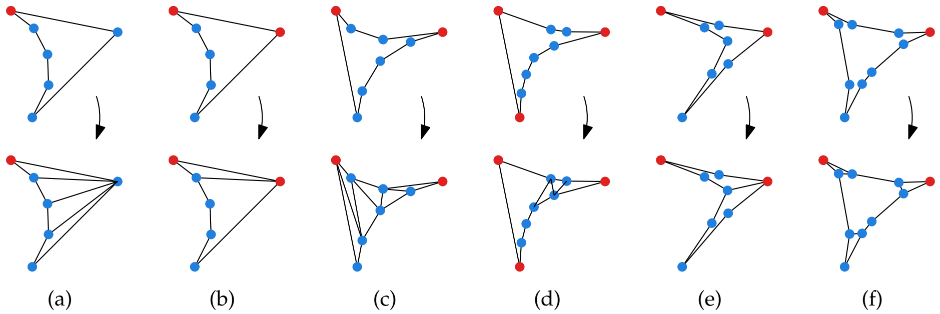


Figure 7.5 We can subdivide any pseudotriangle into $O(1)$ triangles and foxes, plus a number of polygons that have only blue vertices.

the edges adjacent to the corner, and intersecting them with the opposite chain. If this is not possible for any corner, then we can connect the two neighbours of each corner with a blue-blue edge, see Figure 7.5(f). The remaining area has only blue vertices, and can be triangulated in any way. ■

7.2.1.3 Removing a Red Point

Lemma 7.4 *Let p be a red point in a happy subdivision with $r(p) = O(1)$, for which Condition $(*)$ holds. We can remove this point from the subdivision, and partition the gap it leaves into triangles and foxes in $O(\log c(p) + m)$ time, where m is the number of blue-blue edges formed in this step.*

Proof An example of a red point to be removed is shown in Figure 7.6(a). Because of Condition $(*)$, we know that also $b(p) = O(1)$. We can remove p and all its incident edges, of which there are only a constant number. This results in an empty star-shaped polygon which needs to be partitioned into smaller cells again, see Figure 7.6(b). The complexity of this polygon is $c(p)$, and consists of $r(p)$ red points and at most $b(p)$ concave chains of blue points.

We will partition the gap into pseudotriangles. As described in [116], we can add geodesic shortest paths between pairs of convex vertices of a simple polygon, until a (minimal) pseudotriangulation has been found. Since we have only a constant number of convex vertices, we need to insert only a constant number of shortest paths. For a given pair of vertices, we can compute this shortest path in $O(\log c(p))$ time, because we stored the chains of concave vertices in binary trees and we can compute tangents in logarithmic time. After this procedure, the gap has been split into a constant number of pseudotriangles in $O(\log c(p))$ time, see Figure 7.6(c). All the pseudotriangles have only blue concave vertices. We may have to split these binary trees that store the blue vertices into smaller parts, but only a constant number.

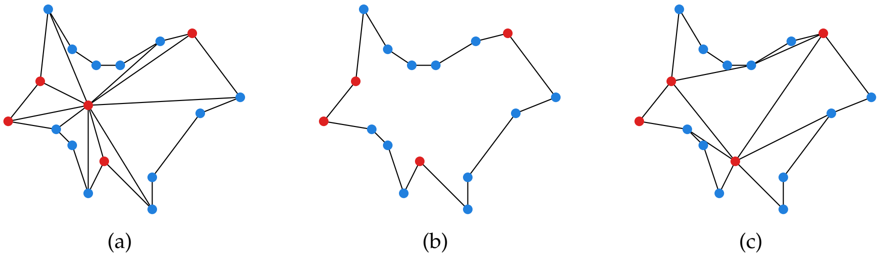


Figure 7.6 (a) A red point p of constant red degree and its incident pseudotriangles. (b) The empty polygon after removing p. (c) A repseudotriangulation of the gap.

One split can be done in logarithmic time, so this also takes $O(\log c(p))$ time for all trees together.

We now apply Lemma 7.3 to all of these pseudotriangles to obtain a partitioning of the gap into a constant number of triangles and foxes, plus any necessary number of completely blue triangles. ■

We now have a happy subdivision again, although Condition $(*)$ may no longer be true for some red vertices on the boundary of the gap.

7.2.2 The Algorithm

With these operations and lemmas, we are now ready to describe the algorithm to solve Problem 7.1. That is, we are given a triangulation with red and blue vertices, and we want to compute a triangulation of only the blue vertices.

First, we must make sure that all red points in the pseudotriangulation satisfy Condition $(*)$. To this end, we simply apply Lemma 7.1 to all red points. This clearly takes linear time in total. Next, we perform a sequence of reduction steps, each time reducing the number of red points by removing a constant fraction.

When we have k red points left, we want to find an independent set of $\Theta(k)$ red points that all have constant red degree. Since this step does not depend on any blue points, and because of Condition $(*)$, this can easily be done in $O(k)$ time.

Then, we remove each of those points by applying Lemma 7.4. The resulting subdivision is still happy, but Condition $(*)$ may not hold anymore for red points that had a red neighbour that was removed. However, we can repair this condition by applying Lemma 7.2 to those red points, but only in the sectors where something changed. The number of red-blue edges in those sectors cannot have increased by more than the number of edges that were added in the removal step. We added no more than a constant number of red-blue edges for each removed point, so this is in total at most $O(k)$.

7.2.3 Time Analysis

The first phase takes $O(n)$ time.

Then, let $1/f$ denote the fraction of the red points that remain after throwing some away in each step (so $f > 1$). Then we perform $\log_f n$ phases, with at the ith phase $k = n/f^i$ red points left. In each phase, we spend $O(k)$ time to find an independent set. Then, we spend $O(\log c(p) + m)$ time for each element in the set. We can charge the m to the blue-blue edges that are created; since there can be at most $O(n)$ blue-blue edges and they are never removed, we spend no more than $O(n)$ time in total on them. The $c(p)$ terms are added over all elements in the independent set, and can be no more than $O(n)$ in total: $\sum_p c(p) = O(n)$. In the worst case, they are divided equally, and we spend $O(k \cdot \log \frac{n}{k})$ time on removing the points. By Lemma 7.2, Condition $(*)$ can be repaired in a sector that was involved in a removal step in $O(\log c(p) + m)$ time, since the number of red-blue edges in such a sector is constant. There are at most $O(k)$ such sectors, so again, in the worst case all blue points are equally divided and we spend $O(k \cdot \log \frac{n}{k})$ time on repairing them.

The total time we spend is now:

$$\sum_{i=0}^{\log_f n} O(\frac{n}{f^i} \log f^i) = \sum_{i=0}^{\log_f n} O(f^i \log \frac{n}{f^i}) = \sum_{k=1}^{n} O(\log \frac{n}{u(k)})$$

where $u(k)$ is the smallest power of f larger than k. We can interpret this as the summation over k of the amount of time charged to removing one red point when there are k left. This time bound is then bounded by:

$$\sum_{k=1}^{n} O(\log \frac{n}{k}) = O(\log \frac{n^n}{n!}) = O(n \log n - \log n!) = O(n)$$

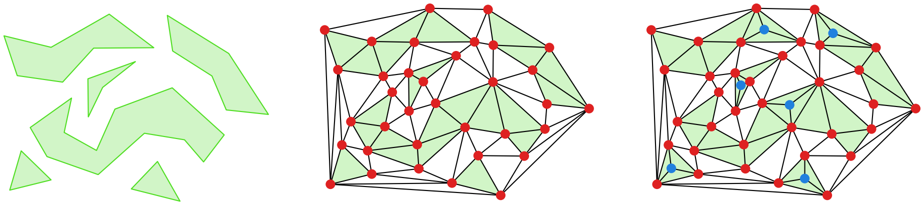
Theorem 7.1 *Given a triangulation embedded in the plane with n vertices that are coloured either red or blue, we can compute a triangulation of only the blue vertices in $O(n)$ time.*

7.3 Triangulating Imprecise Points

In this section, we consider the following problem:

Problem 7.2 *Given a set of disjoint polygonal regions in the plane, preprocess them in such a way that when a point in each region is given, a triangulation of these points can be computed faster than without preprocessing.*

Figure 7.7(a) shows an example set of input regions. The polygons do not need to be simple: they can also have holes or multiple components. We show now how to apply the triangulation splitting result to solve our original problem.



(a) (b) (c)

Figure 7.7 (a) A set of regions in the plane. (b) A triangulation of the vertices of the regions. (c) The sample points have been added to the triangulation.

7.3.1 Preprocessing Algorithm

The preprocessing algorithm takes the set of regions $\mathcal{R}$ as input, and must produce a data structure $H(\mathcal{R})$. We use the scaffolding idea, and first define a point set Q to be the set of all vertices of the regions in $\mathcal{R}$. Then we compute a triangulation $S(Q)$, with the restriction that all edges bounding the regions in $\mathcal{R}$ should be in the triangulation. Figure 7.7(b) shows an example. Such a triangulation can be computed in $O(n \log n)$ time.

Apart from this, we compute store some additional information. We create a list of pointers from each region of $\mathcal{R}$ to the set of triangles in $S(Q)$ that cover this region. The structure $H(\mathcal{R})$ now consists of $S(Q)$ together with this list of pointers.

7.3.2 Reconstruction Algorithm

The reconstruction algorithm takes the structure $H(\mathcal{R})$ as input, together with a set of precise points P that contains one point from each region in $\mathcal{R}$, and must produce a triangulation $S(P)$ as output. We will show how to do this in linear time.

We first insert the points of P into the triangulation $S(Q)$ computed in the preprocessing step. For this we need to locate the points in the triangulation, which we do by simply walking through all triangles that this region points to. The point must lie in one of those triangles, and since each triangle is pointed to only once, we spend only linear time in total. Once the right triangle has been found, we simply split it into three smaller triangles. After inserting all points of P in this way, we obtain a triangulation $S(P \cup Q)$. Figure 7.7(c) shows an example.

Finally, we simply invoke Theorem 7.1 to remove the scaffold Q and obtain a triangulation $S(P)$ of the desired points. We can summarise:

Theorem 7.2 *Let $\mathcal{R}$ be a set of n disjoint polygonal regions in the plane. We can preprocess $\mathcal{R}$ in $O(n \log n)$ time into an $O(n)$ size data structure, such that when a point set P*

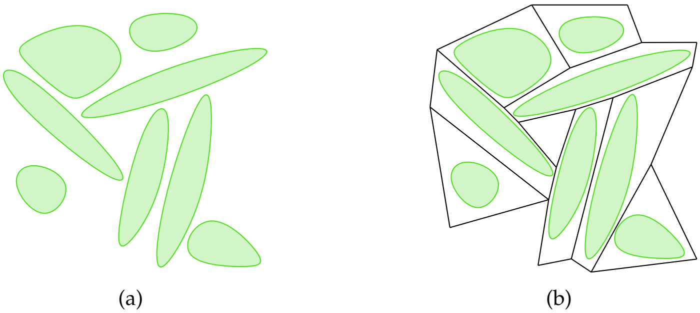


Figure 7.8 If the regions are not polygonal, we can find a polygonal subdivision of the plane such that each cell contains one region.

containing one point from each region in $\mathcal{R}$ *is given, a triangulation of* P *can be computed in* $O(n)$ *time.*

7.4 Extensions

The main improvement of our algorithm over [65] and [86] is that the input regions for the algorithm do not have to be unit disks. In fact, we do not require any fatness or thickness property or bound on the sizes of the regions, and the regions do not have to be convex or connected. However, we did assume that the regions are polygonal and disjoint.

We can extend the approach to also work for regions that are not completely disjoint, as long as the complexity of their overlay is not too high. If we compute the overlay of the polygons and triangulate the resulting arrangement, the method will run in $O(n \log n + m)$ preprocessing and $O(mk \log k)$ reconstruction time, where m is the total complexity of the overlay, and k is the maximum number of regions that overlap in a single point.

If the input regions are not polygonal, we cannot simply triangulate their vertices, but the same approach still works if we first compute a polygonal subdivision of the plane such that each face contains one region. If the regions are convex, a subdivision exists with a complexity that is linear in the number of regions [40], see Figure 7.8. Such a subdivision can be computed in $O(n \log n)$ time [108]. If the regions are not convex, the complexity might increase, depending on the exact shape of the regions: for example, a region with a circular hole and another region which is a disk inside the hole may need an arbitrarily complex polygonal chain to separate.

7.5 Closing Remarks

In this chapter, we presented a pair of algorithms, one for preprocessing a set of disjoint polygonal regions in the plane into a linear size data structure, and one for computing, given this data structure, a triangulation of a set of points that contains one point from each region. The first algorithm runs in $O(n \log n)$ time, the second in $O(n)$ time.

As an important tool to solve this problem, we studied the problem of *splitting* a triangulation. We gave an algorithm that can, in $O(n)$ time, extract a triangulation of a subset of the vertices of a given triangulation.

The results in this chapter also appeared in [127]. The splitting algorithm solves an open problem posed by Chan [24], and the result on preprocessing imprecise points improves an earlier result by Held and Mitchell [65]. The result shows that for any set of disjoint regions, preprocessing for triangulation in the plane is possible. An interesting remaining open question, though, is whether it is also possible to preprocess a set of *lines* in the plane (which are not disjoint) such that a triangulation of one point on each line can be computed faster. Also, the same problem in higher dimensions is wide open. In the next two chapters, we will study under what conditions similar results can be obtained for the *Delaunay* triangulation.

Chapter Eight

Preprocessing for Delaunay Triangulation

Though triangulations in general are an important topic in computational geometry, there is one triangulation of special interest. The so-called *Delaunay triangulation* of a point set P, invented by Boris Delaunay (1890-1980) in 1934 [35], is defined as the triangulation such that if we draw a circle through the corners of any triangle, this circle contains no points of P inside, other than the corners of the triangle themselves. When no four points of P are co-circular, this triangulation is unique.

Since its introduction, it has been proven that the Delaunay triangulation optimises many different criteria, such as the smallest angle of any triangle or the largest enclosing circle of any triangle. An extensive overview of properties can be found in [100]. Because of these properties, in many real-world applications of triangulations the Delaunay triangulation is the standard to be used.

We show in this chapter that we can also preprocess a set of disjoint unit disks in the plane to compute the Delaunay triangulation of any set of points chosen from the disks in linear time. Preprocessing a set of regions for Delaunay triangulation has not been studied before.

As mentioned in the previous chapter, we cannot hope to preprocess a set of overlapping regions for computing even any triangulation faster. In the case of the Delaunay triangulation, we need to restrict the shape even more. For general disjoint regions, the Delaunay triangulation is out of reach: Djidjev and Lingas [36] show that even if the sorted order in any direction of a set of points is given, computing the Delaunay triangulation has a $\Theta(n \log n)$ lower bound. If our set of regions is a set of vertical lines, then all information a preprocessing algorithm could compute is exactly this order (and the distances between the lines, but they can be computed from the order in linear time anyway). As a result of this, we restrict the regions to be disjoint unit

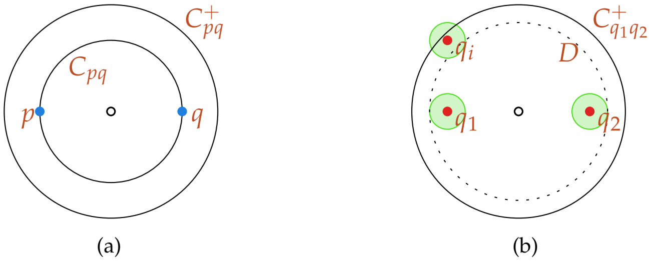


Figure 8.1 (a) The Gabriel circle and expanded Gabriel circle for p and q. (b) The expanded Gabriel circle $C^+_{q_1 q_2}$ contains the centres of any disks R_i with the property that, in the exact point set P, the point p_i can prevent the edge $\overline{q_1 q_2}$ from being Delaunay (i.e., from having an empty circle).

squares in this chapter. Nonetheless, we show in the next chapter how to handle more general regions in a parametrised way.

Let $\mathcal{R}$ be a set of n disjoint unit disks in the plane. We define $Q = \{q_1, q_2, \dots, q_n\}$ to be the centre points of the disks. An exact sample for $\mathcal{R}$ is a set of points $P = \{p_1, p_2, \dots, p_n\}$ drawn one from each disk: i.e., for all $1 \leq i \leq n$, the length $|p_i q_i| < 1$.

We will not strictly follow the scaffolding paradigm in this chapter. Instead, the main idea of our solution is to compute a data structure $H(\mathcal{R})$ that contains enough information to construct a connected subgraph of the Delaunay triangulation of real points. Aggarwal *et al.* [3] gave a clever linear-time algorithm to compute the Voronoi diagram or Delaunay triangulation of points in convex position in the plane. Chin and Wang extended this to compute the constrained Delaunay triangulation of a simple polygon [30]. Rather than define the constrained Delaunay triangulation here, we simply note that if all edges of the simple polygon satisfy the Delaunay empty circle criterion, then the constrained Delaunay triangulation is the same as the Delaunay triangulation. As a simple consequence, the Delaunay triangulation of a point set can be computed using Chin and Wang's algorithm, if a spanning tree consisting of Delaunay edges is already known. The algorithm does require that the polygon is decomposed into trapezoids, which can theoretically be done in linear time by Chazelle's algorithm [25].

8.1 Expanded Gabriel Circles

Before coming to the actual algorithm, we first do some preliminary work. In this section, we define the concept of *expanded Gabriel circles*, and establish several properties about them. We also review some standard graphs and their properties, and

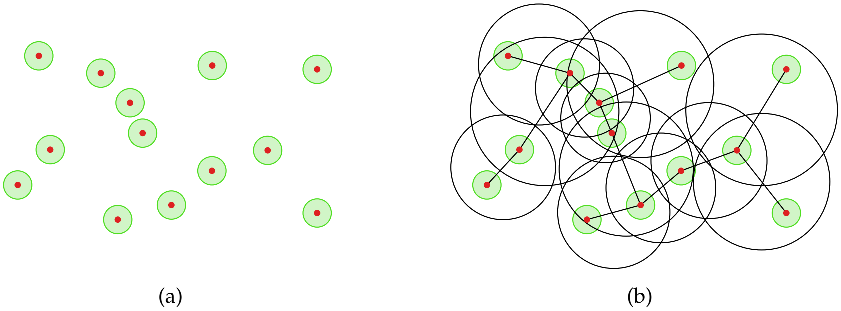


Figure 8.2 (a) A set of imprecise points. (b) The edges of a minimum spanning tree, and the disks that intersect their expanded Gabriel circles.

make several observations about how much these can change when the points they are defined on move slightly.

Gabriel and Sokal [49] define the Gabriel graph for a set of points P. First, for two points p and q, let C_{pq} denote the circle with diameter $\overline{pq}$, as in Figure 8.1(a). Points $p, q \in P$ are joined by edge $\overline{pq}$ if and only if the circumscribing circle C_{pq} contains none of the other points of P. It is well known, and obvious from this definition, that the Gabriel graph is a subgraph of the Delaunay triangulation.

We define the expanded Gabriel circle, C^+_{pq}, as the circle with centre $(p+q)/2$ and radius $|pq|/2+2$. Figure 8.1(a) shows an example. In the context of preprocessing imprecise points, the expanded Gabriel circle contains exactly the centres of those disks that could, in the exact point set, prevent pq from being a Delaunay edge.

Observation 8.1 *For disk centres $q_1, q_2 \in Q$, if no other point $q_i \in Q$ lies in the expanded Gabriel circle $C^+_{q_1q_2}$, then in any exact point set P, the edge $\overline{p_1p_2}$ is Delaunay in P.*

Proof Consider the smallest circle D enclosing the unit disks centred at q_1 and q_2; specifically, the circle D centred at the midpoint $(q_1+q_2)/2$ with radius $|q_1q_2|/2+1$, whose boundary is drawn dotted in Figure 8.1(b). There is another circle inside D that has the samples p_1 and p_2 on its boundary: shrink D with respect to its centre until the first point, say p_1, lies on the boundary, then continue to shrink with respect to p_1 until p_2 is also on the boundary. An exact point p_i can lie inside D only if the corresponding unit disk centre satisfies $q_i \in C^+_{q_1q_2}$. ■

8.1.1 Expanded Gabriel Circles of EMST Edges

A *Euclidean minimum spanning tree (EMST)* of a point set T is a tree spanning all points in P of minimum total length. It is well-known that when removing a tree edge $\overline{uv}$ from the EMST, the tree is partitioned into two connected components such that no

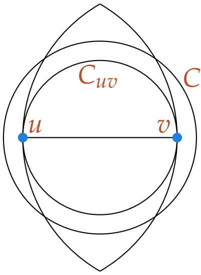


Figure 8.3 The empty lune for EMST edge $\overline{uv}$.

vertex in the component of u can lie strictly inside the circle of radius $|uv|$ around v, and vice versa. This implies that the interior of the lune that is the intersection of both circles is empty. This lune completely contains C_{uv} in its interior except for u and v, so all other points of P must lie outside C_{uv}. Therefore, an EMST for P is a subgraph of the Gabriel graph of P. One other consequence of this fact is that any EMST has maximum vertex degree 6.

Let $T = (Q, E)$ be the Euclidean minimum spanning tree (EMST) of Q. We now forget about the application for a while, and show that each point in the plane (and therefore also the sites of Q) can lie in at most a constant number of the expanded Gabriel circles defined by the edges in E. We use this in later sections to bound the amount of repair work necessary to find a spanning tree of Delaunay edges for a particular sample from the unit disks centred at Q. An example of a minimum spanning tree and the expanded Gabriel circles of its edges is depicted in Figure 8.2.

We do an initial partitioning of spanning tree edges into *long* and *short*, depending on whether an edge's length is greater than, or at most, $L = 2 + 2\sqrt{3} \approx 5.464$. This threshold value is chosen so that we can identify the connected components of the EMST when a long edge is removed.

Lemma 8.1 *Let $\overline{uv}$ be a long edge of the EMST of Q. Any point $w \in Q \cap \operatorname{int}(C^+_{uv})$ for which $|uw| \leq |vw|$ satisfies $|uw| < |uv| \leq |vw|$, and w and u belong to the same component of the EMST after removing $\overline{uv}$.*

Proof Recall that when $\overline{uv}$ is an edge of the Euclidean minimum spanning tree, the lune that is enclosed by the circles of radius $|uv|$ centred at u and at v has no sites in its interior. When $|uv| > L$, this lune pokes outside the expanded Gabriel circle C^+_{uv}, as in Figure 8.3. Since the portion of the perpendicular bisector of $\overline{uv}$ inside the lune cuts the circle C^+_{uv}, we can partition $Q \cap C^+_{uv}$ into the sets U and V, closer to u and v, with no ambiguity.

The distance from u to $w \in U$ is maximised if w is at the intersection of the lune boundary with C^+_{uv}. If we let $\ell = |uv|/2$, then because $\ell > L/2$ we know that $\ell + 2 < \ell\sqrt{3}$, and the triangle $\triangle uvw$ cannot be equilateral, but must have $|uw| < |uv|$.

Now, w and u must belong to the same component of the EMST after removing $\overline{uv}$, since otherwise $\overline{uw}$ would have been a better edge than $\overline{uv}$ to connect the two components. ■

For a given point p in the plane (possibly, but not necessarily, a site from Q), let E_p denote the set of edges of the EMST whose expanded Gabriel circles enclose p, that is, $E_p = \{\overline{uv} \in E \mid p \text{ inside } C^+_{uv}\}$. We partition E_p into two groups: the *near* edges, for which both endpoints are at most $L + 2$ away from p, and the *far* edges, for which at least one endpoint is $L + 2$ or more away from p. Note that every far edge must necessarily be long, and that a near edge can be either short or long. We separately bound the numbers of near edges and far edges in E_p.

An easy packing argument bounds the set of near edges for p, which includes all short EMST edges.

Lemma 8.2 *For any point p in the plane E_p contains at most 70 near edges; i.e., p is in at most 70 expanded Gabriel circles of the edges of the EMST of Q that have both endpoints within distance $L + 2$ of p.*

Proof If a centre from Q is within $L + 2$ of p, the corresponding disk from $\mathcal{R}$ is completely within $L + 3$. At most $\lfloor (L + 3)^2 \rfloor = 71$ unit disks from $\mathcal{R}$ can fit into this area, inducing at most 70[1] edges of the minimum spanning tree. ■

An angle packing argument in the next lemma shows that an input point $q \in Q$ has few far edges. In Lemma 8.4, we will show that this also holds for points $p \notin Q$, but then the constant is worse.

Lemma 8.3 *For any point $q \in Q$, E_q contains at most 8 far edges.*

Proof We consider far edges $F \subset E_q$ in order of decreasing length, removing them from the EMST of Q, and keeping track of the connected component containing q. We assume, without loss of generality, that each far edge is labelled so that the first endpoint is the one closer to q; e.g., for $\overline{uv}$, we have $|qu| \leq |qv|$.

Let T be the current EMST component, which is partitioned into $\{T_u, \overline{uv}, T_v\}$ by removing $\overline{uv}$, the longest edge of $F \cap T$. By Lemma 8.1, we know that q remains in the component of u, namely T_u, and that $|qu| < |uv| \leq |qv|$. We claim that all other edges of F in T belong to T_u: consider another edge $\overline{u'v'} \in F \cap T$, as illustrated in Figure 8.4(a). Since $\overline{u'v'}$ is long, Lemma 8.1 gives $|qu'| < |u'v'| \leq |qv'|$, and ordering by length gives $|u'v'| \leq |uv|$. But $\overline{uv}$ was chosen as the EMST edge joining T_u and T_v, and the shorter edge $\overline{qu'}$ was not; therefore u' must be in T_u with q, and v' too since $\overline{u'v'}$ is an edge of T.

Next, we show that v indicates a sector of the plane as seen from q that contains no other second endpoints of edges of F; that is, no other *far vertices* of F. By definition,

---

[1] The constant of 70 is rather pessimistic. The best penny packing known for a circle of radius $L + 3$ has only 57 disks [56, 122], and even then it seems hard to draw many spanning tree edges between them that actually have p in their expanded Gabriel circle.

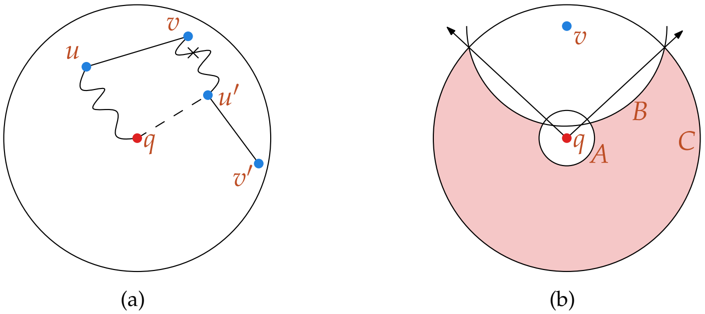


Figure 8.4 Illustrating arguments used to show there are few *far* edges in Lemma 8.3. (a) Removing a long edge $\overline{uv}$ cannot disconnect another long edge from q, since EMST edge $\overline{uv}$ is longer than $\overline{qu'}$. (b) Using definitions in the text, all far vertices of F lie inside circle C and outside circles A and B. This results in an empty sector of angle at least $2\pi/9$ viewed from q.

we know that the circle A of radius $L+2$ around q contains no such vertices, and by the previous paragraph we know that the circle B of radius $|uv|$ around v contains no such vertices, since otherwise there would be a shorter possible edge than $\overline{uv}$ to connect T_u and T_v. Now consider the farthest vertex v' among all vertices in F, so all remaining vertices are inside a circle C of radius $|qv'|$ around q. This vertex must be part of an edge $\overline{u'v'}$ of length at least $|qv'|-2$, otherwise it would not be in E_q. Therefore, also $|uv| \geq |u'v'| \geq |qv'|-2$. Now, all remaining far vertices of F must be in the region $C \backslash (A \cup B)$, see Figure 8.4(b).

To define the free sector, consider the angle that $\overline{qv}$ makes with the intersections between A and B, and the angle it makes with the intersections between B and C. The smaller of those two angles bounds the sector.

Thus, we consider the triangles of side lengths $L+2$, $|qv|$, and $|uv|$ and of $|qv|$, $|qv|$, and $|uv|$. We know that $|qv| < |uv|+2$ and $|uv| > L$. The angle at q is minimised as $|qv|$ approaches $L+2$, which would give, in both cases, the isosceles triangle with angle

$$2 \arcsin\left(\frac{L/2}{L+2}\right) > 0.7494 > 2\pi/9.$$

Thus, inside the empty circle around v we find two sectors of angle $> \pi/4$ on either side of $\overline{qv}$ that contain no far points closer to q than v. At most two empty sectors can overlap—one from the clockwise (CW) and one from the counter-clockwise (CCW) direction around q, which implies that there are at most 8 far edges. ■

Using Lemmas 8.2 and 8.3, we can summarise:

Theorem 8.1 *Let $T = (Q, E)$ be the Euclidean minimum spanning tree of the points Q. The total number of these points in the expanded circles for all edges is linear in n. That is,*

$$\sum_{uv \in E} |C^{+}_{uv} \cap Q| = O(n).$$

We can in fact extend the proof of Lemma 8.3 to bound the number of far edges for an arbitrary point p in the plane, albeit with a large (and overly-pessimistic) constant factor. Since this bound is used only to shorten the description of preprocessing, and not for the algorithm itself, we have not tried to minimise the constant. The next lemma implies that the arrangement of all expanded Gabriel circles has linear complexity.

