Geometric Spanner Networks

Giri Narasimhan

School of Computer Science
Florida International University
Miami, FL 33199
U.S.A.
E-mail: giri@cis.fiu.edu

Michiel Smid

School of Computer Science
Carleton University
Ottawa, Ontario
Canada K1S 5B6
E-mail: michiel@scs.carleton.ca

June 6, 2006

Dedicated to:

My parents, Kalyani and Narasimhan, who taught me the fundamentals of life, and the pursuit of knowledge and excellence, and Kalai Mathee, with whom I collaborate on the advanced concepts. (G.N.)

My parents, Nell and Giel, who (unknowingly) convinced me to become a mathematician and theoretical computer scientist. (M.S.)

Contents

Preface

> Philosophy is written in this grand book - I mean universe - which stands continually open to our gaze. But the book cannot be understood unless one first learns to comprehend the language and read the letters in which it is composed. It is written in the language of mathematics, and its characters are triangles, circles, and other geometric figures without which it is humanly impossible to understand a single word of it.
>
> — Galileo, *The Assayer (Il saggiatore)*, 1623

This book started as a collection of personal notes on Geometric Spanner Networks. Over time, these notes grew, and we realized that they could be of value to many researchers in the field. Gautam Das suggested that it be turned into a monograph. It made sense, because the geometric spanner problem is closely related to several fundamental problems in geometric and graph algorithms, including the minimum spanning tree problem, the Steiner minimum tree problem, the traveling salesperson problem, the shortest path problem, and more. We assume that the reader has had previous exposure to (undergraduate level) basic concepts of discrete mathematics, data structures, probability and combinatorics, algorithm analysis, fundamental algorithms, and mathematical proof techniques. This book can serve as a reference text, and can also be used as a self-study book for anyone interested in research in computational geometry and geometric algorithms.

One of the main features of this book is its attention to detail—detail in the proofs and arguments presented. We have striven to present complete proofs, wherever possible and appropriate, while at the same time peppering it with intuition, so that the reader can understand the underlying train of

thought.

While there are numerous examples of the design of efficient algorithms much before that, by 1864 Charles Babbage foresaw rather clearly the development of the field of algorithms when he wrote in *The Life of a Philospher*:

> *As soon as an* Analytical Engine *exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will then arise—by what course of calculation can these results be arrived at by the machine in the shortest time?*

Many of us were drawn to the field of algorithms because of the elegance, subtlety, precision, and clarity of the ideas and arguments. This is especially true of some of the early work in the field, some of which is now part of undergraduate texts. The field has evolved since then. The algorithms and data structures and analysis tools have become more complex and sophisticated. If one wants to keep up with the advances in the field, one has to learn how to read papers, focus on central ideas, and to skip unnecessary details that may cloud an elegant argument. But this requires training and skill, and it is particularly challenging for the novice, the beginning graduate student. As academics, we already know that "teaching is not the mere imparting of information but the cultivation of an inquiring mind" (J.Krishnamurthy, *Life Ahead* (1963)). But, we face a greater challenge—one of capturing and maintaining the students' interest and in keeping them challenged. Details, when presented in papers, are often boring and time-consuming, especially when the underlying ideas are new. Details are hard to write down. So, it is with good reason that details are often pushed "under the rug".

Then why should the novice be concerned about understanding details? It takes skill and training to convert good ideas and great intuition into an end result (an algorithm, a proof, etc.). When writing a paper, without the attention to details, one is more prone to errors. It is easier to make wild claims and statements without details, and it is harder to see subtle errors without the attention to details. Many of us have refereed papers for journals and conferences, where this is certainly a critical issue. Also, details help you "take apart" an algorithm or a proof; details help you to understand limitations that are not obvious from an overview; details will help you to understand the hurdles that must be overcome to make improvements; finally details will help you to innovate and dig deeper.

Web site

This book has its own web site(s); the following mirror sites:

`www.scs.carleton.ca/~michiel/SpannerBook.html`

`www.cis.fiu.edu/~giri/SpannerBook.html`

will contain a list of known errors and new developments in Geometric Spanner Networks.

Acknowledgements

We gratefully acknowledge all our "co-conspirators", who have worked with us on spanners and related geometric topics, and from whom we have learnt an awful lot. These collaborators include: Lyudmil Aleksandrov, Estie Arkin, Srinivasa Arikati, Sunil Arya, Prosenjit Bose, Danny Chen, Paul Chew, Gautam Das, Christian Duncan, Joachim Gudmundsson, Dagmar Handke, Paul Heffernan, Ravi Janardan, Sanjiv Kapoor, Rolf Klein, Christian Knauer, Aaron Lee, Hans-Peter Lenhof, Christos Levcopoulos, Anil Maheshwari, Joe Mitchell, Pat Morin, Jason Morrison, David Mount, Kirk Pruhs, Jeff Salowe, Warren Smith, Petra Specht, Jan Tusch, David Wood, Martin Zachariasen, Christos Zaroliagis, Norbert Zeh, and Jianlin Zhu.

We thank David Peleg for his perspective on the history of spanners.