Lemma 8.4 *For any point p in the plane $|E_p|$ is constant.*

Proof The disk packing argument in Lemma 8.2 shows that there are at most 71 disk centres within distance $L + 2$ of any point p. As these are vertices in a Euclidean minimum spanning tree (EMST), for which each vertex has degree at most 6, at most 426 edges of E_p can have a vertex within $L + 2$ of p.

We therefore consider only the subset $F \in E_p$ of far edges for p that have both endpoints farther than $L + 2$ from p. We show that the edges of F can be organised into a binary tree whose maximum depth is 8 by the angle packing argument used in Lemma 8.3. Since such a binary tree has at most $2^9 - 1 = 511$ nodes, F has at most $511 + 426 = 937$ edges.

We build this tree from the root at depth 0. Each node ν is associated with a subset of edges, $F_\nu \subset F$, as well as an edge of F_ν. The root is associated with F, and some edge $\overline{uv}$ where v is the point farthest from p. Removing $\overline{uv}$ from the EMST partitions the remaining edges of F into two groups, F_u and F_v, such that the edges in F_u are in the same component as u of the EMST after removing $\overline{uv}$, and the edges in F_v are in the same component as v. By Lemma 8.1, the first remain connected to u and the second remain connected to v. These are the edge sets associated with the children of the root. (In determining connectedness, we include EMST edges and vertices within $L + 2$ of p, even though they are not in $F \setminus \{\overline{uv}\}$.)

In general, at node ν, the edges F_ν are edges of a connected component of the EMST minus the edges associated with the ancestors of ν, the associated edge $\overline{uv} \in F_\nu$ is chosen so that the endpoint v is farthest from p, and removing $\overline{uv}$ partitions the edges of the EMST component into F_u and F_v.

For the edges in F_u, we know that no endpoint can lie within a circle of radius $|uv|$ centred at v. We also know that all endpoints lie within a circle of radius $|pv|$ centred at p, and that none lie within $L + 2$ of p. These constraints on F_u are depicted in Figure 8.5(a). As in the proof of Lemma 8.3, there is a sector with angle greater than $2\pi/9$, as seen from p, that contains no endpoints from F_u.

The other endpoint of $\overline{uv}$ can lie closer to p, as shown in Figure 8.5(b). Still, in this case there is also a sector with an angle of at least $2\pi/9$ that contains no endpoints

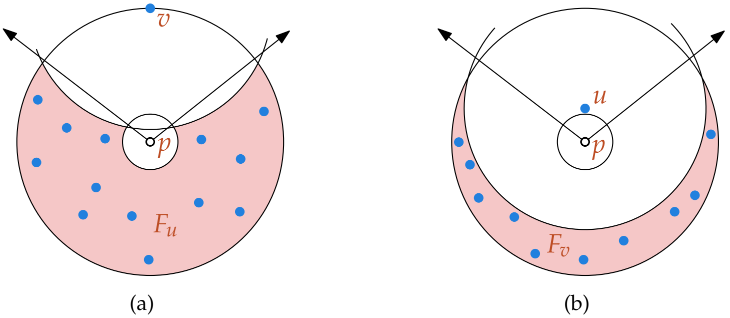


Figure 8.5 (a) The point in the plane p and the furthest endpoint v ensure that all endpoints of F_u lie within the shaded area. (b) The point u can be closer to p, but there is a circle with radius $|uv|$ around it that contains no endpoints of F_v.

of F_v. The point u must still be outside the circle of radius $L + 2$ around p, and the circle of radius $|uv|$ around u intersects the circles around p of radii $L + 2$ and $|pv|$ according to the same restrictions as in the case of v.

This implies that the tree has depth at most 8, and completes the proof of the lemma. ■

8.2 Delaunay Triangulation of Imprecise Points

We are now ready to solve the following problem:

Problem 8.1 *Given a set of disjoint unit disks in the plane, preprocess them in such a way that when a point in each region is given, a Delaunay triangulation of these points can be computed faster than without preprocessing.*

We present a pair of algorithms, one for preprocessing and one for reconstruction, to solve the Delaunay triangulation problem on imprecise points.

Let $\mathcal{R}$ be a set of n disjoint unit disks in the plane that represent the imprecise regions for an unknown point set P. Subsection 8.2.1 details how to preprocess $\mathcal{R}$ in $O(n \log n)$ time into a linear-size data structure $H(\mathcal{R})$. Subsection 8.2.2 shows that given an exact point set P consisting of a point inside each disk of $\mathcal{R}$, we can compute the Delaunay triangulation of P in linear time using $H(\mathcal{R})$.

8.2.1 Preprocessing Algorithm

Recall that Q is the set of centre points of the n disjoint unit disks of $\mathcal{R}$. For $H(\mathcal{R})$, we compute a Euclidean minimum spanning tree of Q, a list of its edges sorted by length, and for each edge $\overline{uv}$ the list of points of Q that fall inside the expanded Gabriel circle C^+_{uv}. Figure 8.2 shows an example. By Theorem 8.1 we know that each point of Q can fall into at most a constant number of expanded Gabriel circles. Thus, the total size of $H(\mathcal{R})$ is linear.

Also, $H(\mathcal{R})$ can be computed in $O(n \log n)$ time. A minimum spanning tree is easy to compute in $O(n \log n)$ time, since it is a subset of the Delaunay triangulation. Sorting the list of EMST edges by length is even easier. Finally, a simple sweep of the arrangement of the expanded Gabriel circles of the EMST edges and the points P can locate all points in their circles; because Lemma 8.4 states that this arrangement has linear size, the sweep can be carried out in $O(n \log n)$ time.

Lemma 8.5 *Preprocessing the n disjoint unit disks $\mathcal{R}$ produces a linear size data structure $H(\mathcal{R})$ in $O(n \log n)$ time.*

Denote the list of EMST edges, sorted by increasing length, by $e_1, \dots e_{n-1}$. We define notation for the connected components of the graph consisting of the first k edges of this list: Let $\mathcal{I}_k$ be the partition of the index set $\{1, \dots, n\}$ induced by the connected components of these first k edges: that is, $i, j \in I$ for some $I \in \mathcal{I}_k$ if and only if q_i and q_j can be joined by edges from $\{e_1, \dots, e_k\}$. We can associate these connected components with $H(\mathcal{R})$ (conceptually, not computationally, as they are needed only for a proof), because our algorithm creates the components (or supersets of them) for points $P = \{p_1, \dots, p_n\}$ drawn from each disk in $\mathcal{R}$.

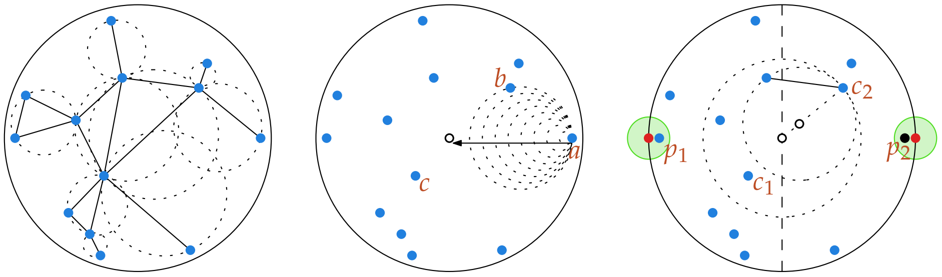
8.2.2 Reconstruction Algorithm

Now, given an exact point set $P = \{p_1, \dots, p_n\}$ of $\mathcal{R}$, and the data structure $H(\mathcal{R})$, we show how to compute in linear time a connected subgraph of the Delaunay triangulation of P. Chin and Wang's algorithm [30] then completes the Delaunay triangulation of P in linear time.

In order to construct such a connected subgraph, we process the edges of the EMST of Q by increasing length. For each such edge e, we find a path in the Delaunay triangulation that connects the same components that e connects in the graph composed of all EMST edges shorter than e. We begin by making an observation, illustrated in Figure 8.6(a), on the portion of a Delaunay triangulation bounded by a circle.

Lemma 8.6 *Let P be a set of points in general position in the plane, C be a circle that encloses a subset $P' = P \cap \mathrm{int}(C)$, and E' be the set of Delaunay edges of P that have empty circles contained inside $\mathrm{int}(C)$. The graph (P', E') is connected.*

Proof Let c be the point of P closest to the centre of C; we show that any point $p \in P'$ is connected to c. Initially, let $a = p$, and, as depicted in Figure 8.6(b), grow a circle from a towards the centre of C, keeping a pinned on the boundary; stop when the circle hits any point $b \in P'$. The edge $\overline{ab}$ is discovered to be a Delaunay edge in E', and the point b is closer to the centre of C than a was. Since P is finite, by setting



(a) (b) (c)

Figure 8.6 (a) The Delaunay edges certified by (dotted) empty circles within a bigger circle form a connected graph. (b) Growing a circle from p towards the centre. (c) The closest point to the centre can be connected to at least one of the points in the other group.

$a = b$ and repeating this procedure, we eventually construct a path from p to c in the graph (P', E'). ■

Suppose now that EMST edge e_k joins $q_1, q_2 \in Q$, and consider the expanded Gabriel circle $C^+_{q_1q_2}$. Lemma 8.6 states that there exists a path of Delaunay edges certified inside $C^+_{q_1q_2}$ that joins the corresponding exact points $p_1, p_2 \in P$; our task is to compute one efficiently, or at least to compute a subgraph of the Delaunay triangulation of P that contains one or more paths.

To reconstruct the Delaunay triangulation, we first want to build up the components of the EMST by adding edges in order; the essential task is to find a path of Delaunay edges joining the exact points $p_1, p_2 \in P$ for two centres $q_1, q_2 \in Q$ that form an edge e_k in the EMST. We will do this inside $C^+_{q_1q_2}$, although we could do it in the smaller $C_{p_1p_2}$ with a slightly longer description of the procedure.

Lemma 8.7 *Let $\overline{q_1q_2}$ be an edge of the EMST of Q, and let $P' \subset P$ denote the set of precise points inside circle $C^+_{q_1q_2}$. Assuming we already connected all EMST edges shorter than $\overline{q_1q_2}$, we can connect q_1 and q_2 in $O(|P'|)$ time.*

Proof When $\overline{q_1q_2}$ is short, penny packing says there are at most a constant number of disks in $C^+_{q_1q_2}$, so we can process $\overline{q_1q_2}$ by computing the Delaunay triangulation of the points P' and discarding edges that are not certified by an empty circle inside $C^+_{q_1q_2}$.

When e_k is long, by Lemma 8.1 there are two components that are separated by the perpendicular bisector of $\overline{q_1q_2}$. Let P'_1 and P'_2 be the partition of P' by this bisector. It suffices to find a Delaunay edge of P from $P'_1 \times P'_2$ since the points within P'_1 (and P'_2) have already been connected earlier in the algorithm.

Let $c_1 \in P'_1$ and $c_2 \in P'_2$ be the closest points to the centre of $C^+_{q_1q_2}$, as illustrated in Figure 8.6(c), and assume that the distance to c_2 is greater, meaning the circle concentric with $C^+_{q_1 q_2}$ through c_2 contains at least one point of P'_1. Shrink this circle with c_2 on the boundary by moving its centre towards c_2 until the last point of P'_1 leaves its interior—this point defines the desired Delaunay edge with c_2. Both steps can be carried out in time proportional to $|P'|$. ■

We spend constant time on each short edge and, by Theorem 8.1, a total of linear time on the long edges. For each edge we find a path of Delaunay edges of P that joins the vertices p_1 and p_2, so the connected components induced by the sequence of edges found will be supersets of the components of $\mathcal{I}_k$ of the first k edges of the EMST of P. Thus, we obtain a connected graph after processing all EMST edges, and can invoke the algorithm by Chin and Wang [30] to complete the Delaunay triangulation.

Theorem 8.2 *Let $\mathcal{R}$ be a set of n disjoint unit disks in the plane. We can preprocess $\mathcal{R}$ in $O(n \log n)$ time into an $O(n)$ size data structure, such that when a point set P containing one point from each region in $\mathcal{R}$ is given, a Delaunay triangulation of P can be computed in $O(n)$ time.*

8.3 Extensions

Our algorithm works for a very specific class of imprecise regions: disjoint disks of equal radius. In practice, this may be a rather strong assumption on the input, especially disjointness is not a very natural property of imprecise point sets. In this section, we show how the result can be extended to less restricted regions.

8.3.1 Overlapping Disks

If we allow the regions to be arbitrarily overlapping disks, then there is little we can hope to prove. In the worst case, all disks could coincide, allowing the constructions that establish the $\Omega(n \log n)$ lower bounds for general Delaunay triangulation [118]. If we limit the depth of overlap, however, our result still holds with the algorithm unchanged.

We say a set of disks is *k-overlapping* if no point in the plane is contained in more than k disks. In this case, the number of short edges that can contain a point p increases. Clearly, there cannot be more than $k(r+2)^2$ disks touching a circle of radius r. This means the constant grows linearly in k. The arguments involving long edges do not depend on the disjointness of the disks.

8.3.2 Other Extensions

If we allow the disks to have different radii, then in general the problem is open. However, when there is a constant fraction $\rho = \frac{r^+}{r^-}$ between the largest radius r^+ and the smallest radius r^-, then we can just increase the radii of the disks such that all disks have radius r^+. Since we know that the sample points lie inside the input disks, they certainly also lie inside the grown disks. Of course the disks start overlapping, but not too much: at most $(\rho + 1)^2$ grown disks contain any given point in the plane.

If the input regions are not disks but squares, then we can grow them to the smallest disks containing them, which are 3-overlapping. If the regions are β*-thick* in the sense that they contain circles of radius r^- but are contained in circles of radius r^+ (the same radii for all regions), with $\beta = \frac{r^+}{r^-}$, then we can again replace them by disks of radius r^+ that are at most $(\beta + 1)^2$-overlapping.

Finally, we can also handle combinations of the above. If we have a set of k-overlapping β-thick regions with radii (say, of their smallest enclosing circles) that differ by at most a factor ρ, then we can translate this into a set of $O(\rho^2\beta^2k)$-overlapping unit disks. These can then be preprocessed in $O(\rho^2\beta^2kn\log n)$ time into a $O(\rho^2\beta^2kn)$ size data structure, such that the Delaunay triangulation of the real points can be computed in $O(\rho^2\beta^2kn)$ time once the points are known. In particular, this means that when these parameters are constants, the same result as for unit disks still holds.

8.4 Closing Remarks

In this chapter, we studied the problem of preprocessing a set of disks to speed up the computation of the Delaunay triangulation. We showed that a set of disjoint unit disks can be preprocessed in $O(n\log n)$ time, such that the Delaunay triangulation of a precise set of points can be computed in linear time once these points are given.

The results in this chapter appeared in [86]. Though they show that preprocessing disks for Delaunay computation is theoretically possible, the reconstruction algorithm does rely on the linear-time polygon triangulation by Chazelle [25], which is rather complicated to implement. Furthermore, though the algorithm generalises to less restrictive regions than unit disks, as discussed in the previous section, the dependence on the parameters is not optimal. In the next chapter, we will discuss a different, more practical approach to solving the problem.

Chapter Nine

Preprocessing More General Regions for Delaunay Triangulation

In this chapter, we will show how to obtain the same result as in the previous chapter. The difference is that the reconstruction algorithm in this case is randomised, and therefore not guaranteed to always finish within a linear number of steps (though with high probability, it will). On the other hand, the data structure in this chapter is more powerful, and will allow us to handle more general input regions than just unit disks in an optimal way.

We consider the same parameters as in the previous chapter. We consider ρ, the fraction between the radii of the smallest enclosing circles of the largest and smallest regions in $\mathcal{R}$; k, the largest depth in the arrangement of $\mathcal{R}$; and β, the largest thickness of any region.

We show how to preprocess a set of k-overlapping disks in $O(n \log n)$ time such that the Delaunay triangulation of the real points can be computed in $O(n \log k)$ expected time, independent of ρ. Furthermore, we show how to preprocess a set of k-overlapping β-thick regions in $O(n \log n)$ time such that the Delaunay reconstruction algorithm takes $O(n(\log(k\beta) + \log\log \rho))$ expected time.

The approach in this chapter is similar to the one in Chapter 7. We first build a *scaffold*, to which we can easily add the precise points when they become available, and then remove the scaffold using a splitting algorithm for Delaunay triangulations. We use the following theorem by Chazelle *et al.*

Theorem 9.1 (Chazelle *et al.* [26]) *Let $P, Q \subseteq \mathbb{R}^2$ be two planar point sets, with total complexity m. Suppose that the Delaunay triangulation of $P \cup Q$ is available. Then the Delaunay triangulation of P can be computed in $O(m)$ expected time.*

9.1 Disks of Different Sizes

In this section we study the following problem:

Problem 9.1 *Given a set of disjoint disks in the plane, preprocess them in such a way that when a point in each region is given, a Delaunay triangulation of these points can be computed faster than without preprocessing.*

In the previous chapter, we needed to assume that all disks in $\mathcal{R}$ had the same size. We now describe an approach that does not have this restriction, and can still preprocess $\mathcal{R}$ in $O(n \log n)$ time and reconstruct the Delaunay triangulation in $O(n)$ expected time.

9.1.1 Quadtrees

The approach in this chapter is based on *quadtrees*. We first discuss what these are and present some extension. Quadtrees are a popular way to create a hierarchical decomposition of space that allows for variable sizes of cells [46]. They are used often in practice due to their simple structure. However, for our purposes the standard definition of a quadtree is too restrictive. Instead, we will define the concept of a *free quadtree*, and then restrict this based on our needs in the various sections in this chapter.

Definition A *free quadtree* T is a rooted tree that corresponds to a hierarchical decomposition of a square region B with side length ℓ in the plane. Each node v of T has an axis-aligned square box B_v associated to it. Furthermore, the following requirements should be met:

1. The root of T has B itself associated to it.
2. If w is a descendent of v in T, then B_w is contained in B_v.
3. If v and w are siblings in T, then B_v is disjoint from B_w.
4. Every node v of T has a box B_v with side length $2^{-k}\ell$ for some positive integer k, and the x- and y-coordinates of its corners are multiples of its side length away from the boundary of the root box.[1]

Such a tree T decomposes the root box into a number of cells. We denote by T_v the subtree of T rooted at v. For each node v, we define $C_v = B_v \setminus \bigcup_{w \in T_v, w \neq v} B_w$ as the cell that consists of the part of B_v that is not covered by the children of v. Note that C_v can be empty, or not connected. Now, each pair of cells C_v and C_w is disjoint, and the union of the cells of all nodes of T covers the root square. We also say the *size* of a node is the side length of its box.

---

[1] In order to be able to ensure this property, we assume that the floor operation can be executed in constant time. In [19], we show that this is in fact not a necessary assumption, but it simplifies the presentation.

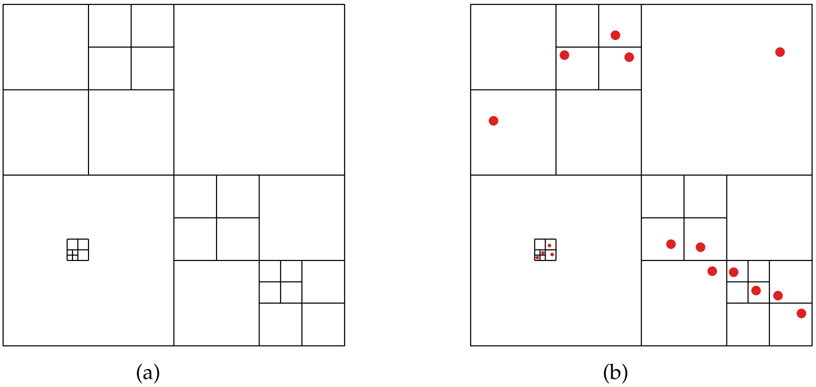
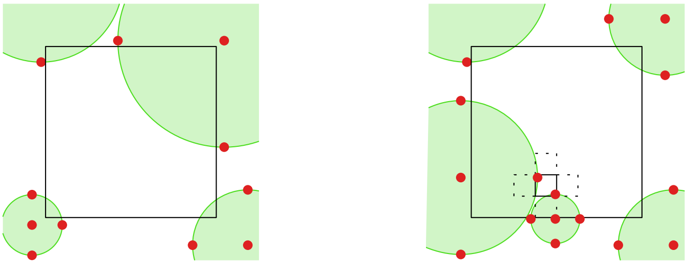

(a) (b)

Figure 9.1 (a) A quadtree. The lower left box contains a cluster node. (b) The quadtree is a valid quadtree for this set of points.

Quadtrees are usually used to capture the differences in density of a geometric scene. In the simplest case, this scene is a set of points. We say that a quadtree T is *valid* for a set of points P if the following properties hold:

1. The root box of T contains all points of P.
2. Each leaf v of T contains at most one point of P inside C_v.
3. Each internal node v of T contains no points of P inside C_v.
4. The complexity of T is $O(|P|)$.

Though it is easy to construct a valid free quadtree for a given point set, we need some more structure to make them useful. We will consider only restrictions of free quadtrees in the rest of this chapter. A standard quadtree as used in the literature is a free quadtree under our definition, and in particular has only two types of nodes: internal nodes v with exactly four children that are half as large as v, and leaf nodes that have no children at all. This definition is sufficient for most practical purposes, but theoretically it is possible that the difference in density of different parts of a point set is arbitrarily large, and to represent such a set with a quadtree would require an unbounded number of boxes. Therefore, Bern *et al.* [14] augment this definition by allowing another type of node: a *cluster node* v is a node with just a single child w, such that the fraction between the sizes of v and w is at least a constant factor 2^c, where c is a large enough integer. With this addition, they prove that for any point set P there exists a quadtree of linear size. Figure 9.1(a) shows an example of a quadtree of this type, and Figure 9.1(b) shows a set of points for which this quadtree is valid.



(a) (b)

Figure 9.2 (a) At most four disjoint disks can intersect any given box B belonging to a leaf of the quadtree, without one of their points being inside B. (b) For a box B belonging to a parent of a cluster node with box C, slightly more disks can intersect the interior of $B \setminus C$, but not more than four can cover any of the four neighbouring boxes, so a crude upper bound is 20.

9.1.2 Preprocessing Algorithm

We now show how to preprocess a set of disjoint disks of arbitrary sizes. We derive a scaffold point set Q from $\mathcal{R}$, and compute a valid quadtree T for Q. Then, once we get a precise point set P, we insert the points of P into the quadtree as well, and compute the Delaunay triangulation of $P \cup Q$. We finally split it using Theorem 9.1 to obtain the Delaunay triangulation of P.

We construct Q by taking from every disk in $\mathcal{R}$ a set of five points representing it, namely the centre point and the top-, bottom-, left- and rightmost points. Then, T can be constructed in $O(n \log n)$ time [14]. We will make an observation about how the boxes of T intersect the regions of $\mathcal{R}$, which will allow us to insert the points of P into Q efficiently later.

Lemma 9.1 *For every node v of T, C_v is intersected by $O(1)$ disks in $\mathcal{R}$.*

Proof There are three types of nodes to consider. First, if v is a normal internal node with four children, then C_v is empty so the condition is trivially true.

Next, suppose that v is a leaf node, so $C_v = B_v$. If a disk D intersects B_v, then either D contains a corner of B_v or B_v contains D's centre or one of its four extreme points [33]. Since the disks in $\mathcal{R}$ are disjoint, there can be at most four disks containing a corner of B_v, see Figure 9.2(a) for an example. And since leaf nodes contain at most one point of Q in their cells, there can be at most one disk with a reference point inside B_v. Thus, B_v intersects at most five disks.

Finally, suppose v is a cluster node, with child w. Then $C_v = B_v \setminus B_w$. We know there are no points of Q in B_v outside B_w, and there are at most four disks that can contain a corner of B_v. However, there can also be disks that intersect B_v and do not cover a corner of B_v, if they have at least one representative point in B_w. To bound the number of these, consider the four squares adjacent to B_w, that is, above, below, to the left and to the right of it. Because of the alignment property (requirement 4) of free quadtrees, these boxes are either completely contained in B_v, or completely outside it. Using the same argument as above, each of these neighbours that is inside B_v is intersected by at most four disks, and every disk D with a representative point in B_w that intersects C_v also intersects one of these orthogonal neighbours: if D has no extreme point or centre in an orthogonal neighbour and does not cover any of its corners, it has to cover its centre. Figure 9.2(b) shows an example involving a cluster node. ■

To finish the preprocessing, we store for each $R_i \in \mathcal{R}$ a list with the cells of the subdivision of T that intersect it. By Lemma 9.1, the total size of these lists, and hence the complexity of the data structure, is linear.

9.1.3 Reconstruction Algorithm

Once we receive the real point set P, we compute the Delaunay triangulation of P by first inserting P into T. For each $p_i \in P$, we find the node v of T that contains p_i in C_v by traversing the list stored with R_i. This takes linear time in total. Since each cell of T contains constantly many input points, we can turn T into a quadtree for $P \cup Q$ in linear time.

Next, we show how to compute the Delaunay triangulation of a set of points when given a quadtree of them. The next lemma is a variant of a theorem by Bern *et al.* [14], to which we refer the reader for further details (see also [15]).

Lemma 9.2 *Let $P \subseteq \mathbb{R}^2$ be a set of n points in the plane, and let T be a quadtree for P. Then, given P and T, we can find the Delaunay triangulation of P in $O(n)$ expected time.*

Proof First, we extend T into a quadtree T' that is balanced and separated. We say a tree T' is *balanced* if no leaf in T' shares an edge with a leaf whose size differs by more than a factor of two, and we say that T' is *separated* if each non-empty leaf of T' is surrounded by two layers of empty boxes of the same size. This can be done by a top-down traversal of T, adding additional boxes for the balance condition and by subdividing the non-empty leaves of T to ensure separation. If after C subdivision steps a non-empty leaf B still does not satisfy separation, we place a small box around the point in B and treat it as a cluster, for which separation obviously holds.

Given T', we obtain a non-obtuse Steiner triangulation $\mathcal{T}$ for P with $O(n)$ additional vertices P' through a sequence of local manipulations, as described by Bern *et al.* [14]. Since all these operations involve constantly many adjacent boxes, the total time for this step is linear. It is a well-known fact that any triangulation without obtuse angles is a Delaunay triangulation, therefore, $\mathcal{T}$ is the Delaunay triangulation of $P \cup P'$, and we can use Theorem 9.1 to extract the Delaunay triangulation of P in $O(n)$ expected time. ■

Now, we apply Lemma 9.2 to compute the Delaunay triangulation of $P \cup Q$ from T, and finally we obtain the Delaunay triangulation of P by using Theorem 9.1 again. We summarise:

Theorem 9.2 *Let $\mathcal{R}$ be a set of n disjoint disks in the plane. We can preprocess $\mathcal{R}$ in $O(n \log n)$ time into an $O(n)$ size data structure, such that when a point set P containing one point from each region in $\mathcal{R}$ is given, a Delaunay triangulation of P can be computed in $O(n)$ expected time.*

9.2 Overlapping Disks

In this section, we study a more relaxed version of Problem 9.1 by dropping the disjointness condition:

Problem 9.2 *Given a set of k-overlapping disks in the plane, preprocess them in such a way that when a point in each region is given, a Delaunay triangulation of these points can be computed faster than without preprocessing.*

We now show how the approach can be extended to disks with limited overlap. Now $\mathcal{R}$ contains n disks in the plane such that no point is covered by more than k disks. Aronov and Har-Peled [7] show that for a given set of regions, the value of k can be approximated up to a constant factor in $O(n \log n)$ time. It is easily seen that reconstructing the Delaunay triangulation after any preprocessing takes $\Omega(n \log k)$ time in the worst case, and we show that this bound can be achieved.

The general strategy is the same as in Section 9.1. Let Q be the $5n$ representative points for $\mathcal{R}$, and let T be a quadtree for Q. As before, T can be found in time $O(n \log n)$ and has complexity $O(n)$. However, the cells of T can now be intersected by $O(k)$ regions, rather than $O(1)$. This means the reconstruction will be slower, but we can optimise the running time by reducing the complexity of T until its precision somehow matches the number of regions that can intersect a box. For this we introduce the notion of λ-*deflated* quadtrees.

9.2.1 Deflated Quadtrees

For an integer $\lambda > 0$, we define a λ-deflated quadtree T' for a point set P to be a quadtree that may contain up to λ points of P in its cells, and therefore can be shown to have $O(n/\lambda)$ nodes. A deflated quadtree is another, weaker restriction of a free quadtree. We distinguish four different types of nodes.

- A *leaf* is a node without children. A leaf may contain up to λ points.
- A *regular internal node* has four children that cover their parent. There can be no points directly associated with a regular internal node.

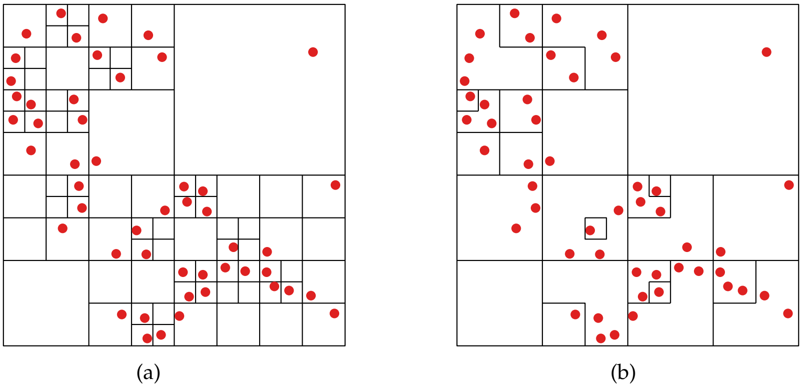

(a)
(b)

Figure 9.3 (a) A set of points and a quadtree for it. (b) A 3-deflated version of the quadtree.

- A *cluster node* is, as before, a node with a single, much smaller child. A cluster node also cannot have any associated points.
- Finally, a *deflated node* is also a node with a single child, which is possibly but not necessarily much smaller than its parent. A deflated node may contain up to λ points in its cell.

Figure 9.3 show a quadtree and a 3-deflated version of it.

Given a quadtree T for P, a λ-deflated quadtree T' can be found in linear time by processing the nodes from top to bottom. For every node v in T, compute $n_v = |B_v \cap P|$. This takes $O(n)$ time. Then, recursively for each node v, if $n_v \leq \lambda$ we replace it with a single leaf. Otherwise, if it has any descendent w such that $n_v - n_w \leq \lambda$, we take the descendent w among them with the smallest number of points and keep w as the only child of v, and associate all points in its other children directly to v. Now v is a deflated node. If there is no such child w, we do not change v. The new tree T' is clearly λ-deflated, and this procedure clearly takes $O(n)$ time. It remains to show that the complexity of T' has gone down by a factor λ.

Lemma 9.3 *A λ-deflated quadtree T' produced as described above has $O(n/\lambda)$ nodes.*

Proof Let T'' be the subtree of T' that contains all nodes v with $n_v > \lambda$, and suppose that every cluster node in T'' has been contracted with its child. We will show that T'' has $O(n/\lambda)$ nodes, which implies the claim, since the child of a cluster node can never be another cluster node, and because all the non-cluster nodes in T' which are not in T'' must be leaves. We count the nodes in T'' as follows.

- Since the leaves of T'' correspond to disjoint subsets of P of size at least λ, there are at most n/λ of them.

- The bound on the leaves also implies that T'' contains at most n/λ nodes with at least two children.
- The number of nodes in T'' with a single child that has at least two children is likewise bounded.
- When an internal node v has a single child w that also has only a single child, then by construction v and w together must contain at least λ points in their cells, otherwise they would not have been two separate nodes. Thus, we can charge $\lambda/2$ points from P to v, and the total number of such nodes is $2n/\lambda$. ■

9.2.2 Preprocessing Algorithm

Given a set of k-overlapping disks $\mathcal{R}$, we first compute a quadtree T for Q as in the previous section. Next, we compute a k-deflated quadtree T' from T. By treating deflated nodes like cluster nodes and noting that the centre and corners of each box of T' can be contained in at most k disks, the same arguments as in Lemma 9.1 lead to the next lemma:

Lemma 9.4 *For every node v of T', C_v is intersected by $O(k)$ disks in $\mathcal{R}$.*

By Lemmas 9.3 and 9.4, the total number of disk-cell incidences in T' is $O(n)$. Thus, in $O(n)$ total time we can find for each $R \in \mathcal{R}$ the list of nodes in T' whose cells it intersects. Next, we determine for each node v in T' the portion X_v of the original quadtree T inside the cell C_v and build a point location data structure for X_v. Since X_v is a partial quadtree for at most k points, it has complexity $O(k)$, and since the X_v are disjoint, the total space requirement and construction time are linear. This finishes the preprocessing.

9.2.3 Reconstruction Algorithm

When we are given a precise set of points P, we first locate the points in the cells of T' just as in Section 9.1.3. This takes $O(n)$ time. Then we use the point location structures for the X_v to locate P in T in total time $O(n \log k)$. Now we turn T into a quadtree for $P \cup Q$ in $O(n \log k)$ time, and find the Delaunay triangulation in $O(n)$ expected time, as before. We arrive at the following theorem:

Theorem 9.3 *Let $\mathcal{R}$ be a set of n k-overlapping disks in the plane. We can preprocess $\mathcal{R}$ in $O(n \log n)$ time into an $O(n)$ size data structure, such that when a point set P containing one point from each region in $\mathcal{R}$ is given, a Delaunay triangulation of P can be computed in $O(n \log k)$ expected time.*

9.3 Thick Regions

The results of the previous sections still assume that the regions in $\mathcal{R}$ are disks. When a region R can have any shape, we can define the *thickness* of R as the ratio between the radii of the smallest enclosing disk of R and the largest disk contained in R: this was the third parameter for which we showed the results in the previous chapter could be generalised. We briefly show now that we can also deal with this situation, although the time bounds now does also become dependent on ρ. We define the *size* of a region R as the radius of the smallest enclosing circle of R.

Given is a set of β-thick regions $\mathcal{R}$ with a factor ρ difference in size between the smallest and largest regions in $\mathcal{R}$. We subdivide the regions into $\log \rho$ groups such that in each group the sizes of the regions differ by at most a factor of 2. For each group $\mathcal{R}_i$, let $n_i = |\mathcal{R}_i|$ and let r_i be the largest radius of a minimum enclosing circle for a region in $\mathcal{R}_i$. We replace every region in $\mathcal{R}_i$ by a disk of radius r_i that contains it. This set of disks is at most $(2k\beta)$-overlapping, so we can build a data structure for $\mathcal{R}_i$ in $O(n_i \log n_i)$ time by Theorem 9.3.

Now, to compute the Delaunay triangulation for a precise point set P, we first compute the Delaunay triangulation of each group P_i in $O(n_i \log(k\beta))$ time using the algorithm in Section 9.2.3. After that, we can combine the triangulations in time $O(n \log \log r)$ using an algorithm by Kirkpatrick [76].

Theorem 9.4 *Let $\mathcal{R}$ be a set of n β-thick k-overlapping regions in the plane such that the ratio of the largest and the smallest region in $\mathcal{R}$ is ρ. We can preprocess $\mathcal{R}$ in $O(n \log n)$ time into a $O(n)$ size data structure, such that when a point set P containing one point from each region is given, the Delaunay triangulation of P can be computed in $O(n(\log(k\beta) + \log \log \rho))$ time.*

9.4 Closing Remarks

In this chapter, we provided an alternative way of obtaining the same result as in the previous chapter: that it is possible to preprocess a set of disjoint unit disks in $O(n \log n)$ time, such that the Delaunay triangulation of a precise set of points can be computed in linear time once these points are given. The difference is that the algorithm in this section is randomised, and that it is much simpler to implement in practice. Furthermore, we showed how to extend the result to partially overlapping thick regions of different sizes.

The results in this chapter also appeared in [19]. Apart from these results, we also present in that paper a very simple randomised algorithm to obtain the same result that does not even rely on quadtrees. Also, we show how to extend the results to various other realistic input models, such as those introduced by De Berg *et al.* [34].

PART IV

Beyond Imprecise Points

Chapter Ten

Imprecise Lines

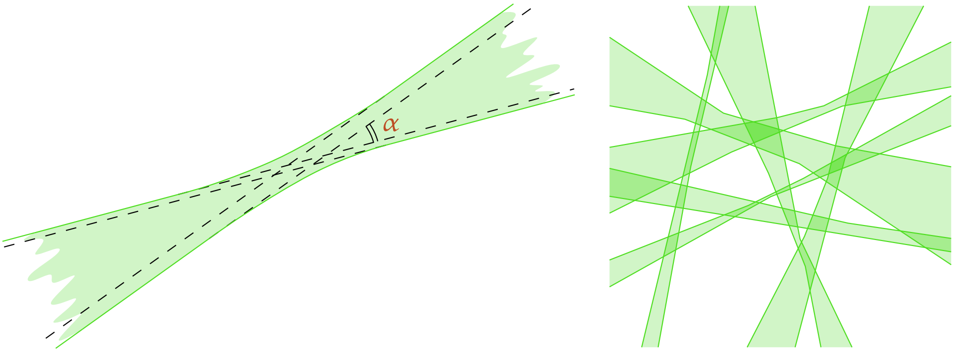
This thesis is about dealing with imprecision in the input data of geometric algorithms. Until now, we have considered only situations where this input data consists of a set of points. In this last part of the thesis, we will move away from the subject of imprecise points, and investigate how imprecision in other geometric objects can be modelled and what algorithmic problems this leads to. A short discussion about this was already given in Section 2.4. In this chapter, we will focus on lines; in the next chapters polygons are studied.

Lines are, next to points, the second most basic geometric objects. We model imprecise lines in the same way as imprecise points: by defining for each imprecise line a set of possible lines. As discussed in Section 2.4.1, though, we run into trouble when trying to define some properties of these sets of lines. Most notably, we would like to define when a set of lines is convex. We propose the following definition.

Definition A set of lines L is *convex* if there exists a direction d such that no line $l \in L$ has direction d, and for any pair of lines $l, m \in L$ the following properties hold:

1. If l and m are parallel, then all other lines parallel to them and between them are also in L.
2. If l and m intersect in a point p, then all other lines through p with a direction between l and m, rotating such that d is not encountered, are also in L.

We will assume from now on that sets of lines are closed sets, although the observations we make can easily be extended to open sets. Let L be a closed convex set of lines, and assume they are sorted by their directions cyclicly from d counterclockwise back to d. Then there are two lines $l, m \in L$ that have the smallest and largest direction. Let the angle between l and m be α. We call α the *limit angle* of L, see Figure 10.1(a). If the lines are undirected, α is smaller than π. If the lines are directed, α must be



(a) (b)

Figure 10.1 (a) An imprecise line with limit angle α. (b) A set of imprecise lines.

smaller than 2π. For undirected lines, this means L will either be a strip ($\alpha = 0$), or it can be described by two convex curves that have asymptotes with an angle α between them. The set L is the set of all lines that lie completely within the portion of the plane bounded by the curves.

For directed lines, something interesting happens as soon as $\alpha > \pi$. We can use the convexity properties to prove that if some line l is in L, any translation of l that is rotated some arbitrarily small ε is also in L. However, when considering imprecise lines it seems reasonable to assume that $\alpha < \pi$. In that case, our definition for directed lines coincides with the definition from [51]. Our definition of convexity also coincides with the dual definition, when we rotate the plane such that d becomes vertical (but since we need a different rotation for each imprecise line, we cannot use the dual definition as it is).

If a set of lines is convex with a limit angle $\alpha < \pi$ and has piecewise linear boundaries of constant description size, we call it a *bundle*. Figure 10.1(b) shows an example of a set of bundles in the plane.

We will show in the next sections that for this model of imprecise lines, we can also develop efficient algorithms. We study two well-known geometric problems that take a set of lines as input. In Section 10.1 we study the problem of linear programming, where the objective is to minimise the height of a point that lies on one side of all input lines. We show that the upper bound on the solution can be computed optimally in linear time, while the lower bound takes $O(n^2)$ time in general. When the layout of the bundles is restricted, though, we show that this time bound can improve to $O(n \log n)$ or even $O(n)$, depending on the exact restriction. In Section 10.2 we study the problem of vertical extent, where the objective is to find the shortest vertical line segment that intersects all input lines. For this problem, we again show that the upper bound can be computed in linear time. The lower bound on this problem

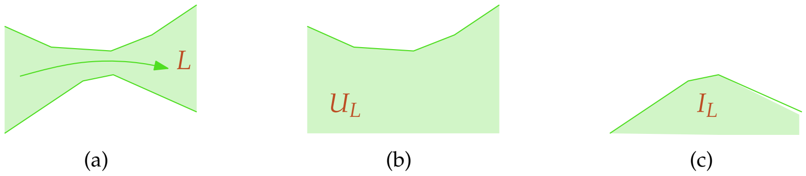


Figure 10.2 (a) An oriented bundle L. (b) The union of all halfplanes to the right of the possible lines in L. (c) The intersection of all halfplanes to the right of the possible lines in L.

takes $O(n^2 \log n)$ time, but under a certain restriction of the bundles this improves to $O(n \log n)$ time.

10.1 Linear Programming

Linear programming is a very important tool used in many disciplines across science. In a linear program, there is a number of variables on which a linear *objective function* needs to be optimised, while satisfying several linear *constraints*. Translated into a geometric problem, the d variables form a d-dimensional Euclidean space, the constraints form $(d-1)$-dimensional hyperplanes, and the objective function becomes a d-dimensional vector. The problem is now to find the point furthest in the direction of the objective function that lies on the correct side of all hyperplanes. By rotation, a linear program can be reformulated so that the objective function is optimised by minimising the dth (“vertical”) coordinate.

In the planar case, $d = 2$, and the hyperplanes become lines. In fact, they become *directed* lines: the solution is constrained to lie to the *left* of every line. This problem can be solved in $O(n)$ time [94]. When the lines are imprecise, we are interested in the lowest and highest possible points that could be the lowest point to the left of a set of lines taken from a collection of bundles $\mathcal{L}$. These points give us bounds on the possible values that the “real” solution can have.

When dealing with *oriented* bundles, we can define some useful regions of the plane, as was already observed in [51]. Let L be a bundle, and $l \in L$ be some line. We define by H_l the halfplane to the right of l. Now, we denote by $U_L = \bigcup_{l \in L} H_l$ and $I_L = \bigcap_{l \in L} H_l$ the union and intersection of all these halfplanes of the lines in L. Figure 10.2 shows an example.

10.1.1 Largest Value

In this section, we study the following problem:

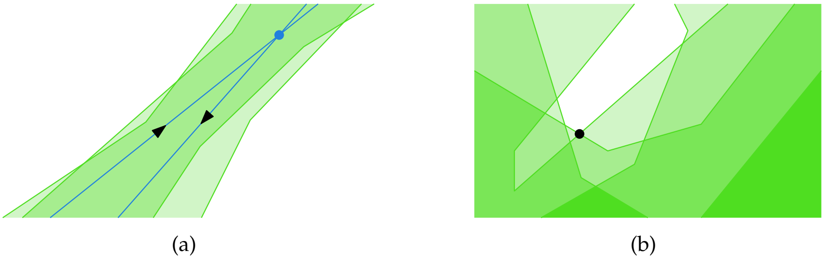

(a) (b)

Figure 10.3 (a) By rotating the two lines, the lowest point to the left of both of them can be made arbitrarily high. (b) The largest possible value is the smallest value in the combined free space.

Problem 10.1 *Given a set $\mathcal{L}$ of imprecise directed lines modelled as oriented bundles, choose one directed line from every bundle such that the point to the left of all lines with the lowest y-coordinate is as high as possible.*

In this problem, we want to choose the lines such that we restrict the solution as much as possible. In particular, this means that if the problem can be made infeasible, we want to do so. In a way, making the problem infeasible corresponds to having the lowest feasible point infinitely high. Indeed, already with two imprecise lines it can be possible to place the lines such that the lowest point on the correct side of them is arbitrarily high, as is illustrated in Figure 10.3(a), and by actually making the lines parallel the feasible space becomes empty.

To solve the problem, consider for each imprecise line L the region U_L. We will treat these regions as forbidden regions, and compute the lowest point that lies outside U_L for all lines. Figure 10.3(b) shows an example.

Lemma 10.1 *The lowest point outside all regions U_L is an optimal solution to Problem 10.1.*

Proof Let p be the lowest point outside the union of the U_L regions, and let p^* be an optimal solution. Clearly, no matter how we choose the lines in $\mathcal{L}$, p will always be in the free space. Thus p^* cannot be any higher than p.

Now p will be on a vertex of the polygonal region that is the union of the U_L regions. This vertex is defined by two lines l and m. If l and m belong to different imprecise lines, we can choose both of them, and clearly the optimal solution p^* must be at least as high as p. If l and m belong to the *same* imprecise line L, then by convexity of L and the fact that the limit angle $\alpha < \pi$, we know that the horizontal directed line from left to right through p is also in L, and we can choose that line. Again p^* must be at least as high as p.

We conclude that p and p^* have the same height, and therefore p is also an optimal solution. ■

To compute the solution, consider the line segments bounding U_L. Take the lines supporting those segments, and do this for all imprecise lines. Apply normal linear programming to the resulting set of lines. The solution we find this way is the solution to the imprecise problem.

Theorem 10.1 *Given a set $\mathcal{L}$ of n imprecise directed lines modelled as oriented bundles, we can pick a line from each bundle such that the lowest point to the left of them is as high as possible in $O(n)$ time.*

10.1.2 Smallest Value

In this section, we study the following problem:

Problem 10.2 *Given a set $\mathcal{L}$ of imprecise directed lines modelled as oriented bundles, choose one directed line from every bundle such that the point to the left of all lines with the lowest y-coordinate is as low as possible.*

In this case, we want to find the lowest point such that in every bundle, there is a line that has this point to its left. Therefore, we can define for each imprecise line L the *potential free space* as the complement of I_L.

Observation 10.1 *If p is a point in the plane, a choice of lines for $\mathcal{L}$ that has p to the left of all of them exists if and only if p lies in the potential free space of all lines in $\mathcal{L}$.*

As a consequence, we are now looking for the lowest point in the intersection of a collection of concave regions in the plane. We could find this point by explicitly computing this intersection, which takes $O(n^2)$ time in general, and in Section 10.1.2.4 we show that in general this is indeed the best we can do.

However, when our collection of imprecise lines satisfies some additional, natural constraints, we can in some cases solve the problem more efficiently. We call a bundle *diagonal* if it contains no horizontal or vertical lines. We call a bundle *upfacing* if all lines in the bundle are directed from left to right. We will call a bundle *δ-fat* if $\alpha < \pi - \delta$ for some constant $\delta > 0$.

10.1.2.1 Diagonal Bundles

Let $\mathcal{L}$ be a collection of diagonal and upfacing bundles, see Figure 10.4(a). In this case, we can solve Problem 10.2 in linear time, because it is an LP-type[1] problem. LP-type problems were already briefly discussed in Section 3.2.2.1 of this thesis.

Recall that an LP-type problem is defined on a set of objects H and a function $w : 2^H \to W$, where W is some totally ordered set of possible values, and the goal is

---

[1] In fact, LP-type stands for "linear programming"-type, and describes a class of problems that are similar to linear programming and can be solved in linear expected time using a generic approach [92]. Perhaps it is not surprising that linear programming with imprecise lines fits in this framework.

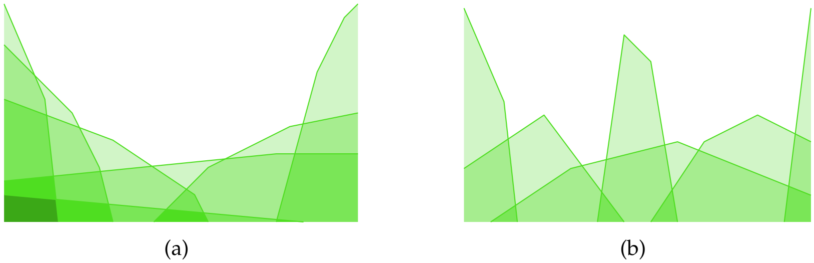

(a) (b)

Figure 10.4 A set of bundles, showing the I_L regions in grey. (a) The potential free space of a set of diagonal, upfacing bundles. (b) The potential free space of a set of upfacing bundles.

to compute $w(H)$. The two axioms that must hold for a problem to be LP-type are monotonicity:

$$\forall_{F \subseteq G \subseteq H} \quad : \quad w(F) \leq w(G)$$

and locality:

$$\forall_{F \subseteq G \subseteq H,\, h \in H} \quad : \quad w(G) = w(F) < w(F \cup \{h\}) \quad \longrightarrow \quad w(G) < w(G \cup \{h\})$$

In this case, $H = \mathcal{L}$ is the set of imprecise lines, or equivalently the set of regions I_L of the lines, and w measures the height of the lowest point in complement of the union of these regions. The first axiom is clearly holds for any set of imprecise lines. For the second to hold, though, we need the restrictions mentioned before. Intuitively, it states that if adding a new region to a collection of regions changes the optimal solution, then the newly added region should be one of the defining regions of the new optimum. This is the case for diagonal upfacing bundles, but not for more general bundles.

10.1.2.2 Upfacing Bundles

Let $\mathcal{L}$ be a collection of upfacing bundles, see Figure 10.4(b). We can solve the problem by computing the upper envelope of the potential free regions in $O(n \log n)$ time [67]. The complexity of this envelope is $O(n\alpha(n))$, where $\alpha(\cdot)$ denotes the inverse of the Ackermann function, and we can clearly find the lowest point on it by just looking at all the vertices.

In certain models of computation, this bound can be proven to be optimal, since we can reduce the *maximum gap* problem to it, which has a $\Theta(n \log n)$ lower bound [81]. The model of computation used in that proof is quite restrictive, though: if we allow to use the *floor* function in constant time, as we do in Chapters 5 and 9, the maximum gap can be computed in linear time [53]. Nonetheless, we now present the reduction from maximum gap.

In the maximum gap problem, we are given a set of real numbers, and we want to report the largest difference between any pair of consecutive numbers (consecutive if they were sorted, but they are given in arbitrary order). For each number we create a bundle. Consider the point $p = (x, 0)$ for the number x. We consider the line l through p with slope 1 and the line m through p with slope -1. We create a bundle L consisting of l and m and all other lines through p with slopes between -1 and 1. Now the region I_L forms a wedge with p as its top, and two halflines with slope 1 and -1 going down. The lowest point above all wedges corresponds to the two consecutive numbers with the largest difference.

10.1.2.3 Fat Bundles

Fatness seems to be a very reasonable restriction on imprecise lines. We already assumed that for each imprecise line, there is some direction d which we know the line does *not* have. We now impose a slightly stronger restriction, by saying that there is at least a small angle of directions the line cannot have. This angle is a positive constant denoted by δ.

Let $\mathcal{L}$ be a collection of fat bundles, see Figure 10.5(a). In this case, we can solve the problem in $O(n \log n)$ time.

Lemma 10.2 *The union of the* I_L *regions of a set of* δ*-fat bundles* $\mathcal{L}$ *can be computed in* $O(n \log n)$ *time.*

Efrat *et al.* [41] prove that the union of a set of δ-fat wedges can be computed in $O(n \log n)$ time. Their proof is an adaptation of [91], with some ideas from [97]: they describe only the differences. The proof can be adapted once more to also work for our I_L regions, which are “clipped” δ-fat wedges. Our adaptation is straightforward, but we give a brief sketch of the whole proof here in order to save the reader from having to reconstruct all adaptations.

Proof The main idea is to divide the imprecise lines into $k = 2\pi/\delta$ groups depending on their angles. Let $d_0, \ldots, d_k$ be k equally spaced directions. Let $\mathcal{L}_i$ be the set of imprecise lines that do not contain any line with direction d_i. Because the imprecise lines are δ-fat, each imprecise line is in at least one group. Within each group, we can compute the union of the clipped wedges I_L in $O(n \log n)$ time, because such a set of wedges is upfacing when we rotate the plane such that d_i becomes vertical. Then we explicitly compute the union of these k structures, which can be done in $O(n \log n + m)$ time where m is the total number of intersections [27]. The bulk of the analysis in [41] is used to show that m is linear in n. Since the number of groups is constant, it is sufficient to prove that for each pair of groups the number of intersections is linear. This follows from the fatness: in the worst case, all “spikes” are exactly δ-fat, but even then they still cannot intersect too often. ■

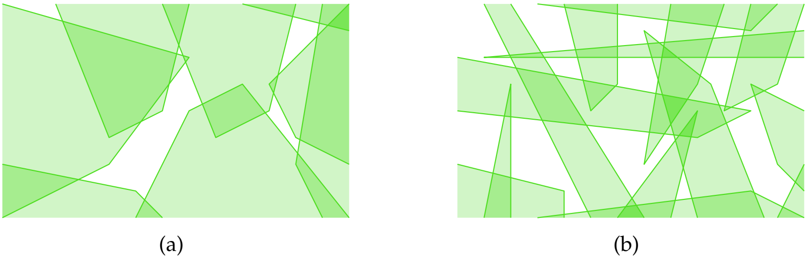

(a) (b)

Figure 10.5 A set of bundles, showing the I_L regions in grey. (a) The potential free space of a set of fat bundles. (b) The potential free space of a general set of bundles.

10.1.2.4 General Bundles

Let $\mathcal{L}$ be a collection of bundles without any further restrictions, as depicted in Figure 10.5(b). In this case, it is likely that there is no algorithm that solves the problem faster than in $\Theta(n^2)$ time.

We reduce from covering a rectangle with strips, which falls in a class of “$O(n^2)$ problems” [50], also sometimes called *3SUM-hard* problems, for which the best known solutions take $\Theta(n^2)$ time. In this problem, the input is a rectangle R and a set of *strips* $\mathcal{S}$: each strip is the region between two parallel lines. The question is to decide whether the union of $\mathcal{S}$ covers R. In our case, we represent each strip S of $\mathcal{S}$ by an imprecise line. For a strip defined by two lines l and m, we create a bundle with limit angle $\alpha = \pi - \delta$ for some very small value δ. Assume l and m have opposite directions, and they lie to the right of each other. We rotate l by δ around some point in R, and then add both l and m to the new bundle L. Then l and m intersect somewhere far away outside R, and we add all other lines between l and m through this intersection point to L too, to make it convex. If δ goes to zero, the part of I_L inside R comes arbitrarily close to the strip S. Finally, we add an extra bundle to represent each side of R, each consisting of one single line. Clearly, the resulting free space is empty if and only if the strips $\mathcal{S}$ cover R.

We note that it is easy to achieve quadratic time by just explicitly computing the arrangement of the bundles.

Theorem 10.2 *Given a set $\mathcal{L}$ of n imprecise directed lines modelled as oriented bundles, we can pick a line from each bundle such that the lowest point to the left of them is as low as possible in*

- $O(n)$ *expected time when $\mathcal{L}$ contains only diagonal upfacing bundles.*
- $O(n \log n)$ *time when $\mathcal{L}$ contains only upfacing bundles.*
- $O(n \log n)$ *time when $\mathcal{L}$ contains only δ-fat bundles for some constant $\delta > 0$.*
- $O(n^2)$ *time for general bundles.*

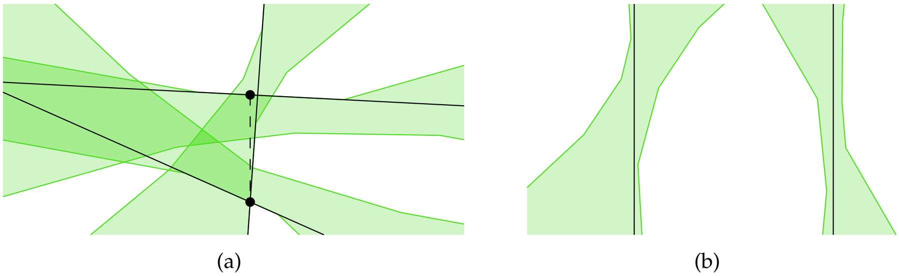


Figure 10.6 (a) Three bundles without any vertical lines. The largest possible vertical extent is achieved by choosing the three shown lines. (b) When there are at least two vertical bundles, the vertical extent could be arbitrarily large.

10.2 Vertical Extent

Another basic geometric problem defined on a set of lines (or, in higher dimensions, hyperplanes), is finding the *vertical extent*. The objective is to find the shortest vertical line segment that intersects all lines. Many geometric shape fitting problems can be reduced to the vertical extent problem by a technique called *linearisation* [62, 63]. It is also directly dual to the *vertical width* problem: the smallest vertical distance between two parallel lines that enclose a set of points in the plane. Vertical extent is also an LP-type problem, and can therefore be solved in $O(n)$ expected time.

Unlike for linear programming, the input to the vertical extent problem is a set of undirected lines. This means that even though the two problems are similar in the traditional, precise setting, some different ideas are needed to solve them when the lines are imprecise. Again, we are interested in the largest and smallest possible values that the vertical extent can have.

10.2.1 Largest Value

Problem 10.3 *Given a set $\mathcal{L}$ of imprecise lines modelled as bundles, choose one line from every bundle such that the vertical extent of the resulting lines is as large as possible.*

We call a bundle *vertical* if it contains at least one vertical line.

When there are no vertical bundles, the largest vertical extent of the bundles is likely to be just the smallest vertical extent of the set of lines that support the boundaries of the bundles, which can be computed in linear time. However, this is not always the case, because it could be that two of the lines defining the vertical extent belong to the same bundle. In this case, the worst vertical extent possible is smaller than the vertical extent of all the supporting lines. Figure 10.6(a) shows this situation.

To handle this, we start by computing the vertical extent of the supporting lines. Assuming no degeneracies, the vertical extent is always defined by three lines: it is the vertical distance between the intersection of two lines and a third line. If these three lines come from three different bundles, then we are done: clearly taking these three lines results in the largest possible vertical extent. However, when two or all of these three lines belong to the *same* bundle, we cannot use all of them, so the optimal solution will be smaller. In this case, though, we can prove that one of the bundles involved must provide one of the three lines that do define the final solution. So, we can guess which line to use, and for each of these we again solve the same problem using all defining lines of the remaining regions. Then, after at most three steps, we have found the solution.