We are indebted to several anonymous referees and to people who sent us feedback on preliminary versions of some of the chapters of this book. They include: Hubert Chan, Rolf Klein, Piyush Kumar, Aaron Lee, Tamas Lukovszki, Joe Mitchell, Marcin Pilat, and Justin Schechter.

Parts of this book were used as lecture notes at the Max-Planck-Institute for Computer Science in Saarbrücken, and for courses at the University of Magdeburg and Carleton University. We thank the students in Magdeburg and Ottawa for their enthusiasm and feedback.

The people at Cambridge University Press have been a pleasure to work with. Pooja Jain has been tremendously helpful. We particularly appreciate the patience and the attention to detail of Lauren Cowles, who has worked with us right from the start.

Initial ideas for the cover design came from Kalai Mathee. The talent of Namitha Raju helped to refine these conceptions and brought them closer to reality. We are grateful to both of them.

A project of this magnitude is never possible without the support and encouragement of friends and family. We are forever indebted to them.

Much of our research on spanners would not have been possible without the resources provided to us by our parent institutions during this period. These include the Max-Planck-Institute, King's College London, University of Magdeburg, Carleton University, University of Memphis, and Florida International University. We also thank DIKU (Copenhagen), University of Magdeburg, Lund University, and Florida International University which hosted several visits, making a lot of this work possible.

We acknowledge the funding provided by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the National Science Foundation (NSF) for some of our personal research reported in this book.

Giri Narasimhan
Miami

Michiel Smid
Ottawa

Part I

Preliminaries

Chapter 1

Introduction

> The truth is, most of us discover where we are headed when we arrive. At that time, we turn around and say, yes, this is obviously where I was going all along.
>
> — Bill Watterson, 1990.

1.1 What is this book about?

In this book, we will consider the following problem:

> **General network design problem:** Given a set S of n points in $\mathbb{R}^d$, how to construct a *good* network that connects these points?

What do we mean by this? A *network connecting the points* of S is a graph $G = (S, E)$ with vertex set S and edge set $E \subseteq S \times S$, such that any two points $p, q \in S$ are connected by a path in G. A *geometric network* is a weighted graph where the vertices correspond to point sites in Euclidean space and the weights on the edges correspond to the distances between the endpoints in the Euclidean metric. Clearly, there are many such networks. For example, the *complete graph*, in which all pairs of distinct points are connected by an edge is one such network connecting the points of S. The

disadvantage, however, is that the number of edges is $\binom{n}{2} = \Theta(n^2)$; it is quadratic in the number of points.

Assume the goal is to construct networks having only "few" edges. How many edges does a connected graph necessarily contain? The answer is $n-1$. In order to prove this, consider an arbitrary network G connecting the points of S, and let m be the number of edges of G. Hence, we have to show that $m \geq n-1$. Let us do the following: As long as G contains a cycle, take an arbitrary cycle, and remove an arbitrary edge from it.

Removing one edge from a cycle does not destroy the connectivity of the graph. Therefore, repeating this operation over and over again, we obtain an acyclic connected graph G', i.e., a *tree*. If m' denotes the number of edges of G', then clearly $m \geq m'$. Hence, it suffices to show that $m' \geq n-1$.

We claim that in fact $m' = n-1$. That is, any tree on n points has exactly $n-1$ edges. The proof is by induction on n. For $n = 1$, the claim is trivial. Let $n \geq 2$, and assume the claim is true for all trees on $n-1$ points. Now let G' be a tree on n points. Since G' is acyclic, there is a point with degree exactly one. (This is easy to prove by contradiction: if all degrees are larger than one, then the graph must contain a cycle.) Remove this point, together with its adjacent edge, from G'. This gives a graph G'' on $n-1$ points; in fact, G'' is a tree. Hence, by the induction hypothesis, G'' has $n-2$ edges. Since G' itself has one more edge, the proof is complete.

Property 1.1.1 *Any network connecting a set of n points must have at least $n-1$ edges.*

Let us call a network *sparse*, if it has $O(n)$ edges, i.e., the number of edges is linear in the number of points. Again, there are many such sparse networks. Later in this chapter, we discuss several of them; each one is *good* in some sense.

The more general network design problem is to connect the set of sites by a network that satisfies a specified set of properties. To measure how *good* a network is, several quality measures have been used in the literature, some of which are listed here:

Network Quality Measures

1. **Size** is defined as the number of edges in the network. In general, it is preferable to have networks with as few edges as possible, perhaps linear in the number of points.

2. **Weight** is defined as the sum of the weights of the edges. Since any network must connect all the points, its weight is bounded from below by the weight of a minimum spanning tree. The weight is a good measure of the cost of building the network; thus, it is often desirable to have networks with small weight.

3. **Stretch Factor** (or **Dilation** or **Distortion**) for two given points is defined as the ratio of the distance between the two points in the network to the metric distance between them. The stretch factor of a network is defined as the maximum stretch factor for any pair of distinct points in the network. It is often required that the stretch factor of the network be bounded by a small constant (which must be at least one). Networks with stretch factor at most t are called t-SPANNERS.

4. **Degree** is the maximum number of edges incident on any point in the network, often required to be bounded by a small constant. For a network, bounded degree implies small size, but not vice versa.