Observation 10.2 *Let $\mathcal{L}$ be a set of bundles. Suppose the vertical extent of the supporting lines of $\mathcal{L}$ is defined by lines $l_1 \in L_1 \in \mathcal{L}$, $l_2 \in L_2 \in \mathcal{L}$ and $l_3 \in L_3 \in \mathcal{L}$. Then the largest possible vertical extent of a set of lines obtained by choosing one from each bundle in $\mathcal{L}$ is defined by at least one line from L_1, L_2 or L_3.*

Proof For clarity: in this proof L_1, L_2 and L_3 may refer to the same bundle. (When L_1, L_2 and L_3 are three different bundles, the solution is directly defined by l_1, l_2 and l_3.) Assume the vertical extent of l_1, l_2 and l_3 is d.

Now, for contradiction, suppose the optimal choice of lines to maximise the vertical extent results in a vertical extent d' defined by three lines m_1, m_2 and m_3, none of which are from any of the bundles L_1, L_2 or L_3. Clearly, $d' \leq d$. Consider the x-coordinate of this optimal vertical extent. The lines l_1, l_2 and l_3 span a vertical interval of at least d at this x-coordinate. This means that at least one line l_i must lie outside the vertical interval of d' formed by m_1, m_2 and m_3. Since l_i is from a different bundle, we can include this line, yielding a solution with a vertical extent that is at least as large as d'. ■

When there are at least two vertical bundles, the problem is trivial and the answer is ∞, since we can take two parallel lines that are arbitrarily close to vertical, and the vertical distance between them will be arbitrarily large (except in the degenerate case where vertical is one of the extreme directions of a bundle). Figure 10.6(b) shows this situation.

When there is exactly one vertical bundle, we cannot get this arbitrarily large extent, but we can simply use the algorithm for no vertical bundles by splitting the bundle into two bundles, one containing all lines “left” of the vertical and one with all lines “right” of the vertical.

Theorem 10.3 *Given a set $\mathcal{L}$ of n imprecise lines modelled as bundles, we can pick a line from each bundle such that the vertical extent of those lines is as large as possible in $O(n)$ time.*

10.2.2 Smallest Value

Problem 10.4 *Given a set* $\mathcal{L}$ *of imprecise lines modelled as bundles, choose one line from every bundle such that the vertical extent of the resulting lines is as small as possible.*

This problem can also be interpreted as computing the shortest vertical segment that intersects all bundles (the region between the curves that define the bundle, to be precise). Again, we separately study the case where none of the bundles contain vertical lines, and the case where some of them do.

10.2.2.1 No Vertical Bundles

Even though the imprecise lines are undirected, when there are no vertical bundles, we can view the lines as directed lines from left to right. Then the bundles are *upfacing* by the definition of Section 10.1, and we can solve the problem in a similar manner as linear programming.

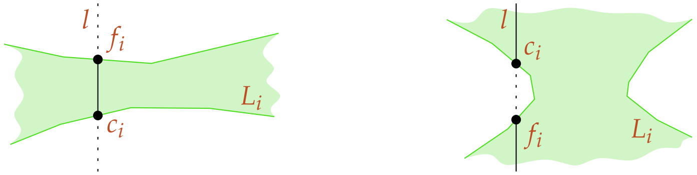
In Section 10.1.2.2, we considered the set of regions I_L for all imprecise lines in $\mathcal{L}$. The complement of the union of these defined the potential free space. Now, consider the polygonal line separating the free space from the regions (this is the upper envelope of the regions). Any vertical line segment that intersects all bundles needs to have its upper endpoint on or above this polyline. In the same way, we compute the intersection of all regions U_L, and the polyline bounding this. The vertical segment needs to have its lower endpoint on or below this polyline. Now, we simply scan both envelopes together to find the shortest vertical segment in time linear in the size of the polylines, which can be at most $O(n\alpha(n))$.

As mentioned before, computing the envelopes takes $O(n \log n)$ time [67], which is the total running time of the algorithm.

10.2.2.2 General Bundles

When there are vertical bundles, the problem becomes more difficult since the vertical lines partition the lines in the bundle into two groups, and we have to decide which of them to choose a line from. In the previous case, any vertical line crossed all bundles in exactly one interval. Now it may cross a bundle in an “inverted” interval, that is, there is an interval where the bundle is *not* crossed but outside the interval it is. When there are vertical bundles, we show how to solve the problem in $O(n^2 \log n)$ time.

To start with, consider a vertical line l (not among the bundles, just any line). We will make some observations about the shortest vertical interval on l that intersects all bundles. First, the boundary of each non-vertical bundle intersects l in exactly two points, with the interior of the bundle between them. The boundary of each vertical bundle either does not intersect l at all (or in a single point), in which case we do not



(a)

(b)

Figure 10.7 (a) The sweepline l intersects a non-vertical bundle in a single interval. (b) The sweepline intersects a vertical bundle in an inverted interval, or lies completely inside the bundle.

care about this bundle since any interval of l intersects it, or it also intersects l in two points, but now with the *exterior* of the bundle between them.

We call an intersection between l and a bundle boundary a *ceiling* if it has the interior of the bundle above it, and a *floor* if it has the interior of the bundle below it. So, for a fixed line l, any bundle L_i has at most one ceiling c_i and one floor f_i, and non-vertical bundles have their floor above their ceiling while vertical bundles have their ceiling above their floor. Figure 10.7 illustrates this. Note that the shortest interval of l that intersects all bundles will have its top at a ceiling and its bottom at a floor (unless there is a single point on l that intersects all bundles), because otherwise the top point could be moved farther down or the bottom point farther up without losing any bundles.

Now, still keeping l fixed, consider for each bundle the point $(-f_i, c_i)$. This determines a set of points P in $\mathbb{R}^2$. We note that the non-vertical bundles all appear below the line $y = -x$, while all the vertical bundles appear above this line, and split P accordingly into P_N and P_V. We say that a point $p = (x_p, y_p) \in \mathbb{R}^2$ *dominates* a point $q = (x_q, y_q) \in \mathbb{R}^2$ when $x_p \geq x_q$ and $y_p \geq y_q$. We say p *strictly* dominates q when $x_p > x_q$ and $y_p > y_q$. Now consider a candidate interval of l as a point $q \in \mathbb{R}^2$. We prove below that this interval is a valid solution exactly when q dominates all points of P_N and is not dominated by any point of P_V. We call the set of such candidate points p the *feasible region* of l. Figure 10.8 shows an example set of bundles, and the corresponding point set and feasible regions at two different positions of l.

Lemma 10.3 *An interval $[a, b]$ of l is a valid solution if and only if the corresponding point $q = (-a, b)$ dominates all points of P_N and is not strictly dominated by any point of P_V.*

Proof A point $p_i = (-f_i, c_i)$ in P_N correspond to an interval $[c_i, f_i]$ that we have to intersect. The interval $[a, b]$ intersects $[c_i, f_i]$ when $c_i \leq b$ and $f_i \geq a$. This is equivalent to saying that q dominates p_i.

A point $p_i = (-f_i, c_i)$ in P_V correspond to an interval $[f_i, c_i]$ of which we have to intersect the complement. The interval $[a, b]$ intersects the complement of $[f_i, c_i]$ when

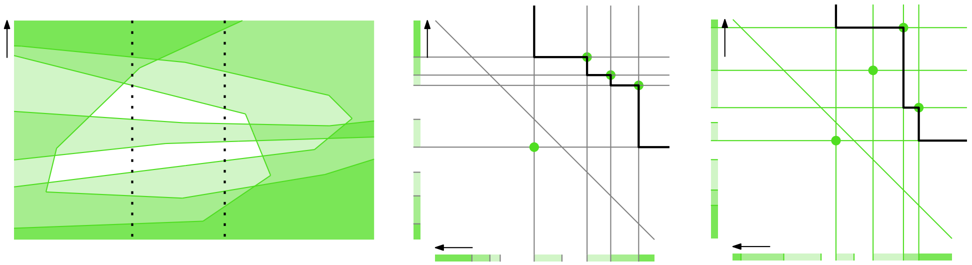

(a) (b) (c)

Figure 10.8 (a) An arrangement with four bundles. One of the bundles does not contain a vertical line, the other three do. The dotted lines are two locations of the sweepline. Between the two vertical lines, there is one event. (b, c) The rectilinear curve that represents the highest feasible floor below each ceiling, and the lowest feasible ceiling above each floor. The ceilings are shown on the vertical axis, the floors on the horizontal axis.

$a \leq f_i$ or $b \geq c_i$. Therefore, it does *not* intersect the complement of the interval when $a > f_i$ and $b < c_i$, which is equivalent to saying that p_i strictly dominates q. ■

Given a set of points P, we define the *staircase envelope* of P to be the curve that separates the region of points that are dominated by some point in P from the rest of the plane. Note that the boundary of the feasible region is determined by the staircase envelope of P_V, cut off by the two lines determined by the points in P_N with the largest x-coordinate and largest y-coordinate. Furthermore, note that the optimal interval of l minimises its length, which corresponds to the point q in the feasible region associated to l that minimises $x + y$, provided it is not below the line $y = -x$.

Now we will sweep a vertical line l through the original space from left to right. On the current line, we maintain a sorted list with all bundle boundaries currently intersected by the sweepline. We also maintain the point set P, the staircase envelope of P_V, and the two lines at the largest x-coordinate and smallest y-coordinate among the points in P_N. Together these define the feasible region on l. If we maintain this structure and remember the shortest segment throughout the sweep, this will give us the solution to the problem. We can compute the structure for the leftmost position of the line in $O(n \log n)$ time. We have an event whenever two boundaries cross, or when a ceiling and a floor meet and vanish or appear. There can be $O(n^2)$ events in total. However, we also need to sort the events on their x-coordinate, which takes $O(n^2 \log n)$ time;[2] this will dominate our running time. We will now describe how to handle these events to maintain the required structure in constant time per event.

---

[2] Whether this can be done any faster than in $O(n^2 \log n)$ time is a long-standing open problem [66, 48, 106].

Intersection of two ceilings. The most important event occurs when two ceilings intersect. In this case, two of the points in P switch their y-coordinate, say p_i and p_j. We need to maintain the correct staircase envelope.

If p_i and p_j are in different sets, nothing changes. If p_i and p_j are both in P_N, we need to check only whether one of them was the point with the largest y-coordinate. If so, the other one takes over this function. This test can be executed in constant time.

If p_i and p_j are both in P_V, suppose without loss of generality that $y_i < y_j$ before the event. There are two cases, depending on the x-coordinates. Suppose $x_i < x_j$. Then p_i is dominated by p_j, so does not appear on the staircase. If p_j also is not on the staircase, then nothing happens. Otherwise, consider the point p_k that is the neighbour of p_j on the staircase. If $x_i > x_k$, then p_i must be inserted on the staircase between p_k and p_j. This can be done in constant time. Otherwise, nothing changes.

In the remaining case, $x_i > x_j$ before the event. In this case, p_i and p_j are independent before the event but p_j will dominate p_i after the event. If p_i was not on the staircase before the event, nothing changes. Otherwise, it must be removed from the staircase. This can be done in constant time, since there are no other points that could appear on the staircase instead (as they would need to have a y-coordinate between y_i and y_j).

Intersection of two floors. This event is symmetric to the case above.

Intersection of a ceiling and a floor. When a ceiling crosses a floor, usually nothing of interest happens, and we do not do anything with this event.

Note, though, that something special happens when the current shortest solution was between the crossing ceiling and floor. In that case, after the event we have a solution of negative length (that is, a point below the "identity diagonal" line). This is no problem though, when the algorithm returns an interval of negative length, it means that there exists a single point in the common interior of all bundles.

Disappearing bundle. Vertical bundles have the property that their ceilings and floors are not x-monotone, so at some point while sweeping the line they can disappear or appear. When the ceiling c_i and the floor f_i of the same bundle meet, then these points disappear from the data structure. This corresponds to a point in P_V touching the identity diagonal. When this happens, it needs to be removed from the staircase if it is still on there. There cannot be any other points of P_V that are dominated by p_i, because all of those are above the diagonal, therefore we can update the staircase envelope in constant time by simply removing p_i.

Appearing bundle. Similarly, at some events a new ceiling and floor of the same bundle appear together. This implies we have to add another point to P_V, somewhere on the identity diagonal. If it falls outside the current envelope, it needs to be added to it. This cannot be done in constant time. However, it can easily be handled in $O(\log n)$ time, and since there are only $O(n)$ events of this type, this does not increase the running time.

Theorem 10.4 *Given a set $\mathcal{L}$ of n imprecise lines modelled as bundles, we can pick a line from each bundle such that the vertical extent of those lines is as small as possible in*

- $O(n \log n)$ *time when $\mathcal{L}$ contains no vertical bundles.*
- $O(n^2 \log n)$ *time for general bundles.*

10.3 Closing Remarks

In this chapter, we introduced a way to model imprecision in lines, and we studied two geometric problems that take a set of lines as input under this model. For the problem of linear programming, we showed that the upper bound on the value can be computed in linear time, while the lower bound can be computed in times varying from $O(n)$ to $O(n^2)$, depending on what further restrictions we make on the model. For the problem of vertical extent, we provided an $O(n \log n)$ algorithm for computing the upper bound, while the lower bound takes $O(n^2 \log n)$ in general but can also be computed in $O(n \log n)$ time when none of the bundles contain a vertical line.

The results in this section also appeared in [88], although that paper makes the false claim that the smallest vertical extent for general bundles can be computed in $O(n^2)$ time. These are the first results on geometric problems that directly model imprecision in lines, and as such there are many directions for further research.

Chapter Eleven

Bounds on the Perimeter of a Polygon

In the last two chapters of this thesis, we will consider the effect of data imprecision in composite structures, as discussed in Section 2.4.2. Perhaps the simplest possible composite structure is a polygon. A polygon is a (cyclic) sequence of points in the plane, such that every pair of consecutive points in the sequence is connected by a straight line segment. Many geometric problems take a polygon as input.

When an input polygon is imprecise, it is not clear how to model the imprecision. A natural thing to do is to describe it as a sequence of imprecise points, where each point is again modelled by a region of possible locations. More precisely, the input is now a sequence $\mathcal{R} = \langle R_0, R_1, R_2, \ldots, R_{n-1} \rangle$ where indices wrap around, so $R_n = R_0$ etc. We are looking for a sequence of points $P = \langle p_0, p_1, \ldots, p_{n-1} \rangle$ with $p_i \in R_i$, and we also regard P as a polygon.

However, before even considering computing complicated things on such an imprecise polygon, there are some other things to keep in mind. Polygons have can certain properties. For example, every polygon has a length: the sum of the lengths of its line segments. Also, we call a polygon *simple* when it has no self-intersections. Simple polygons are an important class of polygons, and arise naturally when using polygons to describe the boundaries of regions. And then, simple polygons also have an area. These properties are easily tested or computed for precise polygons, and this is something we can easily take for granted. However, when the polygon is imprecise, it suddenly becomes challenging to compute the values of these, now also imprecise, properties and measures.

In this chapter, we look at the problem of computing the length, or *perimeter*, of an imprecise polygon; in the next chapter we will study the issue of *simpleness*. Of course, the perimeter of a polygon is a numerical value, so we will follow the paradigm of

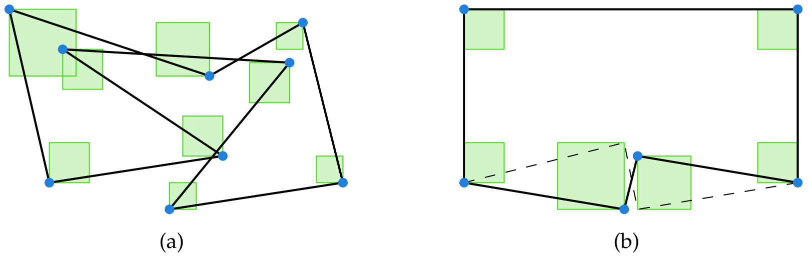

(a) (b)

Figure 11.1 (a) The longest perimeter solution. (b) The longest perimeter may have local optima.

Part II and compute upper and lower bounds to this value. We consider only squares as imprecision regions. As discussed in Chapter 4, disks cause algebraic problems when maximising the area of the convex hull of a set of imprecise points, and the same is true for the perimeter of the convex hull [89], even when the order in which the points appear on the optimal hull is already known. This implies that the same issues also prevent us from providing exact algorithms for the perimeter of a polygon when the regions are disks. We show that in the case of square regions, both the upper bound and the lower bound can be computed in linear time.

11.1 Longest Polygon

We first consider computing upper bounds. For this problem, this is conceptually much easier than computing lower bounds, though in both cases we can obtain a linear-time algorithm. The problem we discuss in this section is the following:

Problem 11.1 *Given a set of axis-aligned squares and a cyclic order on them, choose a point in each square such that the perimeter of the polygon determined by those points in the given order is as large as possible.*

Figure 11.1(a) shows a typical solution. Note that in this problem, the solution space may contain local optima: solutions that cannot be improved by changing the position of a single vertex, and yet are not optimal. The example in Figure 11.1(b) cannot be improved by moving only one point. However, it is clear that “flipping” the two points at the bottom increases the perimeter. To solve the problem, we first show that all of the points have to be chosen at a corner of a square.

Similarly to previous chapters, such as Chapter 4 about the convex hull and Chapter 6 about the diameter, we can again make an observation about the placement of the points in an optimal solution.

Observation 11.1 *There is an optimal solution to Problem 11.1 such that all points are chosen at one of the corners of their square.*

Proof Suppose we have a point set P such that one of the points, say p, is not at a corner. Now consider its neighbours q and r. When we move p, the lengths of $\overline{pq}$ and $\overline{pr}$ are the only ones that change. If p is not at a corner, there are always two opposite directions in which it can move, i.e., it can move over a line segment in both directions. The lengths of $\overline{pq}$ and $\overline{pr}$ are hyperbolic functions of the position of p on this segment, and have a single minimum, so they are convex. Clearly their sum is convex too, so there is at least one direction for p to move in such that the sum of the two lengths does not decrease. ■

The problem can now be solved easily in linear time using dynamic programming. The optimal solution up to a certain square depends only on the optimal solution up to the previous square and the length of the new edge.

We start with a single point p_0, which is one of the corners of one of the squares, chosen to be the first square. Let D_i be the set of the four corners of square R_i. For any i and $d \in D_i$, we define $c_{i,d}$ as the optimal (longest) chain from p_0 to d. Now we have the following recursive relation for $i > 1$:

$$c_{i,d} = \max_{d' \in D_{i-1}} (c_{i-1,d'} + |d'd|)$$

At the end, we just need to connect the four corners of the last square to p_0 again, and take the maximum. This can be computed in linear time. We need to do this computation four times, once for each possible corner of the first square, which still gives a linear-time algorithm.

Theorem 11.1 *Given an ordered set of n arbitrarily sized, axis-aligned squares, the problem of choosing a point in each square such that the perimeter of the resulting polygon is as long as possible can be solved in $O(n)$ time.*

11.2 Shortest Polygon

Now, we study the problem of computing the lower bound.

Problem 11.2 *Given a set of axis-aligned squares and a cyclic order on them, choose a point in each square such that the perimeter of the polygon determined by those points in the given order is as small as possible.*

The optimal solution to the input in Figure 11.2(a) is shown in Figure 11.2(b). As this figure shows, the shortest polygon can touch the squares on just a corner, at a point on an edge, or go straight through the interior. To solve the problem, we will use the fact that we can treat and compute the horizontal and vertical components of the optimal solution separately, giving two partial solutions that can then be combined into the final optimal solution.

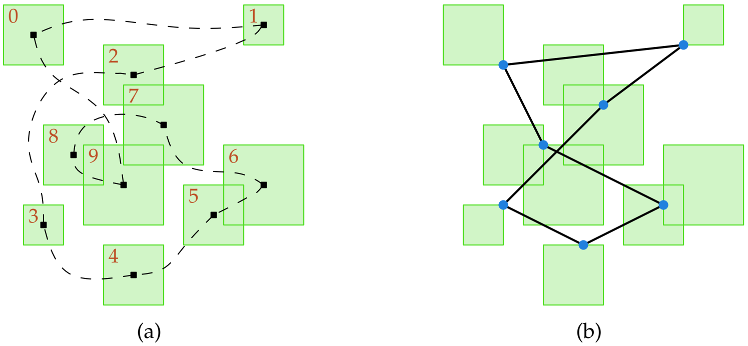


Figure 11.2 (a) An imprecise polygon with 10 vertices. (b) The optimal solution.

11.2.1 1-Dimensional Case

For a sequence of squares $\mathcal{R}$, let $\mathcal{R}_H$ be its horizontal projection. That is, the squares (imprecise 2-dimensional points) of $\mathcal{R}$ become intervals (imprecise 1-dimensional points) in $\mathcal{R}_H$, and the order of the imprecise points remains the same. In the same way, let $\mathcal{R}_V$ be the vertical projection of $\mathcal{R}$. Now we consider the same problem in a lower dimension.

Problem 11.3 *Given a set of intervals $\mathcal{I}$ and a cyclic order on them, choose a point on each interval such that the "perimeter of the polygon" determined by those points in the given order is as small as possible.*

The polygon in this case is degenerate, but its perimeter is just the sum of the distances between each pair of consecutive points. If all intervals have a common intersection (which can easily be checked in linear time), the optimal solution is to place all point at the same spot somewhere in this intersection. Otherwise, the problem can easily be solved in a greedy manner in $O(n)$ time. Let p be the leftmost point among all right endpoints of the intervals in $\mathcal{I}$. There must be an optimal solution that contains p, since it never makes sense to go farther to the right, and we do need to visit some point of p's interval. Now start at p, and go to the closest point of each succeeding interval, according to the given cyclic order. We observe that this procedure results in the optimal solution, and that any other solution can be greedily improved to get to the optimal solution (any point that has two neighbours on one side that can still move in that direction might as well do so).

Observation 11.2 *Problem 11.3 has only one locally optimal solution (perimeter value), when disregarding the positions of the vertices. This solution is also the global optimum.*

If we do regard the positions of the vertices, the optimal solution to Problem 11.3 is generally *not* unique. It consists of two kinds of vertices: those where the polygon changes direction, and those where it does not. The location of the vertices of the first

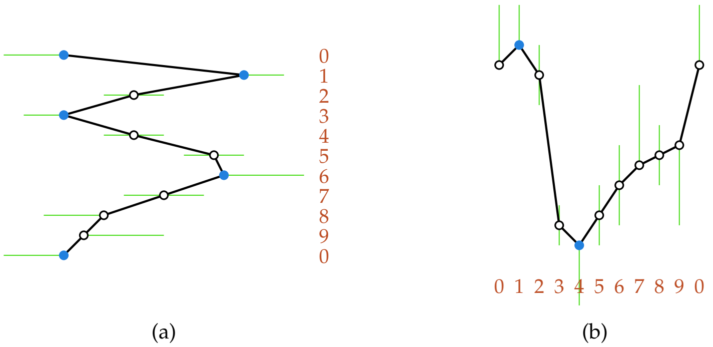


Figure 11.3 The horizontal and vertical projections of the imprecise polygon in Figure 11.2(a) (the intervals are drawn at different y-coordinates/x-coordinates for better visibility, but are in reality all on one horizontal/vertical line). The black vertices are *fixed*, the white vertices are *free*.

kind is the same in any optimal solution: they will be at one end of their interval, and moving them would increase the perimeter of the polygon. We call those vertices *fixed*. The vertices of the second kind can be moved around a bit locally without changing the perimeter of the polygon. We call them *free*. However, the x-order of any sequence of consecutive free vertices is fixed in any optimal solution, since the polygon does not change direction at them. This implies that two given free vertices p and q between two consecutive fixed vertices have the same left/right relation in any optimal solution.

11.2.2 Transformation

We observe the following useful relation between this problem and the original one.

Lemma 11.1 *Let P^* be the optimal solution to Problem 11.2 for a sequence $\mathcal{R}$. Then the horizontal projection P^*_H of P^* is an optimal solution to Problem 11.3 for the horizontal projection $\mathcal{R}_H$ of $\mathcal{R}$.*

Proof Suppose projection P^*_H is not an optimal solution of the reduced problem. By Observation 11.2, the reduced problem does not have any local optima apart from the global optimum, so there must be a point q_H on P^*_H, such that moving q_H to one side would improve the solution. This means that both its neighbours p_H and r_H must lie to the same side of q_H, while the interval over which q_H can move extends to that side. But this means that in P^*, the point q can also move in that direction, such that its distance to both p and r decreases, thus P^* would not be optimal. ■

Clearly, the lemma also applies to the symmetrical case for P^*_V and $\mathcal{R}_V$. Figure 11.3 shows the projections $\mathcal{R}_H$ and $\mathcal{R}_V$ of the example in Figure 11.2(a), and the partial

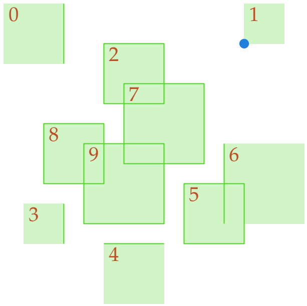


Figure 11.4 We know for each region from which direction the polygon enters, and in which direction it leaves again.

solutions we can compute for those problems.

Now, to solve Problem 11.2, we first compute the horizontal and vertical projections of $\mathcal{R}$ and their optimal solutions P_H^* and P_V^*. Now we lift the coordinates of the vertices of P_H^* and P_V^* back into $\mathbb{R}^2$. Let p be a vertex of P^*. If p_H is fixed in P_H^*, then p must lie on a vertical line segment with that x-coordinate. If p_V is fixed in P_V^*, it must lie on a horizontal line segment. If both are fixed, the location of p is already known.

Furthermore, for consecutive vertices p and q in P^*, the projections p_H and q_H tell us whether p should lie to the left or to the right of q, and the projections p_V and q_V tell us whether p is above or below q. This transforms our sequence $\mathcal{R}$ of squares into a new sequence of squares, line segments and points, where we know the horizontal and vertical direction that P^* will make between consecutive regions. In particular, we know that each region is one of the following:

- A single point where the horizontal and vertical directions both change.
- A horizontal edge where the vertical direction changes, but not the horizontal.
- A vertical edge where the horizontal direction changes, but not the vertical.
- A square where the horizontal and vertical directions both do not change.

The transformed sequence for the example is shown in Figure 11.4.

11.3 Solving the Transformed Problem

Now let $\mathcal{R} = \langle R_0, R_1, \ldots, R_{n-1} \rangle$ be the circular sequence of regions that the polygon must go through. Let p_i denote the point in R_i visited by the polygon, and assume the numbering of the regions is done in such a way that p_0 lies on a given vertical line, say $x = 0$ (so the transformed region is either a vertical line segment or a point). Assume further that after R_0, the optimal solution moves to the right and down; the

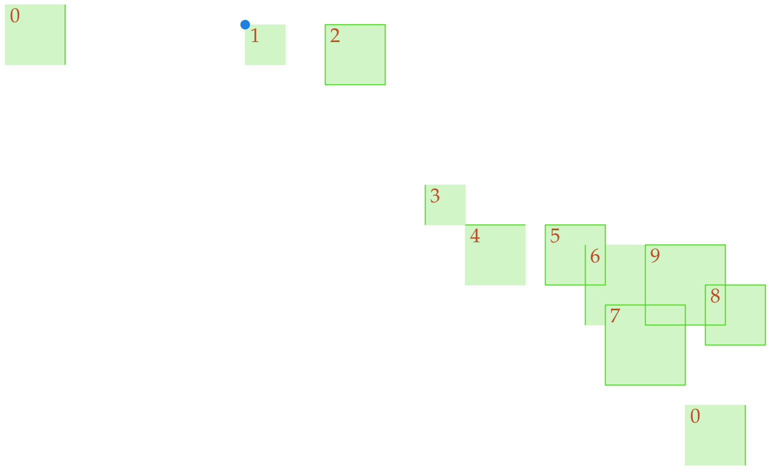


Figure 11.5 A sequence of regions with an xy-monotone optimal solution.

other cases are symmetric. We will transform our sequence of regions into another one, which is no longer circular, and for which the optimal solution is both x- and y-monotone, until it reaches a region R_n that is a translated copy of R_0.

To achieve this, note that whenever the polygon reaches a region R_i that is a horizontal line segment, the y-coordinate y_i of p_i is fixed and the polygon changes its vertical direction here. Now if we take the regions $\{R_j \mid j > i\}$ and mirror them in the line $y = y_i$, the perimeter of the optimal solution does not change. In the same way, if R_i is a vertical line segment we can mirror in the line $x = x_i$, and if it is a point we can mirror in both lines. If we do this for all regions R_i, the optimal solution is an xy-monotone path. Since the number of horizontal and vertical changes in direction of P^* is even, R_n will have the same orientation as R_0 in the end. Figure 11.5 shows the result of this procedure for the regions in Figure 11.4.

We are now looking for the shortest xy-monotone path that passes through a sequence of segments and squares in order, with the restriction that the points p_0 and p_n should be on the same place on R_0 and R_n.

We can simplify the regions a bit more, since we know that the path will be xy-monotone. If we have two regions R_i and R_j with $i < j$, then any part of R_i that extends farther to the right or bottom than R_j can never be used, so we can cut off those parts. This can be done incrementally in linear time, by starting at R_n and keeping track of the rightmost and lowest x- and y-coordinates reachable so far. Symmetrically, we cut off any parts of R_j that extend farther to the left or top than R_i. An example of the result can be seen in Figure 11.6.