5. **Diameter** is the maximum number of edges along a shortest path connecting any two vertices in the network. It dictates the conciseness with which network paths can be described. A weighted diameter, i.e., the total length of the longest path of minimum length could also be used as a quality measure instead. Diameters may be required to be small.

6. **Connectivity** is the vertex or edge connectivity of the network. It is a measure of how fault-tolerant the network is, since it signifies the number of points or links that must fail in order for the network to be disconnected.

7. **Fault Tolerance** is the number of points or links that must fail in order for the network to fail to have some desirable properties. This is slightly more general than the previous property.

8. **Genus** In many applications, it is desirable for a network to be nearly planar. This is often quantified by its genus; it may also be measured by the size of the largest planar subgraph.

9. **Number of Steiner Points** Often, better networks can be designed by adding Steiner points (points not in the input set). However, one may be constrained to have very few or no Steiner points in the network.

10. **Load Factor** of an edge can be defined as the number of shortest paths between some pair of vertices that use this edge (many alternate definitions and generalizations exist). The load factor of a network is defined as the load factor of the most "loaded" edge in the network. In order to prevent bottlenecks, it is desirable that the load factor of a network be small.

Earlier we discussed the property *size*. In general, when designing a network, one would like to impose constraints on a combination of the quality measures mentioned above; when analyzing a network one would like to understand the properties of the network with respect to these quality measures. A common thread among most of the problems mentioned in this monograph is the study of networks with small stretch factor (in combination with other properties); as mentioned earlier, such networks are called *spanners*. Spanners with small size and/or weight are referred to as *sparse spanners*.

To motivate such problems, we mention a few "concrete" examples. Consider the network of highways connecting n cities, where a road is a straight-line segment connecting two cities. If we want to travel from p to q, then we have to travel at least a distance $|pq|$, the Euclidean distance between p and q. Of course, if there is a direct highway linking p and q, then we travel this distance more or less exactly. Otherwise, it would be nice if there is a path between p and q whose length is not too large as compared to $|pq|$, thus giving rise to the concept of a spanner. Observe that the *length* of a path is defined as the sum of the lengths of its edges.

Consider the section of the Scandinavian rail network (prior to the year 2000) shown in Figure 1.1. A quick inspection reveals that if one wants to use the rail system to travel from Malmö (marked by a M) in Sweden to Copenhagen (marked by a C) in Denmark, then the distance between these cities in the rail network is more than five times the direct "as-the-crow-flies" distance. Adding a direct link between these two cities would clearly improve the rail network[1]

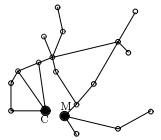


Figure 1.1: *A section of the Scandinavian rail network prior to the year 2000.*

We consider another example of a network **design** problem that relates to stretch factor: Imagine an existing set of highways connecting a collection of cities, where there is a need to upgrade the

[1] A 16-kilometer bridge across the Øresund connecting the two cities was opened to traffic during the summer of 2000.

highway system in a cost-effective manner. Instead of spending the resources to improve all the existing highways, it would be better to improve only a carefully selected subset of the existing highways. If this subset of highway segments constitute a sparse spanner network (small stretch factor, small size and small weight) of the highway system, then it is guaranteed that (i) one could drive from any city to any other using only improved highway segments with only a constant factor increase in the driving distance over distances in the original system of highways, and (ii) the amount of resources needed to upgrade the highway system is small since a sparse spanner has small size (number of highway segments) and weight (total length of the highways).

Given an existing system of highways, it is also easy to understand the significance of network **analysis**. Obvious queries include: "What is the size, weight, stretch factor, diameter, degree or connectivity of the network?", or "What is the farthest pair of cities in the network?", or "What is the stretch factor for a given pair of cities (i.e., what is the ratio of the length of the shortest path between two given cities in the network to the Euclidean distance between them?)". For a federal authority maintaining the highway system, an appropriate query might be: "For which pair of cities in the network is the stretch factor the largest?". If this authority has the resources to build some highway segments, a useful query would be: "Which edge (or k edges) should be added to the network to achieve the greatest decrease in stretch factor and/or load factor?". While budgeting for future improvements to the highway network, a planner may ask: "What is the total length of the edges to be added to the network to achieve a desired fault-tolerance and stretch factor without destroying the planarity?". Planning for emergency situations requires analysis of the fault-tolerance of the network, and this may provoke a query of the type: "What is the maximum increase in stretch factor of the network (assuming it remains connected) if all highway segments within a 50-mile radius of some point are unusable due to a natural disaster?".

The list of desirable properties in a "good" network reflects many contradictory needs. For example, if bounded degree networks are desired, then small diameters are not possible; if small stretch factor networks are needed, then arbitrarily small size or weight may not be possible. Thus many network design problems display interesting tradeoffs between the quality measures and they can be thought of as multi-criteria optimization problems. It is clear that a versatile software package with a suite of network design and analysis tools can be invaluable in making complex, practical decisions regarding geometric networks.