Now, for each (clipped) region R_i, define the points b_i and t_i as the bottom left and top right corners of R_i. These points for all i form two polygonal lines T and B, and

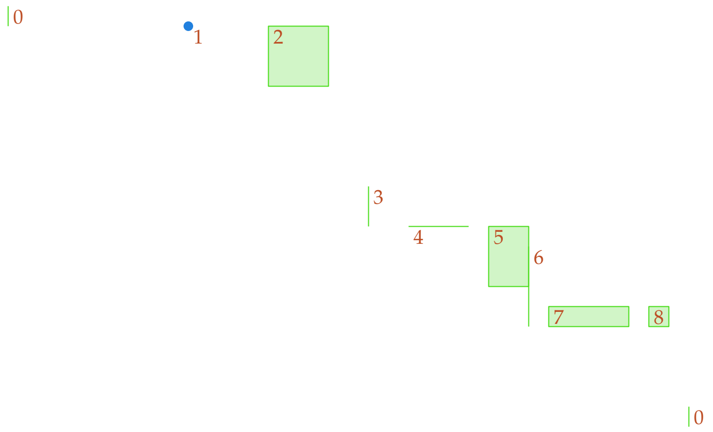


Figure 11.6 The regions clipped to their feasible parts. Regions 8 and 9 coincide.

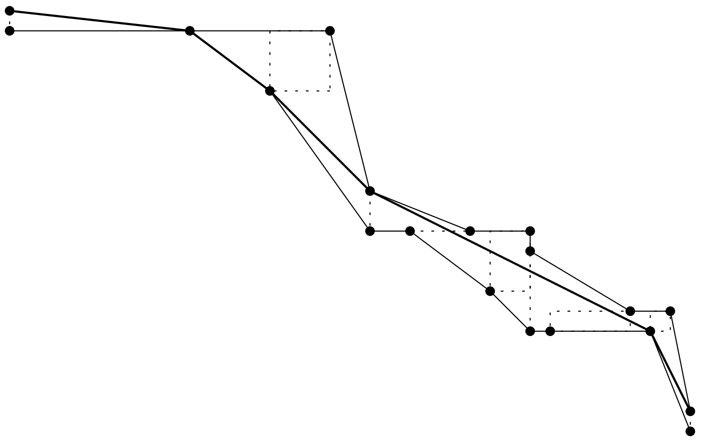


Figure 11.7 The regions that the polygon must pass through form a "cylindrical" tunnel, where the openings at the beginning and the end are identified. The shortest path through the tunnel is shown in bold.

because of the previous clipping, they are both xy-monotone. Furthermore, they form a tunnel such that any other xy-monotone polygonal path that goes through this tunnel corresponds exactly to a solution to our problem. Figure 11.7 shows the final tunnel and the shortest path through it for the example.

To find the shortest path through the tunnel, the only tricky part is that p_0 and p_n

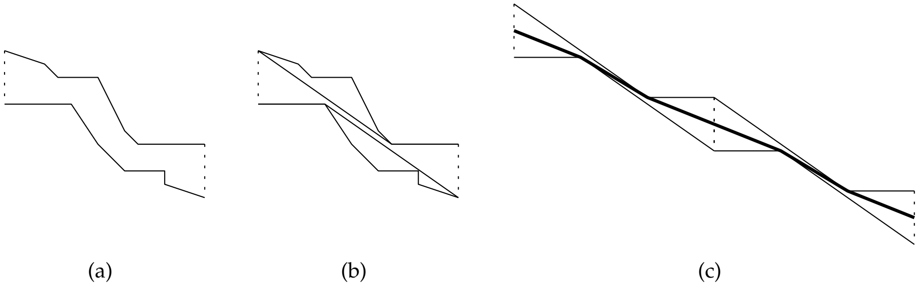


Figure 11.8 (a) A tunnel. (b) The funnel induced by the shortest paths between the top points and bottom points of the boundary intervals. (c) By taking two copies of the funnel, we can compute the shortest path.

have to be chosen at the same place on their intervals. If R_0 is a single point, the problem is just a shortest path inside a simple polygon, which can be solved in linear time [80]. Otherwise, we have to do something more.

Compute the shortest path inside the corridor from t_0 to t_n, and from b_0 to b_n. These two paths form the walls of a narrower corridor through which the optimal solution will go. If the floor and ceiling of this narrower corridor touch anywhere, then we can use that point as a starting point and repeat the whole procedure. If they do not, then they form a *funnel* with a convex ceiling and a concave floor, as shown in Figures 11.8(a) and 11.8(b).

The optimal solution is now either a straight line with the same slope as t_0t_n (possibly there are many such lines, in which case there are many equivalent optimal solutions), or it touches both the ceiling and the floor of the funnel. We can easily test whether the first case is possible in linear time. In the second case, we observe that since the regions R_0 and R_n are identified, we can glue two copies of the funnel together and the optimal solution has to move through the connection in a straight line, as in Figure 11.8(c). If we would know a point where the solution touches the floor, we could simply compute the shortest path between this point and the copy of this point in the copy of the funnel. Now note that if we take the lowest line parallel to t_0t_n that touches the floor, the point where it touches must be on the shortest path. We can find this point in linear time, and then compute the shortest path, also in linear time.

Theorem 11.2 *Given an ordered set of n arbitrarily sized, axis-aligned squares, the problem of choosing a point in each square such that the perimeter of the resulting polygon is as short as possible can be solved in $O(n)$ time.*

11.4 Closing Remarks

In this chapter, we studied the problem of computing the upper and lower bounds on the perimeter of a polygonal chain or a polygon, when the vertices of the polygon are imprecise points modelled as squares. We presented linear-time algorithms for both cases. The algorithms heavily make use of the fact that the regions are squares. For general polygonal regions, an algorithm that runs in $O(n^2 \log n)$ time is already available, although it requires a fixed starting point to be known [37].

The results of this chapter also appeared in [84], together with the results in Chapter 12.

Chapter Twelve

Simpleness of a Polygon

In this chapter, we study the implications of imprecision on the concept of *simpleness*[1] of polygons. Often, a polygon is known from the context to be simple (that is, to have no self-intersections or holes), for example when the polygon describes the boundary of a country, or any boundary of a piece of some flat surface. Then, when the points of the polygon are imprecise, the first thing to do would be to find a placement of the points such that the resulting polygon is indeed simple. Or, when there is no such meta-information available, we might want to know whether the given polygon is, or is not, simple. Since the input is imprecise, there are three possible answers to this question: yes, no, or maybe.

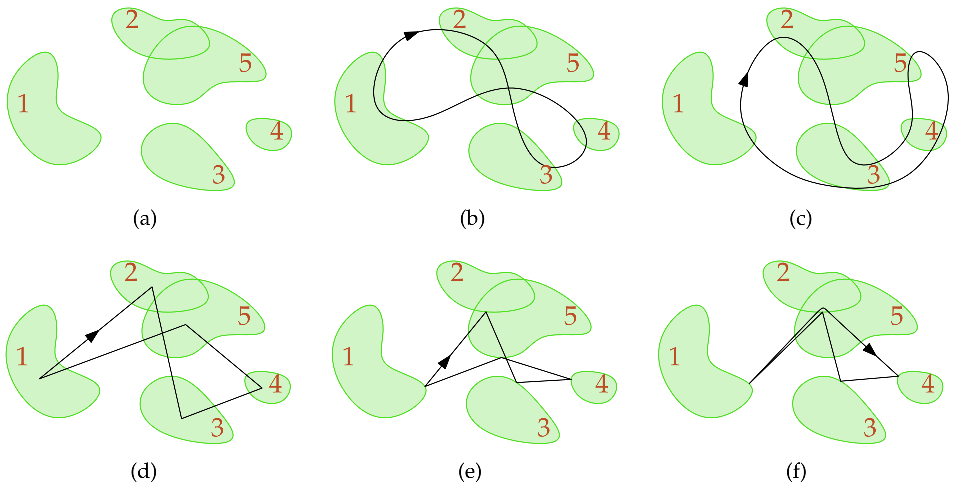
From now on, we consider a slight generalisation of polygons. We define a *tour* of a sequence of points to be any closed curve that passes through the points in the correct order.

More formally, we study the following problem. We are given a sequence $\mathcal{R}$ of n (possibly overlapping) connected regions in the plane. We are looking for a tour (closed curve) that visits all regions of $\mathcal{R}$ in the correct order; that is, we want to pick one point from each region such that the tour goes through those points in the correct order.

We call such a tour *straight* if the connections between consecutive points are straight line segments. In this case it is a polygon with one vertex in each region, and no other vertices; this corresponds to the model of the previous chapter. We also call the sequence $\mathcal{R}$ an *imprecise tour*, or when it is not cyclic an *imprecise path*. When the goal is to compute a *straight* tour or path, we will also call them *imprecise polygon* or *imprecise chain*.

---

[1] We use the term simpleness, not simplicity, since the meaning of the word as used in computational geometry is quite specific and only loosely related to the standard English word.



(a) (b) (c)

(d) (e) (f)

Figure 12.1 (a) Five regions and an order on them. (b) A tour visiting the regions in order. (c) A simple tour visiting the regions in order. (d) A straight tour visiting the regions in order. (e) The shortest tour visiting the regions in order. (f) The shortest simple tour visiting the regions in order (the tour is drawn loosely around the corners for better visibility).

We show in this chapter that deciding whether an imprecise tour admits a straight instance (that is, a polygon), without self-intersections, is NP-hard. Furthermore, the proof extends to several other situations concerning tours, which are also proven to be NP-hard.

12.1 Simpleness and Degenerate Simpleness

We call a tour *simple* if it does not cross itself. We are interested in the existence of a simple straight tour. Figure 12.1 shows an example of an ordered set of regions and some tours through them.

As mentioned, we prove here that determining whether a simple straight tour exists is NP-hard. When we do not force straightness, though, a simple tour always exists. In this case is becomes interesting to look for the *shortest* simple tour. For non-simple tours, it is easy to see that the shortest one is always straight. On the other hand, if we want to find a simple tour, the shortest one is not always straight. We also prove that finding the shortest simple tour is NP-hard; this answers an open question posed by Polishchuk and Mitchell [109].

There are some subtleties involved with simpleness, though. We mentioned that a

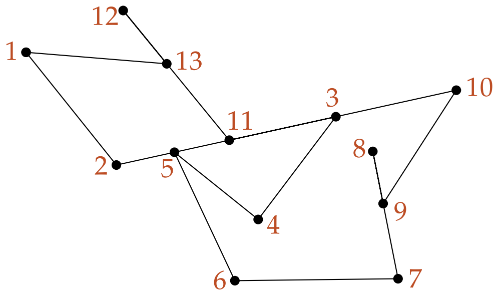


Figure 12.2 A degenerate simple polygon can touch itself in vertices or along edges, but has no crossings. The numbers give the order of the vertices.

simple curve is a curve that does not self-intersect. However, if we insist on simpleness when minimising the length of a tour, the optimum might not exist. For example, in Figure 12.1(f), we can keep making the tour shorter by moving the vertex in region 5 arbitrarily close to the vertex in region 2.

Instead, we will look for *degenerate* simple tours. A degenerate simple curve is a curve that is allowed to touch itself, but not to cross itself. Formally, a curve is degenerately simple if for any $\varepsilon > 0$ we can move each point of the curve over some distance at most ε such that the curve becomes simple. Figure 12.2 shows an example of a degenerate simple polygon.

The shortest degenerate simple tour is always defined, and is always a polygonal tour. Furthermore, if the input regions are in general position (e.g., they are polygons and no three defining vertices are on a line), then the shortest degenerate simple tour is in fact the limit of the shortest simple tour. To make the shortest simple tour well-defined, and for ease of description, we will use only degenerate simpleness in the remainder.

12.2 Simple Straight Tours through Vertical Line Segments

We will study the following problem in this section:

Problem 12.1 *Given a set of parallel line segments and a cyclic order on them, choose a point on each segment such that the polygon determined by those points in the given order is simple.*

We will show that deciding whether this is possible is NP-hard. We prove NP-hardness by reduction from planar 3-SAT [82]. A planar 3-SAT instance is a planar graph, consisting of variable vertices, clause vertices, and edges connecting variables

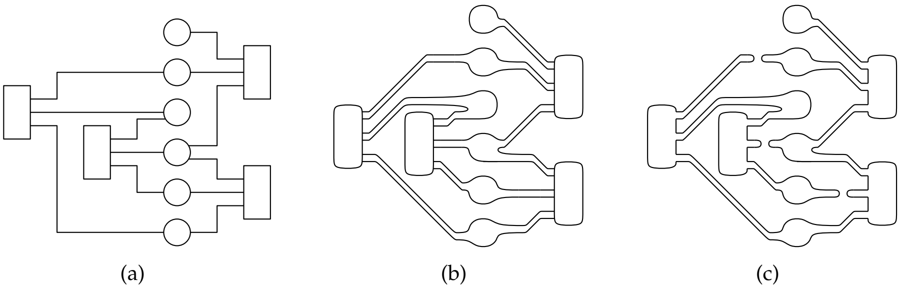

(a) (b) (c)

Figure 12.3 (a) An instance of planar 3-SAT. The circles represent variables, the rectangles represent clauses. (b) The variables and edges have been replaced by variable tentacle trees. (c) In order to make a single tour around the construction, we need to cut some of the variable tentacles. The construction will be done in such a way that the truth assignment on one side of the cut will be transferred to the other side.

to the clauses they appear in, see Figure 12.3(a). We will represent these vertices and edges by imprecise polygonal chains, that are finally connected into an imprecise polygon. Then, a simple polygon can be realised if and only if the 3-SAT instance is satisfiable.

Before going into the construction, we will give a short overview of the overall transformation. Instead of building variable vertices, clause vertices, and edges, we build only two things: variable chains and clauses. For this, replace each variable and all its outgoing edges by a tree-like structure, with tentacles to all clauses that use the variable, as in Figure 12.3(b). We will build these variable chains by having two imprecise polygonal chains alongside each tentacle. Then we connect those chains together as they meet at the clauses. However, in this way we will create many smaller imprecise polygons rather than one big one. Therefore, we need the imprecise polygon to bridge over some of the variable chains, as shown in Figure 12.3(c).

12.2.1 Variables

We represent variables by *scissor gadgets*. A scissor gadget is a construction of two imprecise points and two precise (or degenerate imprecise) points that should be visited in a given order. This will later become part of the input of Problem 12.1. We introduce a very small constant δ, and place the points as follows:

1. A precise point at $(-1, 1)$.
2. An imprecise point as a segment from $(1-\delta, -1+\delta)$ to $(1-\delta, 1-\delta)$.
3. An imprecise point as a segment from $(-1+\delta, -1+\delta)$ to $(-1+\delta, 1-\delta)$.

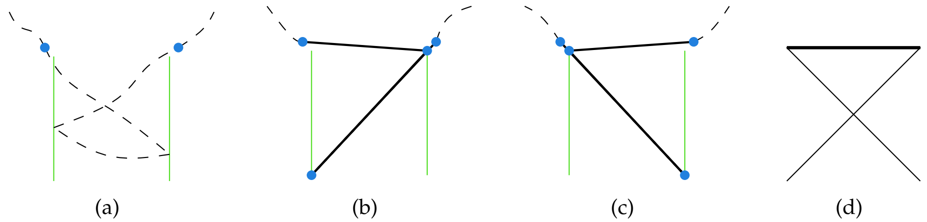


Figure 12.4 (a) The input for a scissor gadget. (b) One of the solutions, representing the state `true`. (c) The other solution, representing the state `false`. (d) Schematic representation.

4. A precise point at $(1, 1)$.

Figure 12.4(a) shows this construction. The black points are the precise points, the line segments are the imprecise points, and the dashed curves depict the order in which the tour should visit these regions. There are two possible ways to make a simple straight tour through this gadget, which represent the two different values of a variable.

One solution is to use the top of the first imprecise point (point 2), and then the bottom of the second imprecise point (point 3). In this case, the last two segments of the tour will coincide but there is no crossing. This situation is shown in Figure 12.4(b), and will represent the value `true` of a variable of the 3-SAT instance. The symmetric situation is also possible, see Figure 12.4(c); this will represent the value `false` of the variable. If we choose any other points in the imprecise points, the tour will cross, so these are the only two possible solutions.

In the remainder of the proof we use a schematic drawing for this configuration of four imprecise points, see Figure 12.4(d). The two diagonal legs in this drawing are places where some part of the tour could be, depending on the value of the variable, and the horizontal base is a place where some part of the tour always is (approximately, depending on δ). The ends of the base are the precise points in this small imprecise chain, and they will need to be connected to the other imprecise chains later to form a valid imprecise polygon.

It is possible to scale, translate and rotate the scissor gadget by $180°$ without changing the fact that it has exactly two solutions. If we place two copies in such a way that the positively sloped leg of the first intersects the negatively sloped leg of the second, and the negatively sloped leg of the first intersects the positive leg of the second, the scissor gadgets become *linked*: if one is in the `true` state, then so is the other, and vice versa. Figure 12.5 shows an example. We need to take care that the intersections between the legs of the scissor gadgets are more than δ away from the endpoints of the legs. Also, the legs should not intersect the base of the other scissor gadget, since then no solution is possible at all.

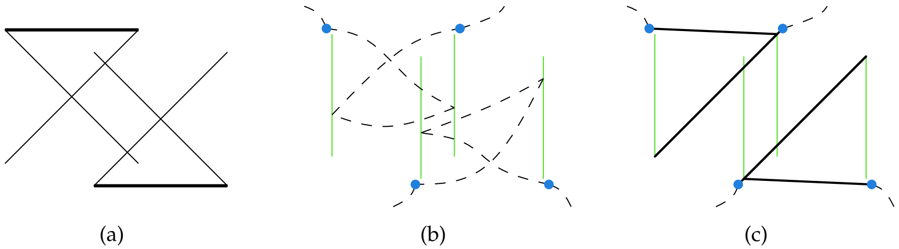


Figure 12.5 (a) Two scissor gadgets linked together. (b) The real situation corresponding to the linked scissor gadgets. (c) One possible solution.

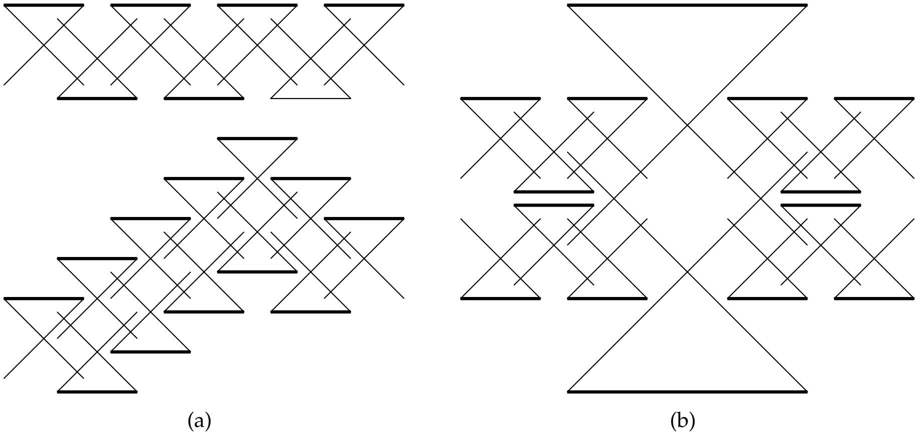


Figure 12.6 (a) Scissor gadgets can be chained together horizontally or diagonally. (b) A junction to split the chain of scissor gadgets.

We can now make a chain of scissor gadgets that all represent the same variable by linking them, as shown in Figure 12.6(a). There are only two possible states to this chain; either all of the scissor gadgets use their positively sloped leg or they all use their negatively sloped leg.

We can also split this chain into more chains with a junction as shown in Figure 12.6(b). In this configuration, the two big scissor gadgets are linked together. Since the two big ones either both use their positively sloped legs or both use their negatively sloped legs, we can then link a small scissor gadget to the group by letting its positively sloped leg intersect the negatively sloped leg of one of the two big ones, and letting its negatively sloped leg intersect the positively sloped leg of the other big one. This way, we can link four chains to the junction. We can also make sloped chains, see Figure 12.6(a). This way we can branch off a variable chain as often as needed for the

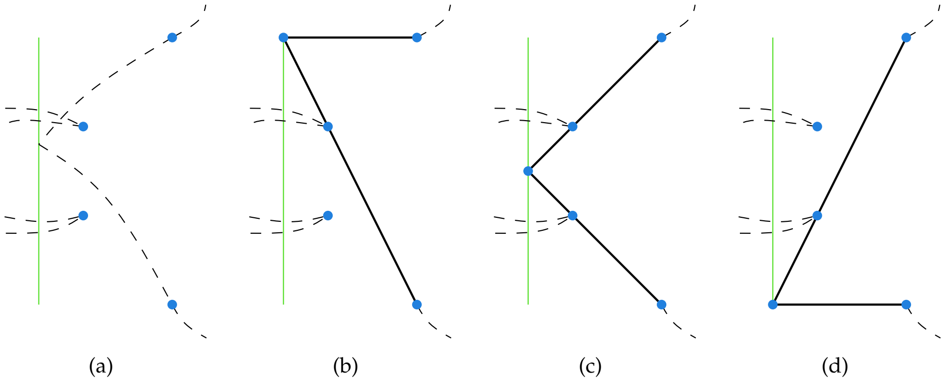


Figure 12.7 (a) The input for a clause. (b) One of the solutions. (c) Another solution. (d) The third solution.

3-SAT construction, and position the chains in the planar drawing of the instance as in Figure 12.3(b).

12.2.2 Clauses

We represent the clauses of the 3-SAT formula by *clause gadgets*. A clause gadget consists of one imprecise point and four precise points. For clauses, there are three unconnected polygonal chains that visit the gadget. The main chain that visits the gadget visits two of the precise points and the imprecise one, as follows:

1. A precise point at $(3,3)$.
2. An imprecise point as a segment from $(0,3)$ to $(0,-3)$.
3. A precise point at $(3,-3)$.

The other two chains visit only one precise point, one at $(1,1)$ and the other at $(1,-1)$. Figure 12.7(a) shows this situation, where the three chains are represented by dashed curves. The three possible solutions for this situation can be seen in Figures 12.7(b), 12.7(c) and 12.7(d), where the imprecise point is chosen at $(0,3)$, $(0,0)$ or $(0,-3)$ respectively. These are the only possible solutions. For the clause gadget we will also use a schematic representation, see Figure 12.8(b).

The idea is that in order to find a global solution, at least one of the three solutions to the clause must be possible. To achieve this, we will connect the three variables that appear in the clause to the three possible solutions, making sure that if a variable is in the wrong state, the corresponding solution to the clause gadget is blocked. For example, if we want to build the clause $a \vee b \vee \neg c$, we intersect the negatively

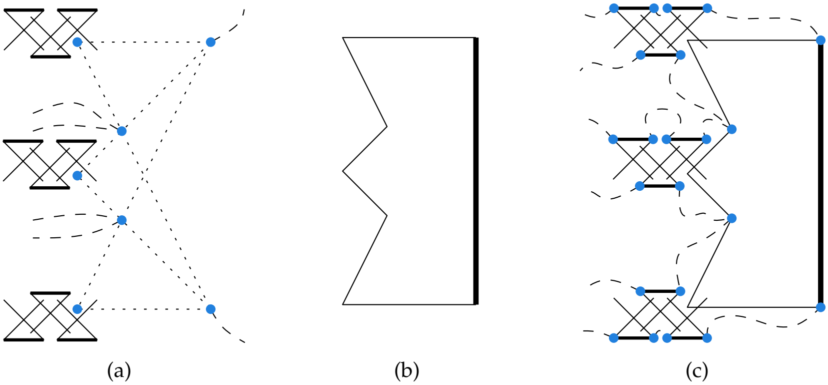


Figure 12.8 (a) The clause attached to the three variables. (b) Schematic representation. (c) We can connect the gadgets by pieces of the tour that consist only of precise points. The dashed curves show the connections.

sloped leg of a scissor gadget belonging to the chain of variable a with one of the three solution paths of the clause, a negatively sloped leg of the chain of variable b with another path, and finally a positively sloped leg of the chain of variable c with the remaining path, see Figure 12.8(a). Now a straight tour through the clause gadget is possible only if at least one of the variables is in the right state, which is exactly the case when the logical clause is satisfied. Note that in this construction a negation gadget is not needed.

12.2.3 Finishing the Construction

Now that we have structures for variables and clauses, we can build an instance of planar 3-SAT by embedding the graph in the plane and making it wide enough to fit all the structures such that they do not interfere. An example of a (part of a) resulting structure can be seen in Figure 12.9(a). However, this does not complete our construction yet.

The scissor and clause gadgets are defined as (sets of) imprecise polygonal chains, but the input to Problem 12.1 is an imprecise polygon. We need to construct an imprecise tour that visits all gadgets in any order, but in such a way that it does not interfere with the gadgets. We will do this by creating precise pieces of the polygon that will connect the imprecise pieces.

As long as the precise pieces of the tour do not intersect any of the schematic drawings of the gadgets in the construction, any possible tour through the gadgets is guaranteed to be still valid. We can easily do this by linking neighbouring gadgets together. To be precise, we link the scissors on both sides of a chain to each other, and at each

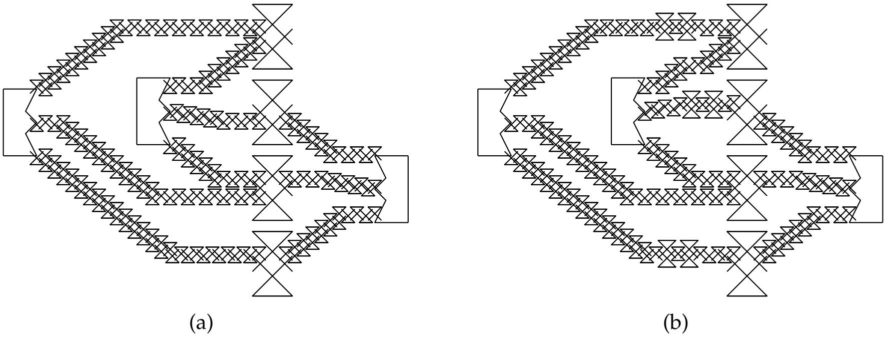


Figure 12.9 (a) Part of a network of variables and clauses to represent planar 3-SAT. (b) The network contains bridges to connect cycles.

clause we link the three incoming chains to the three tour parts of the clause gadget in order, as depicted in Figure 12.8(c). However, this will result in a number of smaller imprecise tours instead of one big imprecise tour, because the embedding of the 3-SAT instance partitions the plane into a number of faces. For a valid input to Problem 12.1, we need one tour to visit all gadgets, and therefore all faces. This means we need a way for the imprecise tour to cross the scissor chains.

For this purpose, we introduce the *bridge gadget*. A bridge gadget is a construction of two imprecise points and four precise points. It looks a lot like the scissor gadget, with the difference that there are now two pieces of the chain that visit the gadget, which both cross the gadget from above to below. We place the points on the two imprecise chains as follows:

1. A precise point at $(-1, 1)$.
2. An imprecise point as a segment from $(1-\delta, -1+\delta)$ to $(1-\delta, 1-\delta)$.
3. A precise point at $(-1, -1)$.

1. A precise point at $(1, 1)$.
2. An imprecise point as a segment from $(-1+\delta, -1+\delta)$ to $(-1+\delta, 1-\delta)$.
3. A precise point at $(1, -1)$.

Figure 12.10(a) shows this construction. There are again two possible ways to make a simple straight tour through this gadget, which represent the two different values of a variable. Either the left chain uses the top of its imprecise point and the right uses the bottom, see Figure 12.10(b), or the other way around, see Figure 12.10(c). These two situations will again represent the values `true` and `false` of a variable. A schematic representation is shown in Figure 12.10(d).

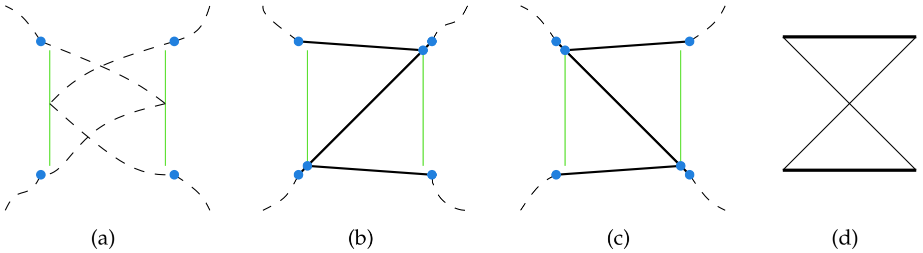


Figure 12.10 (a) The input for a bridge. (b) One of the solutions, representing the state `true`. (c) The other solution, representing the state `false`. (d) Schematic representation.

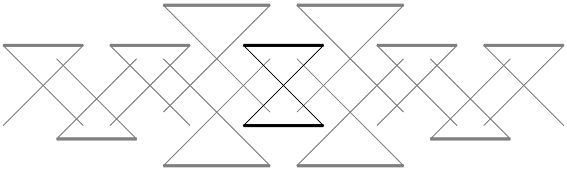


Figure 12.11 A bridge gadget embedded in a chain of scissor gadgets.

Bridge gadgets can also be linked to scissor gadgets. Since they do not have the pointed legs that the scissor gadgets have, we need to build a construction of scissor gadgets around them. As in the junction configuration, we use two bigger scissor gadgets that are linked together, and then we link the bridge gadget to this pair of scissor gadgets. This way they can be embedded in chains of scissor gadgets, and they preserve the property that the whole chain uses either positively or negatively sloped legs, see Figure 12.11. Now we have two imprecise polygonal chains that cross the variable chain.