The network design problem encompasses many interesting and fundamental problems. The *minimum spanning tree* can be thought of as a network with least possible weight and infinite stretch factor; if *Steiner* points can be added, then the network with least possible weight and infinite stretch factor is the *Steiner minimum tree*. As the required stretch factor is decreased and approaches one, the network becomes denser until it ultimately becomes the trivial complete graph. If the degree of each site is required to be two, then the network with least weight is the *traveling salesperson tour* on the points.

1.1.1 Spanning trees

A tree on a set S of n points is an acyclic connected graph on these points. We also call such a graph a *spanning tree* of S. A spanning tree is good in the sense that it has the minimum number of edges.

A set of points has many spanning trees. Sylvester showed in 1857—and, independently, Cayley in 1889—that any set of n points has exactly n^{n-2} spanning trees.

Let T be a spanning tree of the set S. The *weight* $wt(T)$ of T is defined to be the sum of the lengths of its edges, where the *length* of an edge $\{p, q\}$ is the Euclidean distance $|pq|$ between p and q. A *minimum spanning tree* $MST(S)$ of S is a spanning tree of minimum weight. A minimum spanning tree on 13,509 US cities is shown in Figure 1.2.

Property 1.1.2 *A minimum spanning tree of a set S is a* shortest *network connecting the points of S.*

In particular, Property 1.1.2 states the obvious fact that a shortest connected network must be a tree. Thus, a minimum spanning tree is good in the sense that both its number of edges and its weight are minimum. The following property states that it is also good in the sense that it has a small degree. The proof of this property is left as an exercise (see Exercise 4.3).

Property 1.1.3 *In a minimum spanning tree of a set of points in the plane, each point has degree at most six.*

In fact, if S is a finite set of points in $\mathbb{R}^d$, where $d \geq 2$, then the degree of each point of the minimum spanning tree of S is bounded from above by a constant that only depends on d.

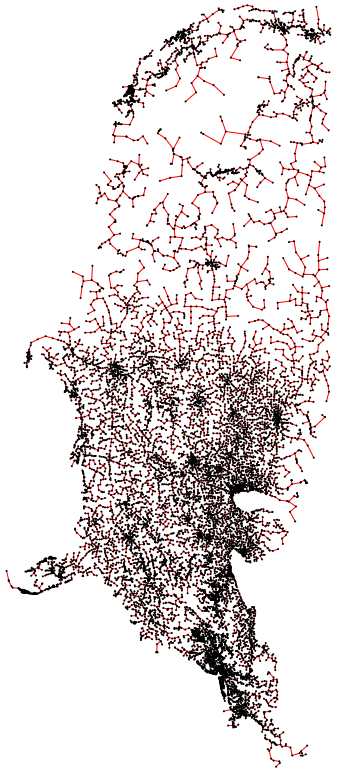


Figure 1.2: *A minimum spanning tree on 13,509 US cities.*

The first algorithm for computing a minimum spanning tree (of an arbitrary weighted graph) is due to Borůvka and dates back to 1926. In Section 2.6, we will present two other algorithms for computing a minimum spanning tree.

1.1.2 Steiner trees

According to Property 1.1.2, a minimum spanning tree is a shortest network that connects a point set S. This is not quite true; it is a shortest graph *with vertex set* S that connects these points.

Assume we consider connected graphs $G = (V, E)$, whose vertex set V contains the set S. In other words, we allow the graph to contain *additional* vertices. The vertices of $V \setminus S$ are called *Steiner points*. Let $SMT(S)$ be such a graph G of minimum weight, where the weight $wt(G)$ of G is defined to be the sum of the lengths of its edges. This graph is called a *Steiner minimum tree* of S, named after Jakob Steiner—a Swiss mathematician who lived from 1796 until 1863. (Apparently, Steiner had nothing to do with Steiner minimum trees.)

The problem of finding a Steiner minimum tree of the vertices of a triangle is called the *Fermat problem*, named after Pierre de Fermat (1601–1665). Gauß (1777–1855) computed the Steiner minimum tree of a set of four German cities. His motivation was to link these cities by railroad.

It is clear that $wt(SMT(S)) \leq wt(MST(S))$. Can a Steiner minimum tree be much smaller than a minimum spanning tree? The following lemma shows that the answer is "no".

Lemma 1.1.4 *Let S be a finite set of points in $\mathbb{R}^d$. Then*

$$wt(MST(S)) \leq 2 \cdot wt(SMT(S)).$$

Proof. Let T be a Steiner minimum tree for S. We will construct a spanning tree T' of S having weight at most twice the weight of T. This will prove the lemma, because the minimum spanning tree of S has weight at most that of T'.

Here is the construction. We double each edge of T. This gives a multigraph W connecting the points of S and the Steiner points of T. The degree of each vertex in W is even. Hence, W contains an *Euler tour*[2] W', which

[2]named after Euler (1707–1783), the father of graph theory

is a tour that visits each edge of W exactly once and that returns to the starting vertex. Observe that this tour may visit vertices more than once. It is clear that the weight of W' is equal to that of W, which in turn is equal to twice the weight of T.

We will construct from W' a spanning tree T' of the points of S. The basic operation is that of *short-cutting*: Assume the Euler tour moves from point a to point b, and then to point c. Then short-cutting means that from point a, we immediately move to point c. By the triangle inequality, such a short-cut operation gives a tour—in general not along all points—having weight at most the weight of W'.

By following the tour W', we repeatedly make short-cuts, in such a way that we (i) remove all Steiner points, and (ii) for each point p of S, remove all visits to p, except the first one. The result is a path that visits each point of S exactly once, and whose weight is at most twice the weight of T. This path is clearly a spanning tree of S; it is the tree T' we were looking for. ■

Thus, the weight of a minimum spanning tree is at most twice that of a Steiner minimum tree. Is the factor 2 best possible? We will see in Exercise 1.5 that, for point sets in the two-dimensional plane, a factor smaller than $2/\sqrt{3}$ is not possible. In 1968, Gilbert and Pollack conjectured that $2/\sqrt{3}$ is in fact the best possible factor. That is, they conjectured that for any finite set S of points in $\mathbb{R}^2$,

$$wt(MST(S)) \leq \frac{2}{\sqrt{3}} wt(SMT(S)).$$

This remained an open problem for over twenty years until Du and Hwang settled the conjecture in 1990.

We mention that Steiner minimum trees are extremely hard to compute; the problem is known to be **NP**-hard. In fact, it is not known if the decision problem for the Euclidean metric is in **NP**. On the other hand, a minimum spanning tree of n points in the plane can be computed in $O(n \log n)$ time.

1.1.3 The traveling salesperson tour

The *traveling salesperson problem* is an important problem that influenced the blossoming of fields such as operations research, polyhedral combinatorics, probabilistic analysis and complexity theory. The *traveling salesperson tour* $TSP(S)$ of a finite set S of points is the shortest tour that visits

each point of S exactly once, and returns to the starting point. Here the assumption is that the set S of points belongs to a metric space. The problem of computing an optimal length tour is **NP**-hard, even when the points lie in Euclidean space. In terms of approximation algorithms, Rosenkrantz, Stearns, and Lewis showed that a factor-2 approximation to the optimal tour for points in an arbitrary metric space can be obtained from the minimum spanning tree; their argument is basically the one that we presented in the proof of Lemma 1.1.4. By combining minimum spanning trees with minimum weight matchings, Christofides improved the approximation factor to 3/2. In 1996, Arora (and, independently, Mitchell) improved on these results, by designing a polynomial-time approximation scheme for the Euclidean case. An improved polynomial-time approximation scheme by Rao and Smith in 1998 made use of the concept of spanners; details will be given in Chapter 19.

1.1.4 Triangulations

Let S be a set of n points in the plane. A *triangulation* of S is a partition of the convex hull of S into triangles, such that the vertices of these triangles are exactly the points of S. Since a triangulation is a planar graph, it follows from Euler's theorem that it has at most $3n - 6 = O(n)$ edges. Observe also that a triangulation is a connected graph. Therefore, it is a sparse network connecting the points of S.

In general, a point set can have many triangulations. Santos and Seidel proved in 2003 that any set of n points in the plane has $O(59^n)$ many triangulations. Some triangulations have special properties:

- The Delaunay triangulation, which is the dual of the Voronoi diagram.
- The minimum weight triangulation, which is a triangulation, whose weight is minimum among all triangulations of the point set. It is not known if the problem of computing this triangulation is in **P** or if it is **NP**-hard.
- The greedy triangulation, which is defined as follows: Sort all $\binom{n}{2}$ edges of the complete graph in increasing order of their lengths. Start with a graph G whose edge set is empty, and consider the edges in sorted order, one after another. Add the current edge to G if and only if it does not intersect any edge already contained in G.

1.2 The topic of this book: spanners

In this book, we will mainly be concerned with the problem of designing algorithms that compute geometric networks whose stretch factor is bounded. As we have mentioned already, such networks will be referred to as spanners.

Definition 1.2.1 (Spanner) Let S be a set of n points in $\mathbb{R}^d$ and let $t \geq 1$ be a real number. A *t-spanner* for S is an undirected graph G with vertex set S, such that for any two points p and q of S, there is a path in G between p and q, whose length is less than or equal to $t|pq|$. Any path satisfying this condition is called a *t-spanner path* between p and q.

In Figure 1.3, six geometric networks on 532 US cities are shown, each of which is a t-spanner for a different value of t. These spanners were computed by the path-greedy algorithm which will be presented in Section 1.4.

Definition 1.2.1 considers a spanner to be an undirected graph. Sometimes, it is useful to consider *directed* spanners:

Definition 1.2.2 (Directed spanner) Let S be a set of n points in $\mathbb{R}^d$ and let $t \geq 1$ be a real number. A *directed t-spanner* for S is a directed graph G with vertex set S, such that for any two points p and q of S, there is a directed path in G from p to q, whose length is less than or equal to $t|pq|$. Any path satisfying this condition is called a *directed t-spanner path* from p to q.

If G is a t-spanner for the point set S, then obviously, G is also a t'-spanner for any real number t' with $t' > t$. This leads to the following definition:

Definition 1.2.3 (Stretch factor) Let S be a set of n points in $\mathbb{R}^d$ and let G be a Euclidean graph with vertex set S. The *stretch factor* of G is the smallest real number t such that G is a t-spanner of S.