Now we can include bridges into the network such that all faces of the embedded planar 3-SAT graph are connected by bridges, see Figure 12.9(b). All we need to do now is connect imprecise chains to each other with a fixed part of the tour (a part that contains only precise points), and we have a valid input for Problem 12.1. This imprecise tour allows for a simple straight tour with one vertex in each region, if and only if the 3-SAT instance can be satisfied.

We still need to set the value of δ appropriately. It should be small enough to make sure that all intersections of the scissor and bridge gadgets with each other and with the clause gadgets are guaranteed to be hit by the actual tours. However, this is just a local property, so the value of δ does not depend on n. The number of imprecise points in the construction is clearly polynomial in the length of the 3-SAT instance, which completes the proof.

Theorem 12.1 *Given an ordered set of n vertical line segments, the problem of deciding whether it is possible to choose a point on each segment such that the resulting polygon is simple is NP-hard.*

It is easy to adapt the gadgets slightly to also allow non-degenerate polygons, without damaging the proof. For the scissor gadgets, just make the vertical line segments slightly longer; for the clauses, move the two central precise points slightly towards the imprecise point. The amount they need to move should be small enough to avoid alternate solutions to the gadgets, but again this is only a local property so the value does not depend on n.

12.3 Simple Straight Tours through General Regions

In this section, we will generalise the NP-hardness proof to more general regions. The problem becomes:

Problem 12.2 *Given a set of scaled copies (homothets) of a given region and a cyclic order on them, choose a point in each region such that the polygon determined by those points in the given order is simple.*

We begin by noting that the proof of the previous section can easily be extended to sets of squares instead of vertical line segments. All segments in the construction are visited by a tour that enters and leaves the segment from the same side (left or right). If we extend the regions to the other side, this does not allow any alternative tours, so we can just replace the segments by squares that have the segments as one of their sides.

However, to extend the proof to arbitrary regions, we need to do some more work. We will first adapt the proof by using vertical axis-aligned rectangles of aspect ratio at most $\varepsilon : 1$ instead of vertical line segments, for suitably chosen ε. Then we will show that for any connected subregions of such rectangles, as long as the bounding box of those subregions coincides with the rectangles, the proof still holds. Finally, we show that we can scale the construction to make the proof hold for any set of connected regions in the plane.

12.3.1 Narrow Rectangles

The reduction from planar 3-SAT remains conceptually the same. In the three basic gadgets we replace line segments by narrow rectangles, see Figure 12.12. However, other than in the above description of the extension to squares, we now extend the segments *towards* the side where the tour is visiting them, to allow for subregions later. More precisely, the left region in the scissor gadget is the rectangle $(-1 + \delta, -1 + \delta + \varepsilon) \times (-1 + \delta, 1 - \delta)$ and the right region is the mirrored version. The regions in the

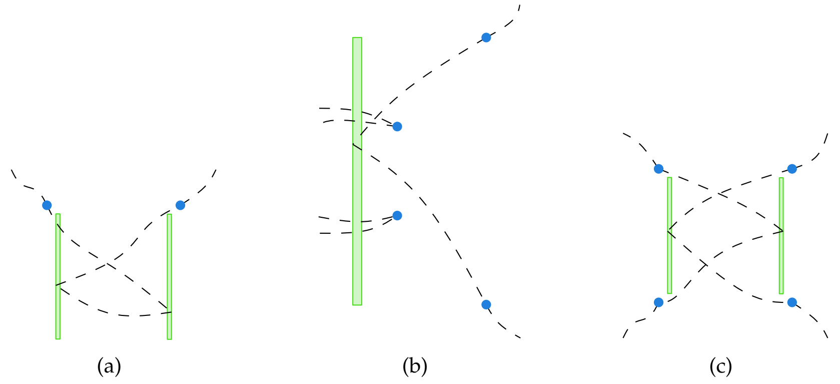


Figure 12.12 Replacing the line segments by narrow rectangles does not affect the gadgets. (a) An adapted scissor gadget. (b) An adapted clause gadget. (c) An adapted bridge gadget.

bridge gadget have the same coordinates as in the scissor gadget. Finally, the region in the clause gadget is the region $(0, \varepsilon) \times (-3, 3)$.

These rectangles allow for more solutions of the tours, by moving the points in the imprecise regions around. Clearly, the amount of freedom depends on ε. We have to make sure that the intersections between different gadgets in the construction are still guaranteed to lie on the tours. However, this is again a local property, so the value of ε does not depend on n.

This proves that, for some ε, the problem is still NP-hard for rectangles of aspect ratio $\varepsilon : 1$. Of course, even more narrow rectangles cause no problems, so a set of rectangles with varying aspect ratios bounded by $\varepsilon : 1$ is also allowed.

12.3.2 General Regions

Now suppose the regions are connected regions in the plane, with bounding boxes with aspect ratio at most $\varepsilon : 1$. We will place the bounding boxes just like the rectangles in the previous case. Since the regions are smaller, clearly there are still no unwanted solutions. However, we need to argue that the two or three solutions that are needed in the construction are still possible. But this is no problem: since the regions have the rectangles as bounding box, and are connected, there is at least one point inside each region at every height (y-coordinate). So for each solution of a gadget in Section 12.2, we can get a solution in the present situation by moving the point to another point at the same height. Since we only extended the regions towards the tours, this will still give a solution.

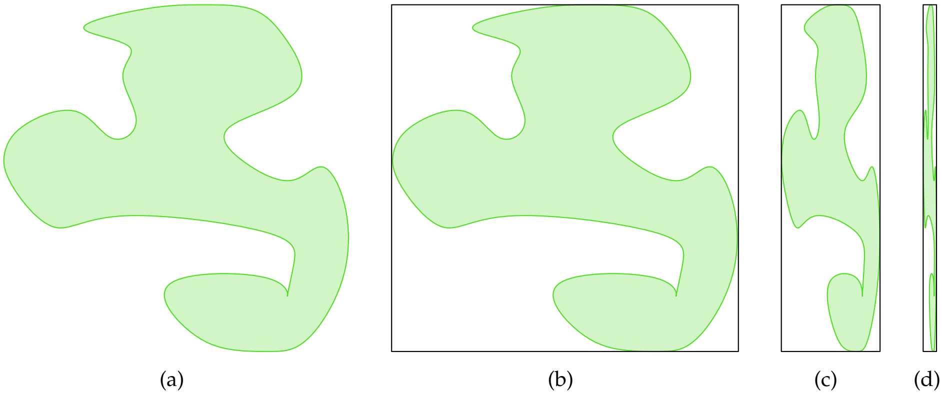

(a) (b) (c) (d)

Figure 12.13 (a) A general region. (b) The region in its bounding box. (c, d) The region scaled to a narrow rectangle.

If we model the points as scaled copies of any connected shape, for example as circles or regular k-gons, the proof can also be used after we scale the whole construction. Let r be the ratio between the width and height of any bounding box of a region. Now scale the plane in the x-direction with a factor $\frac{\varepsilon}{r}$, and the input has become a set of regions with aspect ratio $\varepsilon : 1$. Note that the existence of a simple tour is not affected by this scaling operation. The regions themselves now have narrow rectangles as bounding boxes, see Figure 12.13.

Theorem 12.2 *Given an ordered set of n scaled copies of any connected region, the problem of deciding whether it is possible to choose a point in each region such that the resulting polygon is simple is NP-hard.*

12.4 Shortest Simple Tours through Line Segments

If we drop the requirement that the edges between two consecutive points need to be straight line segments, a simple tour always exists. Take any disjoint point set from the regions, and just draw a curve to each consecutive point through the free space: this is always possible since the complement of the curve remains connected. In this context, it is interesting to consider *shortest* tours. We study the following problem:

Problem 12.3 *Given a set of axis-parallel line segments and a cyclic order on them, choose a point on each segment such that the length of the shortest simple tour passing through those points in the given order is minimised.*

As discussed in Section 12.1, we allow degenerate simple tours as well, so the minimum length tour is well-defined. We show here that finding this minimum tour

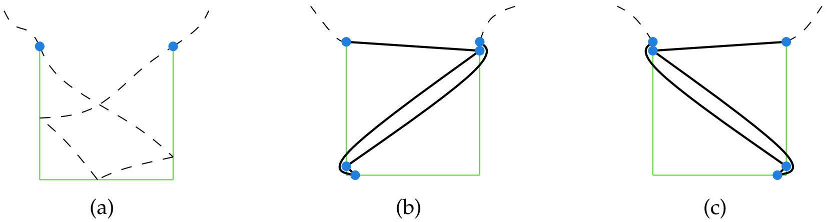


Figure 12.14 (a) Input for the adapted scissor gadget. (b) One of the two locally shortest solutions (the tour is drawn loosely around the corners for better visibility). This solution represents the value `true` of a variable. (c) The other locally shortest solution is symmetric to the first, and represents the value `false`.

is NP-hard, again by reduction from planar 3-SAT. The construction now requires horizontal line segments as well as vertical ones, which means we can no longer use the scaling argument of Section 12.3 to generalise the proof to work on input regions with any shape. However, we can still generalise it to axis-parallel squares or rectangles.

In this case, we need to make slightly more complicated gadgets. In the original scissor gadget, we implicitly used the fact that all connections need to be straight line segments, in order to ensure that the tour goes down to the bottom end of one of the two vertical line segments. When the connections do not need to be straight, we need to explicitly ensure that the tour goes down, so we add a horizontal segment. On the other hand, we no longer need the parameter δ (we can set it to zero). The scissor gadget now contains the following sequence of regions:

1. A precise point at $(-1, 1)$.
2. An imprecise point as a segment from $(1, -1)$ to $(1, 1)$.
3. An imprecise point as a segment from $(-1, -1)$ to $(1, -1)$.
4. An imprecise point as a segment from $(-1, -1)$ to $(-1, 1)$.
5. A precise point at $(1, 1)$.

Figure 12.14 shows this situation and two locally shortest solutions. The first solution uses the sequence of points $(-1, 1)$, $(1, 1)$, $(-1, -1)$, $(-1, -1)$, $(1, 1)$. The length of this tour is $1 + 2\sqrt{2}$. The symmetric solution has the same length, and both are locally optimal. It is not hard to see that any other solution is longer.

We also need to adapt the clause gadget. We now do not require the solutions to be straight, so there are always three locally optimal solutions: the tour can touch the segment above both of the two central precise points, between them, or below both of them. However, we need to ensure that all three solutions have exactly the same length. We can do this by moving the two central precise points. If we

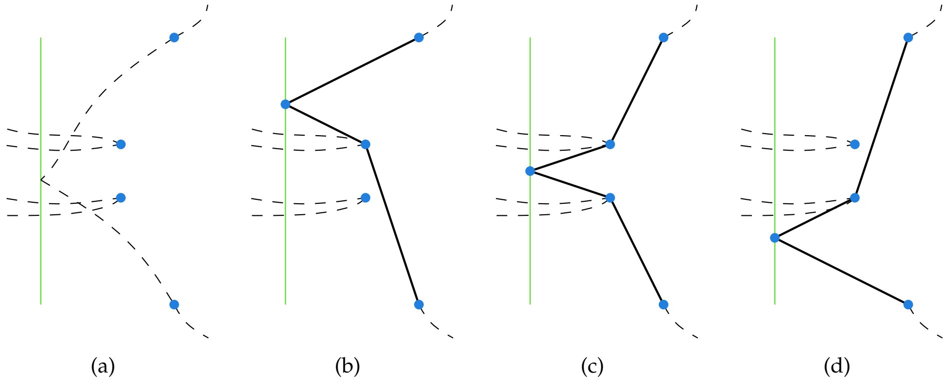

(a) (b) (c) (d)

Figure 12.15 Adapted clause.

place them at $(1.8, 0.6)$ and $(1.8, -0.6)$, and keep the rest of the construction as in Section 12.2.2, then all three solutions will have the same length, namely $2.4\sqrt{5} + 1.2\sqrt{10}$. Figure 12.15 shows the adapted gadget. The three solutions all touch the imprecise point in a different point, and we can connect the variables to the clause as before.

The bridge gadget does not need to be adapted, although also here we can set $\delta = 0$. Both solutions have a total length inside the gadget of $2 + 2\sqrt{2}$, and any other solution is longer.

With these adapted gadgets, we can build the same construction as before. Let ℓ be the length of the fixed part of the tour, plus $1 + 2\sqrt{2}$ for each scissor gadget in the construction, $2.4\sqrt{5} + 1.2\sqrt{10}$ for each clause gadget, and $2 + 2\sqrt{2}$ for each bridge gadget. Now a tour of length ℓ exists if and only if the 3-SAT instance is satisfiable.

As in Section 12.3, we can easily extend the proof to square regions by noting that in all gadgets the given segments might as well be sides of squares (or rectangles), without allowing any shorter solutions.

Theorem 12.3 *Given an ordered set of n axis-parallel line segments or squares, the problem of finding a minimum length simple tour that visits all segments or squares in order is NP-hard.*

12.5 Closing Remarks

In this chapter, we studied the properties of tours through a sequence of regions, with the idea that the regions represent imprecise points. We proved that it is NP-hard to decide whether it is possible to find such a tour that is both *simple* and *straight*. We also proved that it is NP-hard to find the shortest simple non-straight tour, resolving an open problem from [109].

The results given in this chapter also appeared in [84], together with the results from Chapter 11. The hardness results are all variations on the same construction. One remaining open problem in this style, which is slightly outside the scope of this thesis, is whether a shortest simple tour visiting a set of points, instead of regions, can be found efficiently. Regarding imprecise points, the question of ensuring simpleness of all instances of an imprecise object, or more generally, any property of such an object, is still open for many other types of objects, such as spanning trees, triangulations, or other geometric graphs.

Conclusions

The field of computational geometry is concerned with the design and analysis of geometric algorithms. For such algorithms, correctness and efficiency proofs are constructed, or problems are proven to be hard when no correct and efficient algorithms exist. In order to be able to do this, several assumptions about the input data for geometric algorithms are made. One of them is that this data is correct, with absolute certainty and infinite precision. In practical applications, this is often not the case, and as a result the value of these theoretical guarantees may be questionable.

If we want to supply geometric algorithms with theoretical guarantees that are actually observed in practice, we have to loosen our assumptions about the input data to a more realistic level. Depending on the application, we may be confident that each data point, for example, is not more than some value ε away from its given position. We can then construct algorithms that are guaranteed to be correct and efficient as long as the input satisfies this weaker assumption. Furthermore, we can analyse how the imprecision in the input influences the accuracy of the output.

In Chapter 1 of this thesis, we reviewed the history of computational geometry, and showed its intimate relation with the issue of data imprecision. In Chapter 2, we have given a classification of the kinds of algorithmic problems that may arise in this situation, as well as of the existing and new results that are now available. In the rest of the thesis, several concrete solutions to three specific classes of problems were discussed in detail.

Upper & Lower Bounds

In Part II of this thesis, we discussed geometric problems that have a single numeric value as output. We studied these under the imprecision model where each point is represented by a single region, usually a square or a disk, that contains the point. The output, in this case, becomes a set of possible values, and we presented algorithms to compute the lowest and highest possible values it can attain. This then provides tight bounds on the actual value of the output.

In Chapter 3, we considered several shape fitting measures on point sets. We studied the problems of how to compute upper and lower bounds on the smallest axis-aligned bounding box, the smallest enclosing circle, and the narrowest strip containing the set of imprecise points. In most of these cases, we were able to present efficient algorithms, with time bounds between $O(n)$ and $O(n^2)$. However, we also proved that computing the largest width of a set of imprecise points, modelled as line segments, is NP-hard.

In Chapter 4, we studied the problem of computing the largest or smallest convex hull of a set of imprecise points, measuring the size of a convex hull by its area. We gave an $O(n^2)$ time algorithm to compute the smallest area convex hull of a set of squares, without additional restrictions. On the other hand, for computing the upper bound we have an $O(n^7)$ time algorithm that works only for disjoint squares.

Since an $O(n^7)$ time result is not really useful in practice, in Chapter 5 we studied approximation algorithms for this problem. We successfully employed the *core-set* paradigm on sets of imprecise points to obtain $(1+\varepsilon)$-approximation algorithms for computationally hard problems. The dependence of the running time on the input size is linear and does not multiply with the dependence on ε, which makes the algorithms suitable for very large sets of imprecise points. On the other hand, the dependence on ε is often highly polynomial or exponential, which limits the achievable precision.

In Chapter 6, we considered the diameter of a set of points. We showed how to compute the upper bound in $O(n \log n)$ time for either the square or the disk model, using a relatively easy approach. Interestingly, the lower bound in the square model can also be computed in $O(n \log n)$ time, although the algorithm is considerably more involved. For the disk model, exact results are not feasible due to algebraic issues, but we did provide an approximation scheme that runs in $O(n^{c/\sqrt{\varepsilon}})$ time for any $\varepsilon > 0$ and a given constant c.

Future Work

The problems discussed in Part II provide only a sample from the available results of this kind. The papers those chapters are based on contain several more measures, imprecision models, variants, and results. Furthermore, new results have been obtained both by the author of this thesis and by other researchers working on imprecision concurrently.

Nonetheless, many problems are still open, and there are various directions of research to be pursued. To name an example of a concrete problem, how efficiently can the upper bound on the width of a set of imprecise points be computed? Or, what is the status of the problem of finding the largest convex hull when the squares are allowed to intersect? Also, certain results, such as the $O(n^7)$ time algorithm for disjoint squares, have not been proven to be optimal. Can these problems be solved more efficiently?

Apart from these concrete theoretical problems, there are numerous other geometric algorithms that provide a single number, and each of these could be studied in the presence of imprecision. This most likely leads to several interesting problems that are well worth studying.

Preprocessing for Triangulations

When a set of points is unknown, but constrained by a known region for each point, it is interesting to preprocess the regions to speed up computations when the exact locations of the points become known. In Part III of this thesis we studied problems of this kind. On the one hand, this provides us with practical algorithms when input data is imprecise but will be known with higher precision later, or when we want to analyse the data by sampling many point sets from the regions. On the other hand, these results also give new theoretical insight into the complexity of the well-studied precise problems, by narrowing down the parts of the problems that cause the intrinsic hardness of these problems.

In Chapter 7, we gave an algorithm to preprocess a set of disjoint regions in the plane in $O(n \log n)$ time, so that a sample from their regions can be triangulated in linear time. This time bound is optimal, and improves previous results by allowing more general regions. As the main method of our solution, we use an algorithm for splitting a triangulation in linear time. This is a very natural problem that we believe is interesting in its own right, and which improves over an earlier $O(n \log^* n)$ time algorithm.

In Chapter 8, we presented a similar result, but now we preprocess the regions in such a way that the *Delaunay* triangulation can be obtained in linear time, rather than any triangulation. To obtain this result, we must put more strict restrictions on the regions: they are required to be disjoint unit disks. In our solution, we collect enough structure of the output Delaunay triangulation to obtain a connected subgraph. The Delaunay triangulation can then be completed in linear time by Chazelle's and Chin and Wang's algorithms; however, these algorithms are complicated and make the result not useable in practice.

The results in Chapter 8 can also be extended to more general classes of regions than disjoint unit disks, but the dependency of the parameters that describe such regions is not optimal. In Chapter 9, we showed a different approach to reach the same result, which does not rely on the complicated algorithms by Chazelle and Chin and Wang and which allows for a better dependency on those parameters. However, the drawback is that the algorithms used in this method are randomised.

Future Work

The results in Part III show that for certain restricted types of regions in the plane, enough structure can be computed to speed up the computation of either some triangulation or the Delaunay triangulation. We also observed that such results are not possible to obtain for just any sets of regions. An interesting question is how much further the freedom in the regions can be stretched while still making such results possible.

On the other hand, the question naturally extends to geometric structures other than triangulations, such as spanning trees, planar tours, or geometric matchings, to name just a few. Results for some of these problems follow from the results on triangulations, but they have not been studied in depth. Also, higher-dimensional structures could allow for similar preprocessing algorithms. An interesting open question is whether some of the ideas in this thesis can be extended to higher dimensions.

Imprecise Lines and Polygons

In Part IV of this thesis we studied problems which take a set of lines or a composite geometric structure as input in an imprecise context, in particular polygons. The first question in this situation is how the imprecision in such objects can be modelled satisfactorily. For use in applications, it is important to guarantee the internal consistency of all possible instances of such an imprecise object. Once such a model is decided on, we can again compute bounds on output values.

In Chapter 10, we studied two well-known problems on line sets: linear programming and vertical extent. Both of these problems return a single value, and for both we have given efficient algorithms for computing the lower and upper bounds on this value. It seems that computing lower bounds on these problems is easier than computing upper bounds: we provided $O(n)$ respectively $O(n \log n)$ time algorithms that can compute the lower bound on these problems, while the upper bounds in general take $O(n^2)$ or $O(n^2 \log n)$ time respectively. However, under certain additional constraints on the input sets of lines, we show that these time bounds improve.

In Chapter 11, we considered the question of computing the longest and shortest instances of an imprecise polygon. We provided linear-time algorithms for both cases, when the imprecise points are modelled as squares. The algorithm for computing the longest polygon is quite straightforward, while the algorithm for the shortest polygon is more involved. It relies on the fact that the regions are squares by decomposing part of the problem into two independent 1-dimensional problems. These values together provide tight bounds on the interval of possible lengths of the imprecise polygon.

In Chapter 12, we considered the question of guaranteeing the property of *simpleness* of a polygon in the presence of imprecision. We proved that it is NP-hard to decide whether it is possible to find a tour through a set of regions in the plane that is both *simple* and *straight*. As a side-result of our construction, we also proved that it is NP-hard to find the shortest simple non-straight tour, resolving an open problem.

Future Work

The results in Chapter 10 are the first to work directly on imprecise lines. The results look promising, and should motivate further research into other geometric problems on lines. A concrete remaining open problem from this Chapter is whether the smallest vertical extent for general bundles can be computed any faster than in $O(n^2 \log n)$ time.

The results in Chapter 11 hold only for square regions, and the algorithms heavily make use of this. For general polygonal regions, an interesting open problem is whether the currently best known bound of $O(n^2 \log n)$ time can be improved. For circular regions, no results are known at all.

A long-standing open problem that is slightly outside of the scope of this thesis, but closely related to the results in Chapter 12, is the following. Given a sequence of *precise* points in the plane, what is the shortest simple tour that visits them in the given order?

Apart from these questions, the results in this part all apply to polygons. Similar questions can also be asked for other geometric graphs, such as triangulations, planar subdivisions, or higher-dimensional equivalents. There is a lot of room for further results in these areas.

Final Remarks

Data imprecision in computational geometry is an important issue that has long been ignored, but is now receiving an ever growing amount of attention. It is important to achieve a better understanding of the implications of imprecision on existing algorithms, before the application practitioner will adopt them and apply them to real-world problems. At the same time, with the enormous drop in cost of data storage and collection these days, the need for reliable and efficient algorithms is greater than ever.

In this thesis, we presented new algorithms for classical geometric problems, that explicitly take data imprecision into account. These algorithms cannot produce the real answers to those problems, but instead produce information about the possible values that the answers can have. In several cases, this can be done without adding any extra cost to the asymptotic running times of the classical solutions. In some cases, though, computing this information is significantly more costly than using classical algorithms, and in some cases we prove that indeed no efficient algorithms exist. Still, the research is far from finished, and it will be a challenge for science to build on this and other existing work.

However, care must also be taken. While the results that are available now apply to problems that are fundamental in geometry, the potential number of these problems is so large, that blindly embarking on a quest to solve them all would probably be doomed to fail. Perhaps the greatest challenge in this field now is to find the connection with actual practical applications, and to find out which types of results are really most needed or appreciated by the users of our algorithms.

Samenvatting

Dit proefschrift gaat over de effecten van *data-imprecisie* op problemen uit de *computationele meetkunde*. Computationele meetkunde is het vakgebied dat de correctheid van meetkundige algoritmes bestudeert. Zulke meetkundige algoritmes worden overal om ons heen gebruikt, en het is dus van belang dat ze correct werken. Een belangrijk nadeel aan de bewijzen uit de computationele meetkunde is echter dat vaak wordt aangenomen dat de invoer van de algoritmes correct is, met absolute zekerheid en oneindige nauwkeurigheid. Dit is in de werkelijkheid nooit het geval, en als gevolg hiervan kan de praktische waarde van deze bewijzen in twijfel worden getrokken.

In dit proefschrift presenteren we een aantal nieuwe meetkundige algoritmes, waarin we niet meer aannamen dat de invoer oneindig precies is, maar wel tot op zekere hoogte. Als gevolg hiervan is de uitvoer van deze algoritmes ook niet meer een precies antwoord of meetkundig object, maar eerder een beschrijving van wat de precieze uitvoer zou kunnen zijn.

Computationele Meetkunde

Meetkunde is één van de oudste wetenschappen die er bestaan. Het is de wetenschap van ruimtelijke vormen en de relaties daartussen: van punten, lijnen, cirkels, en van vlakken en bollen in hogere dimensies. De meetkunde die wij kennen is gebaseerd op de grondslagen van de oude Grieken; in het bijzonder op de *Elementen* van Euclides [44], die ongeveer 300 jaar voor Christus de toen bekende meetkunde heeft verzameld en opgeschreven. Sindsdien is de meetkunde steeds verder ontwikkeld, en in de afgelopen 2000 jaar zijn ontelbare interessante eigenschappen en relaties ontdekt in deze intrigerende tak van de wiskunde.

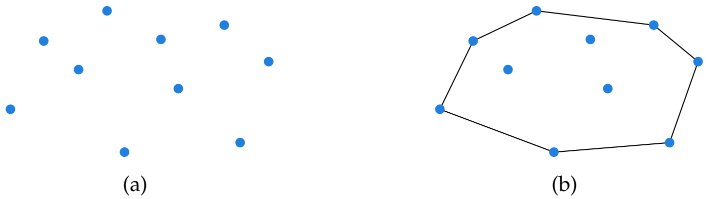
Maar meetkunde is waarschijnlijk al veel ouder dan dat, en is ontstaan om de wereld waarin we leven te kunnen meten, analyseren, en begrijpen. We leven in een ruimtelijke wereld, en de wil en noodzaak om die te begrijpen is de belangrijkste reden voor de voortdurende populariteit van de meetkunde. Maar, zoals het woord al zegt, voordat we die wereld kunnen begrijpen, zullen we hem eerst moeten meten: voorwerpen in de echte wereld die onze interesse hebben, moeten op de één of andere manier worden vertaald naar een wiskundig model, naar punten en lijnen met coördinaten in een Euclidische ruimte. Het probleem is dat dit meten van de wereld niet precies mogelijk is; in de volgende paragraaf gaan we hier dieper op in.

Een *algoritme* is een recept om iets uit te rekenen, een stap-voor-stap beschrijving om iets voor elkaar te krijgen, zonder dat je hoeft te begrijpen wat je doet. Een voorbeeld is het vermenigvuldigen van twee grote getallen. Er zijn verschillende manieren om dat te doen, maar de meeste bestaan uit het cijfer voor cijfer vermenigvuldigen, en de resultaten op de goede manier opschrijven en optellen. Je hoeft alleen maar te weten hoe je twee getallen van één cijfer moet vermenigvuldigen, en hoe je getallen moet optellen, om het algoritme uit te kunnen voeren. Het woord "algoritme" komt van de Perzische wiskundige Muhammad ibn Musa al-Khwārizmī, die rond het jaar 825 het getalsysteem wat wij tegenwoordig gebruiken in het westen introduceerde [5].

Algoritmes bestaan al heel lang, en waren ook al bij de oude Grieken bekend. Ze zijn echter pas echt populair geworden in de twintigste eeuw, toen de computer werd uitgevonden. Computers hebben als voordeel dat ze veel sneller werken dan mensen, en ze maken geen fouten. Aan de andere kant zijn het domme machines die niet begrijpen wat ze doen. Maar ze zijn heel geschikt om algoritmes mee uit te voeren. Een algoritme kan worden beschreven aan de hand van de *invoer* en *uitvoer*, een gebruiker hoeft niet te weten hoe het werkt maar alleen de juiste invoer aan te leveren en dan komt de juiste uitvoer eruit.

Omdat algoritmes steeds ingewikkelder worden en een gebruiker er op moet vertrouwen dat het doet wat het moet doen, gaan algoritmes vaak gepaard met een wiskundig *bewijs van correctheid*. Bovendien heeft een algoritme een *tijdscomplexiteit*: als een algoritme een grote hoeveelheid invoerdata krijgt, zeg n getallen, dan kan de tijd die het algoritme er over doet om de uitvoer te berekenen worden uitgedrukt als functie van n. We zijn hierbij vooral geïnteresseerd in het asymptotisch gedrag van deze functie, dat wil zeggen, hoe efficiënt het algoritme is voor grote waardes van n. Dit kan worden uitgedrukt met de zogenaamde *grote O*-notatie: bij een algoritme met een tijdscomplexiteit van $O(n)$ is de looptijd proportioneel met de invoergrootte, terwijl bijvoorbeeld bij een algoritme met complexiteit $O(n^2)$ de looptijd proportioneel is met het kwadraat van de invoergrootte.

Bewijzen van correctheid en tijdsanalyses worden bestudeerd in de tak van de informatica die *analyse van algoritmes* heet. De laatste jaren zijn de kosten voor het vergaren en opslaan van grote hoeveelheden data enorm gedaald. Om die data inzichtelijk te maken zijn algoritmes nodig, en het liefst algoritmes die snel grote hoeveelheden data kunnen verwerken. Als gevolg hiervan is de efficiëntie van algoritmes momenteel belangrijker dan ooit te voren.