Property 1.1.1 and the definition of the minimum spanning tree imply the following property:

Property 1.2.4 *Let S be a set of n points in $\mathbb{R}^d$ and let $t \geq 1$ be a real number. Any t-spanner of S has at least $n-1$ edges, and weight at least $wt(MST(S))$.*

The complete graph is a 1-spanner, but it has a quadratic number of edges. In fact, if we assume that no three points of S are on a line, then the complete graph is the only 1-spanner for S. Therefore, t-spanners are in general only interesting for values of t that are larger than one. As you may expect, we are interested in *sparse* spanners, i.e., t-spanners with only $O(n)$ edges.

Basic spanner problem: Let S be a set of n points in $\mathbb{R}^d$, and let $t > 1$ be a real number. Does there exist a t-spanner for S having at most $c_{td}n$ edges, where c_{td} is a real number that only depends on t and d? If so, how much time does it take to compute such a t-spanner?

In this book, we will see that the answer to the first question is "yes": Sparse t-spanners exist for values of t that are arbitrarily close to one. Moreover, such t-spanners can be computed in $O(n \log n)$ time, where the constant in the Big-Oh bound depends on the stretch factor t and the dimension d.

In later chapters, we will also consider the existence and construction of sparse t-spanners having one or more of the following additional properties:

More spanner problems: Let S be a set of n points in $\mathbb{R}^d$, and let $t > 1$ be a real number.

1. Does there exist a t-spanner for S, in which each point has a *degree* that only depends on t and d?
2. Does there exist a t-spanner for S, whose *weight* is proportional to the weight of a minimum spanning tree of S?
3. Does there exist a sparse t-spanner for S, whose *spanner diameter* is small? Here, the spanner diameter is defined to be the smallest integer D, such that each pair of points is connected by a t-spanner path containing at most D edges.
4. Can we construct such t-spanners in $O(n \log n)$ time?

It should be clear that the above properties are conflicting. For example, any t-spanner whose degree is bounded by a constant must have spanner diameter $\Omega(\log n)$. The proof of this claim is left as an exercise; see Exer-

cise 1.11.

In most places in this book, the stretch factor t is a real number that is larger than, but arbitrarily close to, one. We will always assume that the dimension d is a constant. As such, Big-Oh bounds contain factors that depend on t. On the other hand, factors that only involve d will be omitted in these bounds.

1.3 Using spanners to approximate minimum spanning trees

In this section, we present a first application of spanners. Let S be a set of n points in the plane. Since any minimum spanning tree $MST(S)$ of S is contained in the Delaunay triangulation $DT(S)$ of S, we can compute $MST(S)$ in the following way: First, in $O(n \log n)$ time, compute $DT(S)$. Then, compute the minimum spanning tree of $DT(S)$. This minimum spanning tree is in fact a Euclidean minimum spanning tree of the set S. Since $DT(S)$ contains only a linear number of edges, the entire algorithm has a running time of $O(n \log n)$.

For point sets in $\mathbb{R}^d$, where $d \geq 3$, it is unlikely that the same running time can be obtained. In fact, for $d = 3$, it is even unlikely that an algorithm exists that computes a minimum spanning tree within a time bound that is significantly smaller than $n^{4/3}$. This leads to the natural question of whether there exist fast algorithms that *approximate* a minimum spanning tree. The following theorem shows that this question has a positive answer, provided we have a fast algorithm for computing a spanner.

Theorem 1.3.1 *Let S be a set of n points in $\mathbb{R}^d$, let $t > 1$ be a real number, and let G be an arbitrary t-spanner for S. A minimum spanning tree of G is a t-approximate minimum spanning tree of S, i.e., the weight of any minimum spanning tree of G is at most $t \cdot wt(MST(S))$.*

Proof. Let T be a minimum spanning tree of S, and number its edges arbitrarily as $e_1, e_2, \ldots, e_{n-1}$. Consider edge e_i. Since G is a t-spanner for S, there exists a t-spanner path P_i in G between the endpoints of e_i. Thus, the length $wt(P_i)$ of P_i is at most t times the length of edge e_i. It follows that

$$\sum_{i=1}^{n-1} wt(P_i) \leq \sum_{i=1}^{n-1} t \cdot wt(e_i) = t \cdot wt(MST(S)).$$

Let G' be the subgraph of G, whose edge set is the union of the edge sets of the paths P_i, $1 \leq i \leq n-1$. Then G' is a connected graph with vertex set S, and its weight is at most $t \cdot wt(MST(S))$. Since the weight of a minimum spanning tree of G is less than or equal to the weight of G', the proof is complete. ∎

1.4 A simple greedy spanner algorithm

At this point, it is not clear whether for any set S of n points in $\mathbb{R}^d$, and for any real number $t > 1$, a t-spanner for S having a subquadratic number of edges actually exists. In this section, we present a simple algorithm that, in fact, computes such a spanner.

As discussed before, t-spanners generalize the notion of spanning trees, which require to have paths connecting every pair of points. A t-spanner is required to have reasonably short paths between every pair of points. This observation suggests that an algorithm to construct t-spanners can be obtained by generalizing Kruskal's minimum spanning tree algorithm (which will be discussed in Section 2.6.1). We remark that this spanner algorithm does not necessarily compute a t-spanner of minimum weight.

Simple greedy spanner construction: Generalizing Kruskal's minimum spanning tree algorithm gives a greedy algorithm for constructing spanners. The algorithm starts with a graph G having vertex set S and whose edge set is empty. It considers all pairs of distinct points of S in non-decreasing order of their distances. The decision whether or not to add the current pair $\{p, q\}$ as an edge to G is made as follows: Instead of checking whether the vertices p and q are connected, check whether they have a path of length at most $t|pq|$ that connect them in G.

A formal description of this algorithm is given below.

Algorithm PATHGREEDY(S, t)

Comment: This algorithm takes as input a set S of n points in $\mathbb{R}^d$ and a real number $t > 1$. It returns a t-spanner for S.

```
sort the (n 2) pairs of distinct points in non-decreasing order of their
distances (breaking ties arbitrarily), and store them in list L;
E := ∅;
G := (S, E);
for each {p, q} ∈ L   (* consider pairs in sorted order *)
do δ := length of a shortest path in G between p and q;
    if δ > t|pq|
    then E := E ∪ {{p, q}};
         G := (S, E)
    endif
endfor;
output the graph G
```

Algorithm PATHGREEDY(S, t), with the value $t = n - 1$, produces the same output as Kruskal's minimum spanning tree algorithm, when given the complete Euclidean graph on S as input. The proof of this claim is left as an exercise (see Exercise 1.12).

The following lemma states the obvious fact that the path-greedy algorithm computes a t-spanner.

Lemma 1.4.1 *Let S be a set of n points in $\mathbb{R}^d$, and let $t > 1$ be a real number. The output of algorithm* PATHGREEDY(S, t) *is a t-spanner for S.*

Of course, many questions arise from the algorithm given above. Why is it not a minimum-weight t-spanner? Is there a non-trivial bound on the weight of the t-spanner? Is there a non-trivial bound on the number of edges included in the t-spanner? Is there a non-trivial bound on the degree of each vertex in the t-spanner? How can the algorithm be implemented efficiently? All these questions are dealt with in detail in Chapters 14 and 15.

Exercises

1.1 We mentioned in Section 1.1.1 that any set of n points has exactly n^{n-2} spanning trees. Verify this claim for small values of n.

1.2 (1) Let T be an arbitrary spanning tree of S. Prove that $wt(T) \leq (n-1) \cdot wt(MST(S))$.
(2) Let $\epsilon > 0$ be an arbitrarily small constant, and let n be a sufficiently large integer. Give an example of a set S of n points in the plane and a spanning tree T of S for which $wt(T) \geq (n - 1 - \epsilon) \cdot wt(MST(S))$.

1.3 Prove that a Steiner minimum tree really is a tree.

1.4 Consider a triangle with three vertices p, q, and r. Determine the minimum spanning tree and the Steiner minimum tree of these three points. There are two cases to consider, depending on whether or not all angles of the triangle are at most 120°.

1.5 Let S be the set of vertices of an equilateral triangle. Prove that $wt(MST(S)) = (2/\sqrt{3}) \cdot wt(SMT(S))$.

1.6 Prove that for any finite set S of points in $\mathbb{R}^d$,

$$wt(MST(S)) \leq wt(TSP(S)) \leq 2 \cdot wt(MST(S)).$$

1.7 Let S be a finite set of points in $\mathbb{R}^d$, and let S' be a non-empty subset of S. Prove the following monotonicity properties.

1. $wt(TSP(S')) \leq wt(TSP(S))$.
2. $wt(SMT(S')) \leq wt(SMT(S))$.

Prove that, in general, this monotonicity property does not hold for the minimum spanning tree.

1.8 Let S be a set of n points in the plane, not all on a straight line.
(1) Prove that a triangulation of S exists.
(2) Assume that no three points of S are collinear. Let h be the number of points on the convex hull of S. Prove that any triangulation of S has exactly $3n - 3 - h$ edges.

1.9 Get yourself a German dictionary, and find out the meaning of the word *Spanner*.

1.10 Let t be a real number with $1 < t < 2$, and consider an arbitrary t-spanner G for a set S of points in $\mathbb{R}^d$.
(1) Prove that each closest pair in S is connected by an edge in G.
(2) Let $p \in S$, and let q be a nearest neighbor of p in S. Are p and q connected by an edge in G?
(3) Now let $t = 2$. Prove that there is an edge in G whose endpoints form a closest pair in S.

1.11 Let G be an arbitrary connected graph with n vertices, and let $k \geq 1$ be an integer constant. Assume that each vertex of G has degree at most k. Prove that there are two vertices p and q such that *each* path between p and q contains $\Omega(\log n)$ edges.

1.12 Prove that algorithm PATHGREEDY(S, t) in Section 1.4, with the value $t = n-1$, is Kruskal's minimum spanning tree algorithm KRUSKAL(G), with G being the complete Euclidean graph on S, see Section 2.6.1.