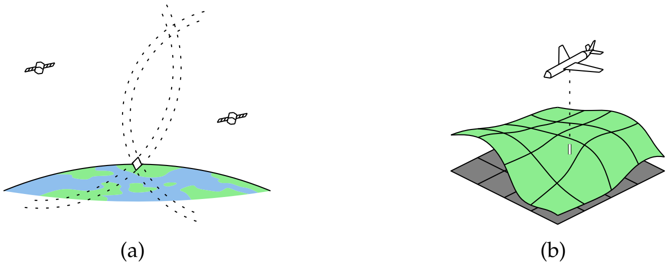


Figuur 16 (a) Een verzameling van tien punten in het vlak. (b) De convex hull van de punten.

Een *meetkundig algoritme* is een algoritme om een meetkundig probleem mee op te lossen. Vóór het tijdperk van de computers werden zulke algoritmes weinig gebruikt, want ze zijn ingewikkeld om met de hand uit te voeren, en aan de andere kant zijn mensen meestal heel goed in het "zien" van ruimtelijke eigenschappen. Maar sinds we computers gebruiken voor simulaties, spellen, ontwerp, etc. zijn meetkundige algoritmes nodig. Een klassiek voorbeeld van een meetkundig probleem is het berekenen van de zogenaamde *convex hull*, van een verzameling punten in het vlak. Dit is de kleinste convexe deelverzameling van het vlak die alle invoerpunten bevat. Een voorbeeld van een verzameling van tien punten is te zien in Figuur 16(a). Voor een mens is het makkelijk genoeg om te zien wat de convex hull van deze punten is; het antwoord is te zien in Figuur 16(b). Als we echter geen tien maar een miljoen punten hebben, dan hebben we een computer nodig om het probleem op te lossen.

Er bestaan vele toepassingsgebieden voor meetkundige algoritmes. In de meeste gevallen is er een duidelijke connectie tussen de Euclidische ruimte waarin de meetkundige problemen geformuleerd zijn, en de werkelijkheid waarin wij leven. In CAD (*Computer Aided Design*), bijvoorbeeld, worden echte voorwerpen ontworpen met behulp van een computer. In Geografische Informatiesystemen worden analyses uitgevoerd op een twee-dimensionaal model van (een gedeelte van) het aardoppervlak. In Computer Graphics wordt een artificiële 3-dimensionale wereld geprojecteerd op een twee-dimensionaal scherm, zodat het er uitziet als een mens het zou zien. In Moleculaire Biologie wordt de interactie tussen complexe moleculen bestudeerd met behulp van een drie-dimensionaal model. In al deze gevallen zijn meetkundige algoritmes nodig om de bewerkingen en analyses uit te kunnen voeren.

De eerste systematische analyse van de correctheid en tijdscomplexiteit van meetkundige algoritmes is gedaan door Michael Ian Shamos in zijn proefschrift *Computational Geometry* [120]. Door de meetkundige eigenschappen van deze problemen expliciet te bestuderen, is het vaak mogelijk om betere en snellere algoritmes te ontwerpen dan in de praktijk worden gebruikt. In de laatste dertig jaar zijn veel interessante resultaten bereikt. Echter, om zulke wiskundige bewijzen mogelijk te maken, moeten deze algoritmes wel aannemen dat de invoer correct is. Het feit dat dit in de praktijk niet altijd het geval is, is één van de redenen waarom deze algoritmes in sommige gevallen nog steeds niet zijn opgenomen in praktische toepassingen.

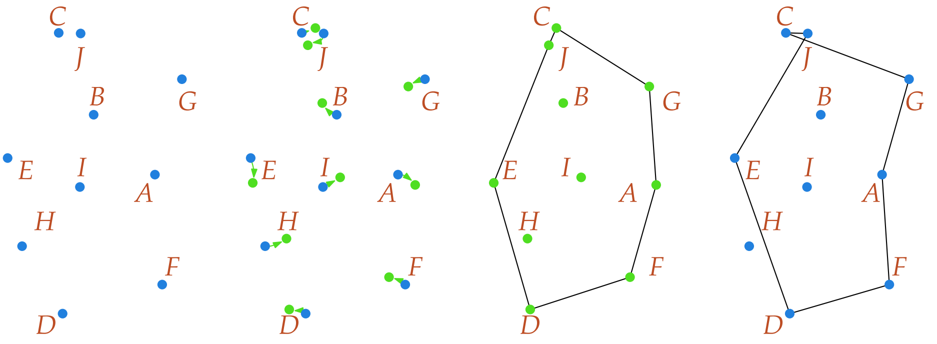

(a) (b)

Figuur 17 (a) Een locatie wordt door een GPS-ontvanger berekend door de afstand tot een aantal satellieten te meten. (b) In een LIDAR systeem schiet een vliegtuig laserstralen naar beneden om de afstand tot de grond te meten.

Imprecisie

Meetkunde is belangrijk omdat het een wiskundig model van de wereld om ons heen vormt. Om die wereld te analyseren, kunnen we metingen doen, en de resulterende gegevens met een meetkundig algoritme behandelen. Van zulke algoritmes kan wiskundig worden bewezen dat ze doen wat ze moeten doen. Toch wil dat niet zeggen dat het resultaat van een algoritme correct is, en dat komt doordat bij het meten van gegevens fouten worden gemaakt. Invoerdata wordt verzameld met meetinstrumenten, maar zulke instrumenten zijn nooit volledig precies. De precisie van meetinstrumenten neemt wel steeds meer toe, maar er zal altijd een fout blijven. Een andere oorzaak van imprecisie is dat het simpelweg niet mogelijk is om bepaalde continue data *overal* te meten, waardoor metingen geïnterpoleerd moeten worden. Bovendien kan in sommige toepassingen bewust worden gekozen voor data met minder hoge precisie, bijvoorbeeld door financiële overwegingen of met het oog op privacy. Dit alles resulteert in wat we *data-imprecisie* noemen.

Om een voorbeeld te noemen, een zeer polulaire manier om de locatie van een punt op aarde te bepalen is GPS (*Global Positioning System*). In dit systeem wordt een positie berekend door de afstand van het punt tot een aantal satellieten die om de aarde cirkelen te berekenen. Het punt kan dan worden gezien als het snijpunt van een aantal sferen in de ruimte, gecentreerd om de satellieten. Figuur 17(a) illustreert dit. Als de metingen echter niet precies zijn, resulteert dit niet in een enkel punt maar een gebied van mogelijke locaties. Een ander voorbeeld is het meten van hoogtedata. Zulke data wordt vaak verzameld door met een vliegtuig over een terrein te vliegen, en met regelmatige tussenpozen een laserstraal naar beneden te sturen en te meten hoe lang het duurt voor deze terugkomt. Deze techniek heet LIDAR (*Light Detection and Ranging*), en is geïllustreerd in Figuur 17(b). Ook hier geldt dat de meting niet precies is, bijvoorbeeld doordat de hoogte van het vliegtuig niet precies bekend is, of omdat voorwerpen op de grond de laserstraal verstoren.

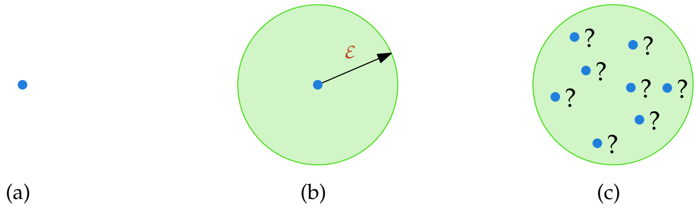


(a) (b) (c) (d)

Figuur 18 (a) Een verzameling punten in het vlak. (b) Een iets verschoven variant van de puntverzameling. (c) De convex hull van de verschoven punten. (d) Dezelfde hull, terugvertaald naar de originele punten.

Als algoritmes die bewijsbaar correct zijn worden toegepast op imprecieze data, dan kunnen de resultaten toch onverwacht zijn. Beschouw, als voorbeeld, de volgende situatie. Stel dat we een verzameling punten in een vlak in de echte wereld hebben, en we zijn geïnteresseerd in de convex hull van die punten. We gaan er vanuit dat de punten namen hebben van A tot en met I, zoals de verzameling in Figuur 18(a). Nu meten we de locaties van deze punten met een meetapparaat dat een kleine fout maakt. Het resultaat is een andere puntverzameling, die wel lijkt op de werkelijke verzameling maar net anders is, zie Figuur 18(b). Op deze gemeten verzameling laten we nu een bewijsbaar correct algoritme los, wat resulteert in de correcte convex hull van de gemeten punten, zoals weergegeven in Figuur 18(c). Zo'n hull kan compact worden weergegeven door de volgorde waarin de punten op de hull voorkomen op te slaan, in dit voorbeeld dus $D - E - J - C - G - A - F - D$. Echter, deze volgorde geeft helemaal niet de correcte hull weer van de punten in de werkelijkheid. Als we de werkelijke punten in deze volgorde verbinden krijgen we het resultaat in Figuur 18(d). Dit is duidelijk niet het gewenste resultaat.

Meetkundige algoritmes worden veel gebruikt in de praktijk. Deze algoritmes moeten wel omgaan met imprecisie, anders zouden ze niet werken. Vaak zijn dit echter heuristieken die in de praktijk zijn getest, en geen algoritmes met bewijsbaar correct gedrag. Computationele meetkunde is een relatief jonge tak van de informatica, en in de eerste jaren is vrijwel alle aandacht uitgegaan naar het ontwerpen van meetkundige algoritmes zelf, zonder veel acht te slaan op imprecisie. Langzaam maar zeker verschuift deze aandacht, onder druk van steeds grotere hoeveelheden imprecieze data die in de praktijk wordt verzameld. Inmiddels zijn er veel verschillende manieren om imprecisie te modelleren voorgesteld, en een aantal resultaten zijn al beschikbaar. Geen van deze methoden lost het probleem van data-imprecisie op: zolang data imprecies is zal het nooit mogelijk zijn om met volledige zekerheid een "correct"



(a) (b) (c)

Figuur 19 (a) Een traditioneel, precies punt. (b) Een imprecies punt, gemodelleerd als schijf met straal ε. (c) Het werkelijke punt kan overal binnen de schijf liggen.

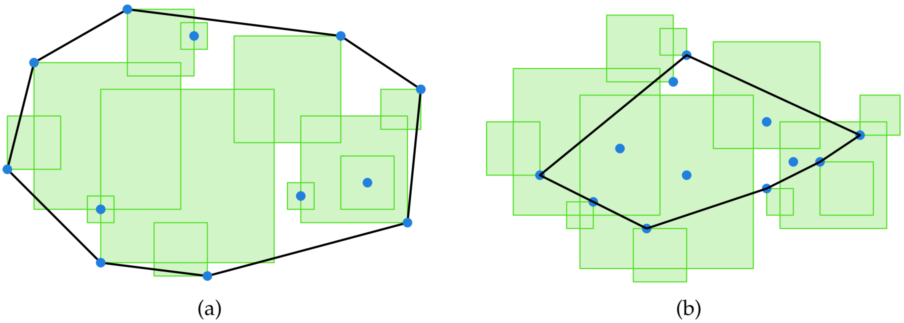
antwoord te geven op meetkundige vragen. Maar het is wel degelijk mogelijk om meer inzicht en informatie te berekenen dan het simpelweg negeren van het probleem.

Grenzen

In Deel II van dit proefschrift bestuderen we het probleem van het berekenen van boven- en ondergrenzen aan de uitkomst van een meetkundig algoritme, als de invoer een verzameling imprecieze punten is. Als een meetkundig probleem een enkel getal als uitvoer heeft, dan moet een traditioneel algoritme dat getal berekenen. Als de punten echter imprecies zijn, hangt de waarde van het antwoord af van waar de invoerpunten precies zijn. In dit geval hebben we algoritmes nodig die berekenen wat de hoogst mogelijke en laagst mogelijke waardes zijn die het antwoord kan hebben.

Om imprecisie in een punt te modelleren, zeggen we dat de invoer van een algoritme nu niet een verzameling punten in het vlak is, maar een verzameling *deelverzamelingen* van het vlak. Iedere deelverzameling, of gebied, stelt een imprecies punt voor. Dit betekent dat we weten dat het werkelijke punt ergens in dat gebied ligt, alleen niet waar. Dit model is voor het eerst geïntroduceert in 1989 door Salesin, Stolfi en Guibas [61], die om een precies punt een cirkelschijf met straal ε trokken om aan te geven dat het punt een fout van maximaal ε heeft. Figuur 1.6 toont zo'n imprecies punt. Behalve een schijf met straal ε, is het ook mogelijk om andere vormen als gebieden te gebruiken. Als bijvoorbeeld de imprecisie in de x- en y-coördinaat van een punt onafhankelijk zijn, ligt het meer voor de hand om een vierkant als gebied te gebruiken.

We nemen in de hoofdstukken in dit deel steeds aan dat er een "werkelijke" verzameling punten $P = \{p_1, \ldots, p_n\}$ is die onbekend is, waarbij $p_i \in \mathbb{R}^2$. In plaats daarvan hebben we een verzameling gebieden $\mathcal{R} = \{R_1, \ldots, R_n\}$ gegeven, met $R_i \subset \mathbb{R}^2$, en de garantie dat we voor elke i weten dat $p_i \in R_i$. We zijn geïnteresseerd in de waarde van een bepaalde functie μ die een verzameling punten als invoer heeft en een enkel getal als uitvoer, dat wil zeggen, we willen $\mu(P)$ weten. Maar omdat P onbekend is,


(a) (b)

Figuur 20 (a) Een verzameling imprecieze punten, gemodelleerd als vierkanten, met één punt in elk gebied zodat de convex hull van deze punten zo groot mogelijk is. (b) Dezelfde verzameling vierkanten, nu met één punt per gebied zodat de convex hull van de punten zo klein mogelijk is.

berekenen we in plaats daarvan de kleinste en grootste mogelijke waardes die μ kan aannemen op een puntverzameling P die conform is met $\mathcal{R}$.

In Hoofdstuk 3 beschouwen we drie zogenaamde *pasvormfuncties*. Dit zijn functies die een bepaalde meetkundige vorm zo goed mogelijk om een puntverzameling heen passen. Voor μ nemen we oppervlakte van de kleinste omvattende asparallelle rechthoek, de straal van de kleinste omvattende cirkel, en de breedte van de smalste strip (in een willekeurige richting) die de punten bevat. Als imprecisiegebieden hebben we vierkanten en cirkels bestudeerd. In alle gevallen hebben we het probleem van het berekenen van boven- en ondergrenzen bestudeerd, en in de meeste gevallen hebben we efficiënte algoritmes ontwikkeld, met looptijden tussen de $O(n)$ en $O(n^2)$. Het berekenen van de bovengrens voor de smalste strip lijkt echter een veel moeilijker probleem te zijn, waar we geen bevredigend resultaat voor hebben. Wel bewijzen we dat dit probleem NP-moeilijk is wanneer de imprecisiegebieden lijnstukken zijn.

In Hoofdstuk 4 bestuderen als functie μ de oppervlakte van de convex hull van de punten. We zijn nu geïnteresseerd in de grootste en kleinste mogelijke oppervlakte van de convex hull van een verzameling imprecieze punten, gemodelleerd als vierkanten. Figuur 20 toont een voorbeeld van dit probleem. De ondergrens kan worden berekend in $O(n^2)$ tijd, terwijl de bovengrens $O(n^7)$ tijd kost; bovendien werkt dat algoritme alleen voor niet-overlappende vierkanten. Aangezien dat niet een erg praktisch resultaat is, bestuderen we in Hoofdstuk 5 approximatie-algoritmes voor hetzelfde probleem. Deze algoritmes zijn gebaseerd op het idee van *kernverzamelingen*, wat is geïntroduceerd door Agarwal en Har-Peled [2]: in plaats van een duur algoritme op een grote invoerverzameling uit te voeren, is het vaak mogelijk om eerst een representatieve deelverzameling van de invoer te selecteren en alleen daarop het dure algoritme los te laten.

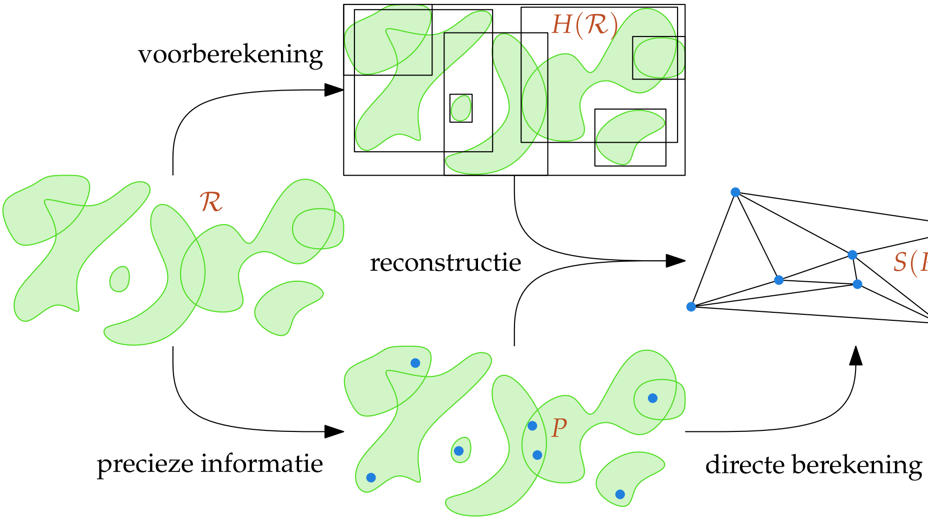
In Hoofdstuk 6 nemen we voor μ de diameter van een puntverzameling: de grootste afstand tussen twee punten in de verzameling. De diameter is een belangrijke maat voor de uitgestrektheid van een puntverzameling. We laten zien hoe de bovengrens in $O(n \log n)$ tijd kan worden berekend, zowel voor vierkanten als voor cirkelschijven. Interessant genoeg kan de ondergrens voor vierkanten ook in $O(n \log n)$ tijd berekend worden, terwijl dit voor cirkels op algebraïsche moeilijkheden stuit. Voor dat geval geven we wel een approximatie-algoritme.

Voorberekenen

In Deel III van dit proefschrift benaderen we data-imprecisie vanuit een andere richting. In plaats van grenzen te berekenen aan de waardes die een functie kan aannemen, gaan we er in dit deel vanuit dat alhoewel de punten op dit moment imprecies zijn, we deze punten later wel te weten zullen komen. In dit geval is de uitdaging om zoveel mogelijk van de moeilijke berekeningen die we moeten doen, nu al te doen, zodat we later, al we de “echte” punten krijgen en misschien niet veel tijd meer hebben, de berekening zo snel mogelijk kunnen afronden.

Het is natuurlijk wat optimistisch om te denken dat imprecieze punten later ineens precies zullen worden. Maar ook al is dit niet mogelijk, het is wel vaak mogelijk om zo nodig bepaalde metingen met hogere nauwkeurigheid over te doen. Een andere mogelijkheid is dat in de werkelijkheid we helemaal geen betere metingen doen, maar dat we een groot aantal mogelijke puntverzamelingen “raden”, door steeds één willekeurig punt uit elke gebied te nemen. Als we voor al deze puntverzamelingen onze berekening doen, kan op de uitkomsten een statistische analyse worden gedaan. In dit geval is het ook belangrijk dat de berekening op de echte punten zelf zo snel mogelijk is.

Ook vanuit een theoretisch oogpunt is het interessant om te weten hoeveel moeilijke, of op z’n minst tijdsintensieve, berekeningen van een algoritme al van te voren kunnen worden gedaan, en dus niet afhankelijk zijn van de exacte posities van de punten. Om dit precies te maken, hebben we weer een onbekende puntverzameling P en een verzameling gebieden $\mathcal{R}$ zodanig dat elk punt van P in één van die gebieden ligt. We zijn nu geïnteresseerd in een meetkundig object $S(P)$, bijvoorbeeld een *triangulatie* van de punten. We hebben een exact algoritme dat $S(P)$ kan berekenen als P gegeven is, en wat een bepaalde tijd kost, in het geval van triangulatie $O(n \log n)$. De vraag is nu of het mogelijk is om een tussenprodukt $H(\mathcal{R})$ te berekenen dat later kan helpen met de berekening van $S(P)$ wanneer P bekend is. We hebben nu twee nieuwe algoritmes nodig: een *voorberekeningsalgoritme* dat $H(\mathcal{R})$ kan berekenen, en een *reconstructiealgoritme* dat $S(P)$ kan berekenen als P en $H(\mathcal{R})$ gegeven zijn. Figuur 21 geeft een visuele illustratie van dit idee. Het moge duidelijk zijn dat beide algoritmes samen nooit sneller kunnen zijn dat het snelste algoritme om $S(P)$ direct te berekenen. De vraag is echter hoe snel we het reconstructiealgoritme kunnen

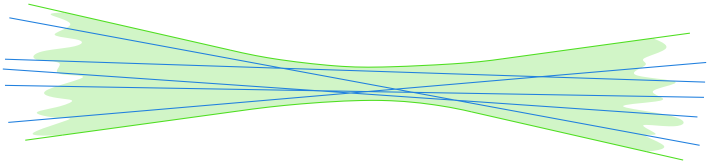


Figuur 21 Een schema dat de datastroom laat zien bij gebruik van voorberekenings- en reconstructiealgoritmes, in vergelijking met directe berekening.

krijgen, door zoveel mogelijk berekeningen naar het voorberekeningsalgoritme te verschuiven.

In Hoofdstuk 7 beschouwen we een triangulatie van P als de structuur $S(P)$. We modelleren de imprecieze punten in $\mathcal{R}$ als niet-overlappende polygonen, en laten zien dat het mogelijk is om deze voor te berekenen in $O(n \log n)$ tijd, zodanig dat het reconstructiealgoritme nog maar $O(n)$ tijd kost. Dit resultaat is optimaal, en is een verbetering van eerder werk door Held en Mitchell [65], die hetzelfde resultaat bewijzen voor een verzameling eenheidscirkels. Een belangrijk deelresultaat dat we gebruiken om het probleem op te lossen is een algoritme om een triangulatie te *splitsen*: gegeven een triangulatie waarvan de knopen rood en blauw gekleurd zijn, kan ons algoritme een triangulatie van alleen de blauwe (of de rode) knopen berekenen in $O(n)$ tijd.

In Hoofdstuk 8 geven we een soortgelijk resultaat, maar nu is $S(P)$ niet een willekeurige triangulatie maar specifiek de *Delaunay* triangulatie. Dit is een triangulatie die voor het eerst is beschreven door Delaunay [35], en een belangrijke plaats inneemt in de computationele meetkunde. Ook hier geven we een voorberekeningsalgoritme dat in $O(n \log n)$ tijd loopt en een reconstructiealgoritme dat $O(n)$ tijd nodig heeft, maar de imprecisiegebieden zijn minder algemeen: we nemen aan dat dit eenheidscirkels zijn. De oplossing is gebaseerd op de minimaal opspannende boom van de imprecieze punten, en we laten zien dat dit genoeg informatie is om een samenhangende deelverzameling van de Delaunay triangulatie te reconstrueren wanneer P gegeven is. Vervolgens kan de triangulatie worden voltooid met het algoritme van Chin en



Figuur 22 Een bundel lijnen. De bundel bevat alle lijnen die volledig binnen het grijze gebied liggen.

Wang [30]. Dit algoritme werkt in $O(n)$ tijd, maar is helaas wel nogal ingewikkeld en niet erg bruikbaar in de praktijk.

In Hoofdstuk 9 laten we zien hoe hetzelfde resultaat op een andere manier kan worden bereikt. De aanpak in dit hoofdstuk is gebaseerd op *quadtrees*: een subdivisie van het vlak volgens een boomstructuur. Deze methode is makkelijker te implementeren dan die in het vorige hoofdstuk, maar is wel gerandomiseerd, dat wil zeggen, de looptijd is niet gegarandeerd $O(n)$ bij iedere invoer maar de kans dat het niet zo is is (verwaarloosbaar) klein. Bovendien is dit resultaat op een optimale manier uit te breiden naar algemenere imprecisiegebieden, namelijk cirkels die niet even groot zijn of deels overlappen, of gebieden die geen cirkels zijn maar wel in zekere zin “dik”.

Meer dan Punten

Een verzameling punten is één van de meest voorkomende vormen van meetkundige invoer, maar er zijn ook andere mogelijkheden. Een algoritme kan ook een verzameling *lijnen* als invoer krijgen, of een samengestelde figuur die is opgebouwd uit meerdere punten en lijnen of lijnstukken. In Deel IV van dit proefschrift gaan we hier verder op in.

Een imprecieze lijn kan op dezelfde manier als een imprecies punt worden gemodelleerd: als een deelverzameling van de verzameling van alle lijnen in het vlak. Echter, we zouden graag wat extra eisen aan zo’n deelverzameling stellen om er goed mee te kunnen werken. Voor een verzameling punten zijn een cirkelschijf of een vierkant natuurlijke keuzes, maar voor lijnen zijn er geen equivalente begrippen. We definiëren daarom een *bundel* lijnen als de verzameling van alle lijnen die tussen twee gegeven curven in het vlak liggen. Figuur 22 toont een voorbeeld van zo’n bundel.

We eisen van de curven die de bundel omsluiten dat deze stuksgewijs lineair zijn, en dat het aantal stukken constant is. De verzameling lijnen gegeven door een bundel is samenhangend, en bovendien in zekere zin *convex*. Convexiteit is niet eenduidig gedefiniëerd voor verzamelingen lijnen, hoewel er verschillende definities zijn voorgesteld in de literatuur [51, 54, 114]. Als we echter aannemen dat een imprecieze lijn in minstens één richting zeker *niet* ligt, dan vallen een aantal definities samen en voldoen onze bundels daaraan.

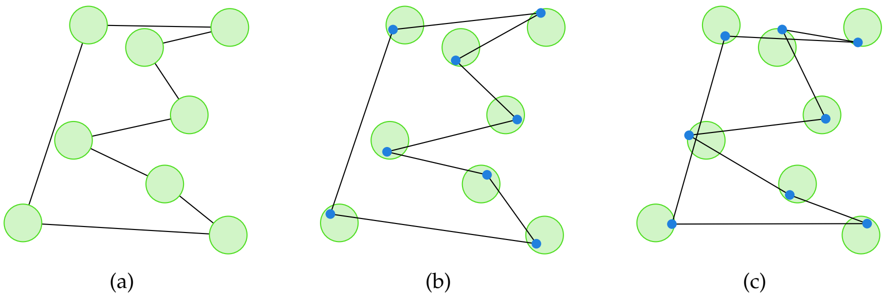
In Hoofdstuk 10 bestuderen we twee concrete problemen die een verzameling lijnen als invoer hebben in dit model. Bij *lineair programmeren* het het de bedoeling om het laagste punt dat boven een verzameling lijnen ligt te vinden. We laten zien dat de bovengrens van deze waarde wanneer de lijnen imprecies zijn in lineaire tijd kan worden berekend. De ondergrens kan in $O(n^2)$ time berekend worden, maar in sommige specifieke gevallen kan het ook sneller. Het andere probleem is de *verticale omvang* van een verzameling lijnen: het kortste verticale lijnstuk dat een gegeven verzameling lijnen snijdt. Wanneer de lijnen imprecies zijn, kunnen we de bovengrens voor de omvang in $O(n \log n)$ tijd berekenen. De ondergrens kan worden berekend in $O(n^2 \log n)$ tijd, en onder bepaalde voorwaarden ook in $O(n \log n)$ tijd.

Wanneer de invoer van een meetkundig probleem een *samengestelde* figuur is, is het minder duidelijk hoe imprecisie op een goede manier gemodelleerd kan worden. Eén van de eenvoudigste samengestelde figuren is de veelhoek of *polygoon*: een reeks punten in het vlak die door lijnstukken worden verbonden. Andere voorbeelden van samengestelde figuren zijn triangulaties, subdivisies, of de hoger-dimensionale equivalenten hiervan.

We zouden zo'n imprecies polygoon op dezelfde manier als punten en lijnen kunnen modelleren, als een deelverzameling van de verzameling van alle mogelijke polygonen. De vraag is echter hoe zo'n deelverzameling op een compacte manier beschreven kan worden. Als alternatief kunnen we een imprecies polygoon beschrijven door de hoekpunten van het polygoon als imprecieze punten te modelleren. In Hoofdstuk 11 geven we twee algoritmes om de bovengrens en ondergrens van de lengte van zo'n polygoon te berekenen. Beide algoritmes werken in $O(n)$ tijd.

Dit model leidt echter wel tot een probleem: als we van het werkelijke polygoon extra eigenschappen weten, bijvoorbeeld dat het convex is of geen zelfdoorsnijdingen heeft, dan is het niet duidelijk dat die eigenschappen ook worden gegarandeerd in alle mogelijke instantiaties van het model. Bekijk als voorbeeld Figuur 23. Twee mogelijkheden voor het werkelijke polygoon zijn te zien, waarvan er één zelfdoorsnijdingen heeft.

Veel polygonen in praktische toepassingen hebben geen zelfdoorsnijdingen, bijvoorbeeld als zo'n polygoon de grens van een gebied weergeeft. Een belangrijke algoritmische vraag is nu, wanneer een imprecies polygoon in dit model is gegeven, hoe de hoekpunten geplaatst kunnen worden zodat het resulterende precieze polygoon inderdaad geen zelfdoorsnijdingen heeft. In Hoofdstuk 12 laten we zien dat dit een NP-moeilijk probleem is. Uit een kleine aanpassing van de constructie volgt bovendien dat een aantal andere problemen, zoals het berekenen van het kortste pad zonder zelfdoorsnijdingen dat een gegeven reeks gebieden in de juiste volgorde doorloopt, ook NP-moeilijk zijn.



Figuur 23 (a) Een imprecies polygoon, waarvan de hoekpunten zijn gemodelleerd als imprecieze punten. (b) Een mogelijke vorm van de "werkelijke" polygoon. (c) Een andere mogelijkheid. Deze polygoon heeft echter zelfdoorsnijdingen.

Conclusie

Data-imprecisie in de computationele meetkunde is een belangrijk probleem dat lang genegeerd is, maar langzaamaan steeds meer aandacht krijgt. Het is van groot belang om een beter beeld te krijgen van hoe imprecisie de uitkomsten van bestaande algoritmes kan beïnvloeden, voordat ze op grote schaal in de praktijk gebruikt zullen worden. Tegelijkertijd zorgt de enorme afname in kosten voor het verzamelen en opslaan van grote hoeveelheden data voor een steeds grotere vraag naar betrouwbare en efficiënte algoritmes.

Dit proefschrift levert een bijdrage aan deze analyse. Het probleem van imprecisie is daarmee niet opgelost, en zal ook nooit helemaal opgelost kunnen worden. Ondanks wat theoretische informatici graag zien is het niet mogelijk om data-imprecisie volledig in modellen te vangen waarbinnen de oplossingen met wiskundige zekerheid kunnen worden gegeven. Desondanks is het zeker mogelijk om meer te doen dan simpelweg het probleem negeren, en dit proefschrift toont aan dat in veel gevallen nuttige informatie over de uitvoer te berekenen is.

Bibliography

[1] M. Abellanas, F. Hurtado, and P. A. Ramos. Structural tolerance and Delaunay triangulation. *Information Processing Letters*, 71:221–227, 1999.

[2] P. K. Agarwal and S. Har-Peled. Approximating extent measures of points. In *Proc. 12th Symposium on Discrete Algorithms*, pages 148–157, 2001.

[3] A. Aggarwal, L. J. Guibas, J. Saxe, and P. W. Shor. A linear-time algorithm for computing the Voronoi diagram of a convex polygon. *Discrete and Compututational Geometry*, 4(6):591–604, 1989.

[4] H.-K. Ahn, S. W. Bae, S.-S. Kim, M. Korman, I. Reinbacher, and W. Son. Square and rectangle covering with outliers. In *Proc. 25th European Workshop on Computational Geometry*, pages 273–276, 2009.

[5] M. al Khwārizmī. On calculation with hindu numerals, $\sim$825.

[6] E. M. Arkin, A. Efrat, C. Erten, F. Hurtado, J. Mitchell, V. Polishchuk, and C. Wenk. Shortest tour of a sequence of disjoint segments in L_1. In *Proc. 16th Fall Workshop on Computational Geometry*, 2006.

[7] B. Aronov and S. Har-Peled. On approximating the depth and related problems. *SIAM Journal on Computing*, 38(3):899–921, 2008.

[8] R. Atanassov, P. Bose, M. Couture, A. Maheshwari, P. Morin, M. Paquette, M. Smid, and S. Wuhrer. Algorithms for optimal outlier removal. *Journal of Discrete Algorithms*, 7(2):239–248, 2009.

[9] C. Bajaj. The algebraic degree of geometric optimization problems. *Discrete and Computational Geometry*, 3:177–191, 1988.

[10] D. Bandyopadhyay and J. Snoeyink. Almost-Delaunay simplices: Nearest neighbour relations for imprecise points. In *Proc. 15th Symposium on Discrete Algorithms*, pages 410–419, 2004.

[11] D. Bandyopadhyay and J. Snoeyink. Almost-Delaunay simplices: Nearest neighbor relations for imprecise 3D points using CGAL. *Computational Geometry: Theory and Applications*, 38(1-2):4–15, 2007.

[12] E. T. Bell. *Men of Mathematics*. Dover publications, New York, 1937.

[13] G. Belovsky. Optimal foraging and community structure: implications for a guild of generalist grassland herbivores. *Oecologia*, 70:35–52, 1986.

[14] M. Bern, D. Eppstein, and J. Gilbert. Provably good mesh generation. *Journal of Computer and System Sciences*, 48:384–409, 1994.

[15] M. Bern, D. Eppstein, and S.-H. Teng. Parallel construction of quadtrees and quality triangulations. *International Journal of Computational Geometry and Application*, 9(6):517–532, 1999.

[16] P. Bhattacharya and A. Rosenfeld. "Convexity" of sets of lines. *Pattern Recognition Letters*, 19:1199–1205, 1998.

[17] J.-D. Boissonnat and S. Lazard. Convex hulls of bounded curvature. In *Proc. 8th Canadian Conference on Computational Geometry*, pages 14–19, 1996.

[18] R. Bruce, M. Hoffmann, D. Krizanc, and R. Raman. Efficient update strategies for geometric computing with uncertainty. *Theory of Computing Systems*, 38(4):411–423, 2005.

[19] K. Buchin, M. Löffler, P. Morin, and W. Mulzer. Delaunay triangulation of imprecise points simplified and extended. In *Proc. 11th Algorithms and Data Structures Symposium*, LNCS 5664, pages 131–143, 2009.

[20] S. Cabello. Approximation algorithms for spreading points. *Journal of Algorithms*, 62(2):49–73, 2007.

[21] L. Cai and J. M. Keil. Computing visibility information in an inaccurate simple polygon. *International Journal of Computational Geometry and Applications*, 7:515–538, 1997.

[22] CGAL, Computational Geometry Algorithms Library. `http://www.cgal.org`.

[23] T. M. Chan. Approximating the diameter, width, smallest enclosing cylinder, and minimum-width annulus. *International Journal of Computational Geometry and Applications*, 12((1-2)):67–85, 2002.

[24] T. M. Chan. Three problems about simple polygons. *Computational Geometry: Theory and Applications*, 35(3):209–217, 2006.

[25] B. Chazelle. Triangulating a simple polygon in linear time. *Discrete and Computational Geometry*, 6(5):485–524, 1991.

[26] B. Chazelle, O. Devillers, F. Hurtado, M. Mora, V. Sacristán, and M. Teillaud. Splitting a Delaunay triangulation in linear time. *Algorithmica*, 34:39–46, 2002.

[27] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. *Journal of the ACM*, 39(1):1–54, 1992.

[28] B. Chazelle and J. Matoušek. On linear-time deterministic algorithms for optimization problems in fixed dimension. *Journal of Algorithms*, 21:579–597, 1996.

[29] B. Chazelle and W. Mulzer. Computing hereditary convex structures. In *Proc. 25th Symposium on Computational Geometry*, pages 61–70, 2009.

[30] F. Y. L. Chin and C. A. Wang. Finding the constrained Delaunay triangulation and constrained Voronoi diagram of a simple polygon in linear time. *SIAM Journal on Computing*, 28(2):471–486, 1998.

[31] K. L. Clarkson and C. Seshadhri. Self-improving algorithms for Delaunay triangulations. In *Proc. 24th Symposium on Computational Geometry*, pages 148–155, 2008.

[32] P. Colley, H. Meijer, and D. Rappaport. Optimal nearly-similar polygon stabbers of convex polygons. In *Proc. 6th Canadian Conference on Computational Geometry*, pages 269–274, 1994.

[33] M. de Berg, H. David, M. J. Katz, M. H. Overmars, A. F. van der Stappen, and J. Vleugels. Guarding scenes against invasive hypercubes. *Computational Geometry: Theory and Applications*, 26(2):99–117, 2003.

[34] M. de Berg, A. F. van der Stappen, J. Vleugels, and M. J. Katz. Realistic input models for geometric algorithms. *Algorithmica*, 34(1):81–97, 2002.

[35] B. Delaunay. Sur la sphère vide. A la memoire de Georges Voronoi. *Izv. Akad. Nauk SSSR, Otdelenie Matematicheskih i Estestvennyh Nauk*, 7:793–800, 1934.

[36] H. Djidjev and A. Lingas. On computing Voronoi diagrams for sorted point sets. *International Journal of Computational Geometry and Applications*, 5:327–337, 1995.

[37] M. Dror, A. Efrat, A. Lubiw, and J. S. B. Mitchell. Touring a sequence of polygons. In *Proc. 35th Symposium on Theory of Computing*, pages 473–482, 2003.

[38] H. Edelsbrunner. Finding transversals for sets of simple geometric figures. *Theoretical Computer Science*, 35:55–69, 1985.

[39] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*, volume 10 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Heidelberg, West Germany, 1987.

[40] H. Edelsbrunner, A. D. Robison, and X. Shen. Covering convex sets with non-overlapping polygons. *Discrete Mathematics*, 81:153–164, 1990.

[41] A. Efrat, G. Rote, and M. Sharir. On the union of fat wedges and separating a collection of segments by a line. *Computational Geometry: Theory and Applications*, 3:277–288, 1993.

[42] A. Einstein. Zur Elektrodynamik bewegter Körper (Electrodynamics of moving bodies). *Annalen der Physik*, 17:891–921, 1905.

[43] J. S. Ely and A. P. Leclerc. Correct Delaunay triangulation in the presence of inexact inputs and arithmetic. *Reliable Computing*, 6:23–38, 2000.

[44] Euclid. Στοιχεῖα (Elements), $\sim$300 BC.

[45] J. Fiala, J. Kratochvil, and A. Proskurowski. Systems of distant representatives. *Discrete Applied Mathematics*, 145:306–316, 2005.

[46] R. A. Finkel and J. L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, 1974.

[47] P. G. Franciosa, C. Gaibisso, G. Gambosi, and M. Talamo. A convex hull algorithm for points with approximately known positions. *International Journal of Computational Geometry and Applications*, 4(2):153–163, 1994.

[48] M. L. Fredman. How good is the information theory bound in sorting? *Theoretical Computer Science*, 1:355–361, 1976.

[49] K. R. Gabriel and R. R. Sokal. A new statistical approach to geographic variation analysis. *Systematic Zoology*, 18:259–278, 1969.

[50] A. Gajentaan and M. H. Overmars. On a class of $O(n^2)$ problems in computational geometry. *Computational Geometry: Theory and Applications*, 5:165–185, 1995.

[51] J. Gates. Some dual problems of geometric probability in the plane. *Combinatorics, Probability and Computing*, 2:11–23, 1993.

[52] K. F. Gauss. Theorie der Bewegung der Himmelskörper, die die Sonne in Kegelschnitten umkreisen (Theory of the motion of the heavenly bodies revolving around the sun in conic sections), 1809.

[53] T. Gonzalez. Algorithms on sets and related problems. Technical Report 75–15, Department of Computer Science, University of Oklahoma, Norman, OK, 1975.

[54] J. E. Goodman. When is a set of lines in space convex? *Notices of the American Mathematical Society*, 45:222–232, 1998.

[55] M. T. Goodrich and J. Snoeyink. Stabbing parallel segments with a convex polygon. *Computer Vision, Graphics, and Image Processing*, 49:152–170, 1990.

[56] R. Graham, B. Lubachevsky, K. Nurmela, and P. Östergård. Dense packings of congruent circles in a circle. *Discrete Mathematics*, 181:139–154, 1998.

[57] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1:132–133, 1972.

[58] C. Gray and W. Evans. Optimistic shortest paths on uncertain terrains. In *Proc. 16th Canadian Conference on Computational Geometry*, pages 68–71, 2004.

[59] C. Gray, M. Löffler, and R. I. Silveira. Minimizing slope change in imprecise 1.5d terrains. In *Proc. 21th Canadian Conference on Computational Geometry*, pages 55–58, 2009.

[60] C. Gray, M. Löffler, and R. I. Silveira. Smoothing imprecise 1.5d terrains. In *Proc. 6th Workshop on Approximation and Online Algorithms*, pages 214–226, 2009.

[61] L. J. Guibas, D. Salesin, and J. Stolfi. Epsilon geometry: building robust algorithms from imprecise computations. In *Proc. 5th Symposium on Computational Geometry*, pages 208–217, 1989.

[62] S. Har-Peled and K. R. Varadarajan. Approximate shape fitting via linearization. In *Proc. 42nd Symposium on Foundations of Computer Science*, page 66, Washington, DC, USA, 2001. IEEE Computer Society.

[63] S. Har-Peled and Y. Wang. Shape fitting with outliers. In *Proc. 19th Symposium on Computational Geometry*, pages 29–38, New York, NY, USA, 2003. ACM.

[64] F. Hassanzadeh and D. Rappaport. Approximation algorithms for finding a minimum perimeter polygon intersecting a set of line segments. In *Proc. 11th Algorithms and Data Structures Symposium*, LNCS 5664, pages 363–374, 2009.

[65] M. Held and J. S. B. Mitchell. Triangulating input-constrained planar point sets. *Information Processing Letters*, 109(1):54–56, 2008.

[66] A. Hernández Barrera. Finding an $o(n^2 \log n)$ algorithm is sometimes hard. In *Proc. 8th Canadian Conference on Computational Geometry*, pages 289–294, 1996.

[67] J. Hershberger. Finding the upper envelope of n line segments in $O(n \log n)$ time. *Information Processing Letters*, 33:169–174, 1989.

[68] J. E. Hopcroft and R. M. Karp. An $n^{\frac{5}{2}}$ algorithm for maximum matching in bipartite graphs. *SIAM Journal on Computing*, 4:225–231, 1973.

[69] D. J. Huggard. A linear programming model of herbivore foraging: imprecise, yet successful? *Oecologia*, 100(4):470–474, 1994.

[70] S. Jadhav, A. Mukhopadhyay, and B. K. Bhattacharya. An optimal algorithm for the intersection radius of a set of convex polygons. *Journal of Algorithms*, 20:244–267, 1996.

[71] W. Ju and J. Luo. New algorithms for computing maximum perimeter and maximum area of the convex hull of imprecise inputs based on the parallel line segment model. In *Proc. 21st Canadian Conference on Computational Geometry*, pages 1–4, 2009.

[72] I. Kant. Kritik der reinen Vernunft (Critique of pure reason), 1781.

[73] A. A. Khanban. *Basic Algorithms of Computational Geometry with Imprecise Input*. PhD thesis, Imperial College, London, 2005.

[74] A. A. Khanban and A. Edalat. Computing Delaunay triangulation with imprecise input data. In *Proc. 15th Canadian Conference on Computational Geometry*, pages 94–97, 2003.

[75] Y. Kholondyrev and W. Evans. Optimistic and pessimistic shortest paths on uncertain terrains. In *Proc. 19th Canadian Conference on Computational Geometry*, pages 197–200, 2007.

[76] D. G. Kirkpatrick. Efficient computation of continuous skeletons. In *Proc. 20th Symposium on Foundations of Computer Science*, pages 18–27, 1979.

[77] C. Knauer, M. Löffler, M. Scherfenberg, and T. Wolle. The directed Hausdorff distance between imprecise point sets. In *Proc. 20th International Symposium on Algorithms and Computation*, 2009 (to appear).

[78] H. Kruger. Basic measures for imprecise point sets in $\mathbb{R}^d$. Master's thesis, Utrecht University, 2008.

[79] H. Kruger and M. Löffler. Geometric measures on imprecise points in higher dimensions. In *Proc. 25th European Workshop on Computational Geometry*, pages 121–124, 2009.

[80] D. T. Lee and F. P. Preparata. Euclidean shortest paths in the presence of rectilinear barriers. *Networks*, 14:393–410, 1984.

[81] D. T. Lee and Y. F. Wu. Geometric complexity of some location problems. *Algorithmica*, 1:193–211, 1986.

[82] D. Lichtenstein. Planar formulae and their uses. *SIAM Journal on Computing*, 11(2):329–343, 1982.

[83] M. Löffler. Smallest and largest convex hulls for imprecise points. Master's thesis, Utrecht University, 2005.

[84] M. Löffler. Existence and computation of tours through imprecise points. *International Journal of Computational Geometry and Applications*, 2008. in press.

[85] M. Löffler and J. Phillips. Shape fitting on point sets with probability distributions. In *Proc. 17th European Symposium on Algorithms*, LNCS 5757, pages 313–324, 2009.

[86] M. Löffler and J. Snoeyink. Delaunay triangulations of imprecise points in linear time after preprocessing. *Computational Geometry: Theory and Applications*, 2009. doi:10.1016/j.comgeo.2008.12.007, in press.

[87] M. Löffler and M. van Kreveld. Approximating largest convex hulls for imprecise points. *Journal of Discrete Algorithms*, 6(4):583–594, 2008.

[88] M. Löffler and M. van Kreveld. Geometry with imprecise lines. In *Proc. 24th European Workshop on Computational Geometry*, pages 133–136, 2008.

[89] M. Löffler and M. van Kreveld. Largest and smallest convex hulls for imprecise points. *Algorithmica*, 2008. doi:10.1007/s00453-008-9174-2, in press.

[90] M. Löffler and M. van Kreveld. Largest bounding box, smallest diameter, and related problems on imprecise points. *Computational Geometry: Theory and Applications*, 2009. doi:10.1016/j.comgeo.2009.03.007, in press.

[91] J. Matoušek, J. Pach, M. Sharir, S. Sifrony, and E. Welzl. Fat triangles determine linearly many holes. *SIAM Journal on Computing*, 23:154–169, 1994.

[92] J. Matoušek, M. Sharir, and E. Welzl. A subexponential bound for linear programming. *Algorithmica*, 16:498–516, 1996.

[93] N. Megiddo. Linear-time algorithms for linear programming in $\mathbb{R}^3$ and related problems. *SIAM Journal on Computing*, 12(4):759–776, 1983.

[94] N. Megiddo. Linear programming in linear time when the dimension is fixed. *Journal of the ACM*, 31:114–127, 1984.

[95] N. Megiddo. On the ball spanned by balls. *Discrete and Computational Geometry*, 4:605–610, 1989.

[96] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, UK, 2000.

[97] N. Miller and M. Sharir. Efficient randomized algorithms for constructing the union of fat triangles and pseudodiscs. Unpublished manuscript.

[98] A. Mukhopadhyay, E. Greene, and S. V. Rao. On intersecting a set of isothetic line segments with a convex polygon of minimum area. In *Proc. 2007 International Conference on Computational Science and Its Applications*, LNCS 4705, pages 41–54, 2007.

[99] A. Mukhopadhyay, C. Kumar, E. Greene, and B. Bhattacharya. On intersecting a set of parallel line segments with a convex polygon of minimum area. *Information Processing Letters*, 105(2):58–64, 2008.

[100] O. Musin. Properties of the Delaunay triangulation. In *Proc. 13th Symposium on Computational Geometry*, pages 424–426, 1997.

[101] Y. Myers and L. Joskowicz. The linear parametric geometric uncertainty model: Points, lines and their relative positioning. In *Proc. 24th European Workshop on Computational Geometry*, pages 137–140, 2008.

[102] Y. Myers and L. Joskowicz. Point distance problems with dependent uncertainties. In *Proc. 25th European Workshop on Computational Geometry*, pages 73–76, 2009.

[103] T. Nagai and N. Tokura. Tight error bounds of geometric problems on convex objects with imprecise coordinates. In *Proc. Japanese Conference on Discrete and Computational Geometry*, LNCS 2098, pages 252–263, 2000.

[104] T. Nagai, S. Yasutome, and N. Tokura. Convex hull problem with imprecise input and its solution. *Systems and Computers in Japan*, 30(3):31–42, 1999.

[105] S. Ntafos. Watchman routes under limited visibility. *Computational Geometry: Theory and Applications*, 1(3):149–170, 1992.

[106] J. O'Rourke. Open problem 41. `http://maven.smith.edu/~orourke/TOPP/P41.html`.

[107] Y. Ostrovsky-Berman and L. Joskowicz. Uncertainty envelopes. In *Proc. 21st European Workshop on Computational Geometry*, pages 175–178, 2005.

[108] M. Pocchiola and G. Vegter. On polygonal covers. In B. Chazelle, J. Goodman, and R. Pollack, editors, *Advances in Discrete and Computational Geometry*, volume 223 of *Contemporary Mathematics*, pages 257–268. AMS, Providence, 1999.

[109] V. Polishchuk and J. S. B. Mitchell. Touring convex bodies - a conic programming solution. In *Proc. 17th Canadian Conference on Computational Geometry*, pages 290–293, 2005.

[110] C. Ptolemaeus. Geographia, $\sim$150.

[111] D. Rappaport. A convex hull algorithm for discs, and applications. *Computational Geometry: Theory and Applications*, 1(3):171–181, 1992.

[112] D. Rappaport. Minimum polygon transversals of line segments. *International Journal of Computational Geometry and Applications*, 5:243–256, 1995.

[113] J. Robert and G. Toussaint. Computational geometry and facility location. Technical Report SOCS 90.20, McGill University, Montreal, 1990.

[114] A. Rosenfeld. "Geometric properties" of sets of lines. *Pattern Recognition Letters*, 16:549–556, 1995.

[115] A. Rosenfeld. Fuzzy geometry: An updated overview. *Information Sciences*, 110(3-4):127–133, 1998.

[116] G. Rote, C. A. Wang, L. Wang, and Y. Xu. On constrained minimum pseudotriangulations. In *Proc. 9th International Computing and Combinatorics Conference*, LNCS 2697, pages 445–454, 2003.

[117] A. Schönhage. On the power of random access machines. In *Proc. 6th Colloquium on Automata, Languages and Programming*, LNCS 71, pages 520–529. Springer-Verlag, 1979.

[118] R. Seidel. A method for proving lower bounds for certain geometric problems. In G. T. Toussaint, editor, *Computational Geometry*, pages 319–334. North-Holland, Amsterdam, Netherlands, 1985.

[119] J. Sember and W. Evans. Guaranteed voronoi diagrams of uncertain sites. In *Proc. 20th Canadian Conference on Computational Geometry*, 2008.

[120] M. I. Shamos. *Computational Geometry*. Ph.D. thesis, Department of Computer Science, Yale University, New Haven, CT, 1978.

[121] R. I. Silveira and R. van Oostrum. Flooding countries and destroying dams. In *Proc. 10th Workshop on Algorithms and Data Structures*, LNCS 4619, pages 227–238, 2007.

[122] E. Specht. The best known packings of equal circles in the unit circle (up to $n = 500$). `http://hydra.nat.uni-magdeburg.de/packing/cci/cci.html`, July 2007.

[123] J. Steiner. Systematische Entwickelung der Abhängigkeit geometrischer Gestalten von einander (Systematic development of the dependencies of geometric objects on each other), 1832.

[124] A. J. Stewart. Robust point location in approximate polygons. In *Proc. 3rd Canadian Conference on Computational Geometry*, pages 179–182, Aug. 1991.

[125] X. Tan and T. Hirata. Finding shortest safari routes in simple polygons. *Information Processing Letters*, 87:179–186, 2003.

[126] T. Tasdizen and R. T. Whitaker. Feature preserving variational smoothing of terrain data. In *Proc. 2nd Workshop on Variational Geometric and Level Set Methods in Computer Vision*, 2003.

[127] M. van Kreveld, M. Löffler, and J. Mitchell. Preprocessing imprecise points and splitting triangulations. In *Proc. 19th International Symposium on Algorithms and Computation*, LNCS 5369, pages 544–555, 2008.

[128] F. Weller. Stability of Voronoi neighborship under perturbations of the sites. In *Proc. 9th Canadian Conference on Computational Geometry*, pages 251–256, 1997.

[129] C.-K. Yap. Robust geometric computation. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 41, pages 927–952. Chapman & Hall/CRC, 2004.

[130] L. A. Zadeh. Fuzzy sets. *Information and Control*, 8(3):338–353, 1965.

[131] J. Zhang and M. Goodchild. *Uncertainty in Geographical Information*. MPG Books Ltd, Bodmin, 2002.

Curriculum Vitae

Maarten Löffler was born on August 14, 1982, in Amsterdam. In 1994, he started his pre-university education at the Marnix College in Ede, from which he graduated at Atheneum level in 2000, on the subjects Dutch, English, Mathematics A and B, Physics, Chemistry, Biology and Music. In 2000, he started as a student in Mathematics and Computer Science at Utrecht University. He received his Master Degree in Computer Science *cum laude* in 2005, on his thesis called "Smallest and Largest Convex Hulls for Imprecise Points", which he wrote under supervision of Dr. Marc van Kreveld. In 2006, he also received his Master Degree in Mathematics *cum laude*. In 2005, he started as AIO (Assistent In Opleiding) at the Department of Information and Computing Sciences of Utrecht University, and he wrote his PhD thesis in 2009.

Acknowledgements

Writing a PhD thesis is not something you do on your own, even when you have four years to do so. I will now try to mention everybody who has, directly or indirectly, contributed to the construction of this thesis. I will probably fail, but nonetheless it will be interesting to see how many names I can manage to squeeze into these two pages.

In the first place, I would like to thank Marc van Kreveld, who has been my supervisor these four years and who gave me the opportunity to start on my PhD. Parts II and IV of this thesis are based on work by me and Marc. For Part III, though, I collaborated with several other people as well. In 2007, I visited Jack Snoeyink for a month, which led to the results in Chapter 8. Afterwards, I worked from a distance with Joe Mitchell, and this resulted in Chapter 7 of this thesis. Finally, Chapter 9 is based on joint work with Kevin Buchin, Pat Morin and Wolfgang Mulzer.

In Chapter 1 of the introductory part of this thesis, I gave a very short overview of the history of geometry and computational geometry, and the role that imprecision has played in it. It is impossible to do justice to the great scientists of the past who are mentioned in that chapter in such a short space. I based myself largely on the great book by Eric Bell, *Men of Mathematics* [12], and advise the interested reader to take a look at it. I would also like to thank all the people who contributed to Wikipedia for making a lot of information easy to find on the internet, and in particular for making Figure 1.2 available.

In Chapter 2, I sketched an overview of the relevant work that has been done on data imprecision in computational geometry. Apart from the work that appears in detail in this thesis, I have also been able to work with Chris Gray, Christian Knauer, Hein Kruger, Jeff Philips, Rodrigo Silveira, Marc Scherfenberg and Thomas Wolle on problems in this field. I would also specifically like to thank Joachim Gudmundsson and Pat Morin for organising the NICTA Workshop on Computational

Geometry for Imprecise Data in December 2008 at the NICTA University of Sydney Campus. The other participants of this workshop were Hee-Kap Ahn, Sang Won Bae, Dan Chen, Otfried Cheong, Allan Jørgensen, Stefan Langerman, Marc Scherfenberg, Michiel Smid, Tasos Viglas and Thomas Wolle. The workshop led to several interesting results. But perhaps even more importantly, it forced me to think about my own research so far and how this fitted in the general topic of data imprecision at a higher level.

Most of the work in this thesis appeared in the proceedings of various conferences, and some of it has been published in journals. For these purposes, most chapters have been read and checked by many anonymous referees, as well as by Alon Efrat. These reviewers provided valuable feedback that certainly improved the final shape of the material. After converting everything into a thesis, it has been read extensively again by Marc, and also by Renske Löffler and Johan de Ruiter. Their feedback is greatly appreciated. Finally, it has been read by the official reading committee consisting of Karen Aardal, Mark de Berg, Stefan Langerman, Jan van Leeuwen and Jack Snoeyink.

I think I have now mentioned everyone who has had a direct influence on this thesis. However, nearly as important are all the other people who I have closely interacted with, and who made the past four years such a pleasant experience for me.

Specifically, I am glad I sometimes also had the opportunity to think about other problems than those involving imprecision. In my time as a researcher so far, I had the privilege of obtaining some very precise results while working with, apart from some people already mentioned, Boris Aronov, Pankaj Agarwal, Magdalene Borgelt, Jit Bose, Maike Buchin, Sergio Cabello, Luc Devroye, Jaochim Gudmundson, Herman Haverkort, Elad Horev, Bart Jansen, Tom de Jong, Matya Katz, Thierry de Kok, Roi Krakovski, Jun Luo, Damian Merrick, Elena Mumford, Matthew O'Meara, Günter Rote, Bettina Speckmann and Mostafa Vahedi. I very much enjoyed our collaborations, and hope to be able to work with you again in the future.

For the past four years, I have shared an office with Rodrigo Silveira (apart from a short intermezzo, when we moved to another building and Jur van den Berg and Dennis Nieuwenhuisen joined us for a few months). I will remember our countless discussions about how to decorate our office, the temperature (and time) differences between our desks, the art of digging, and most importantly, what is the optimal time for having tea. Our office has often been used as a meeting place when discussing problems with the other geometers who have populated our group, such as René van Oostrum, Jun Luo, Magdalene Borgelt, and Kevin and Maike Buchin. And of course, there were all the others who are or have been part of our group, as well as those in the *other* group, to share lunch, play table football, or attend conferences in Center Parcs with.

Finally, I would like to thank all my friends and my family, especially papa, mama en zus, for being there and making my life beside work such a happy one during the past four years, and all the time before.

Advanced School for Computing and Imaging

This work was carried out in the ASCI graduate school.
ASCI dissertation series number 182.

NWO
Nederlandse Organisatie voor Wetenschappelijk Onderzoek

This work was supported by NWO through the Open Competition project GOGO.