1.13 Let S be a set of n points in $\mathbb{R}^d$, let $t > 1$ be a real number, and let G be the t-spanner that is computed by algorithm PATHGREEDY(S, t) in Section 1.4. Prove that G contains a minimum spanning tree of S.

1.14 In this exercise, we will introduce a spanner whose size depends on the lengths of the sides of the bounding box of the point set. Let S be a set of n points in the hypercube $[1, W]^d$, and choose real constants α and β such that $\alpha \geq \beta > 1$. Construct a sequence $S_0, S_1, \ldots, S_h$ of subsets of S, such that the following four properties hold:

1. $S_h \subseteq S_{h-1} \subseteq \ldots \subseteq S_1 \subseteq S_0 = S$.
2. $|S_h| = 1$ and $h = O(\log W)$.
3. For each i with $0 \leq i \leq h$, and for any two distinct points p and q in S_i, we have $|pq| \geq \beta^i$.
4. For each i with $0 \leq i < h$, and for each $p \in S_i$, there exists a point q in S_{i+1}, such that $|pq| \leq \beta^{i+1}$.

For each i with $0 \leq i \leq h$, define

$$E_i := \{\{p, q\} : p, q \in S_i \text{ and } |pq| \leq \alpha\beta^i\}.$$

Let G be the graph with vertex set S and edge set $E := \bigcup_{i=0}^{h} E_i$.
(1) Give an efficient algorithm that constructs the graph G. (Use a grid to compute the subsets $S_0, S_1, \ldots, S_h$.)
(2) Prove a tight upper bound on the number of edges of G.
(3) Assuming that $\alpha > 2\beta/(\beta - 1)$, prove that G is a t-spanner for

$$t = \max\left(\beta\frac{\alpha(\beta-1)+2\beta}{\alpha(\beta-1)-2\beta}, \frac{\alpha}{\beta}\right).$$

Bibliographic notes

It may be hard to believe, but spanners are known to occur in nature! Many rivers are known to be t-spanners for $t \approx \pi$. That is, the length of a river divided by the Euclidean distance between its source and mouth is roughly equal to π. See Stølum [1996].

History of spanners: While concepts similar to that of a spanner may have appeared in some form in earlier work from other related areas, its first appearance in the Computational Geometry literature can be traced to a paper by Chew [1986]. (The full version of this paper appeared in Chew [1989].) Chew defined the concept only for the complete Euclidean graph, although the word *spanner* was not used in his work. The formal definition of a spanner for an arbitrary graph in the general setting was first introduced about a year later (and, in fact, independently from Chew), in Peleg and Ullman [1987]. (The full version appeared as Peleg and Ullman [1989].) The term *spanners* was coined in that paper. Spanners were further investigated in the follow-up paper by Peleg and Schäffer [1989].

In the literature, stretch factor is also referred to by other terms such as *dilation* (in Peleg and Schäffer [1989]) and *distortion* (in Linial, London, and Rabinovich [1995]).

The field of geometric spanners has been surveyed by Soares [1992], Bern and Eppstein [1997], Eppstein [2000], Mitchell [2000], and Gudmundsson and Knauer [2006].

Related Topics: Much research has been done on spanners in general (i.e., non-geometric) graphs. A good starting point is the book by Peleg [2000]. Without being exhaustive, we also mention the papers by Peleg and Ullman [1987], Peleg and Schäffer [1989], Peleg and Ullman [1989], Cai and Corneil [1992, 1995a,b], Madanlal, Venkatesan, and Rangan [1996], Prisner [1997], Brandstädt, Chepoi, and Dragan [1999], and Le and Le [1999].

With regard to other related topics, excellent overviews of the history of the minimum spanning tree problem can be found in Graham and Hell [1985] and Nešetřil [1997]. The first algorithm for computing a minimum spanning tree was reported in two early papers [Borůvka, 1926a,b]. The most popular minimum spanning tree algorithms are the greedy algorithms by Kruskal [1956], Prim [1957], and Dijkstra [1959]. Many other algorithms exist for the minimum spanning tree problem; see the work of Fredman and Willard [1994], Karger, Klein, and Tarjan [1995], Chazelle [2000b], and Pettie and Ramachandran [2002].

Euclidean variants of the minimum spanning tree problem were studied by Yao [1982a], Vaidya [1988], Agarwal et al. [1991], Callahan and Kosaraju [1993], and Krznaric, Levcopoulos, and Nilsson [1999]. Good expected time algorithms were considered by Bentley and Friedman [1978], Bentley, Weide, and Yao [1980], Clarkson [1989], and Narasimhan and Zachariasen [2001]. The Euclidean minimum spanning tree of a set of n points in the plane can be computed in $O(n \log n)$ time. The fastest known algorithms for computing the minimum spanning tree of a three-dimensional point set have running times that are close to $O(n^{4/3})$; see Agarwal et al. [1991] and Callahan and Kosaraju [1993]. In Erickson [1995], it is argued that it is unlikely that a faster algorithm exists.

Several proofs of the fact that a set of n points has exactly n^{n-2} spanning trees can be found in the books by Aigner and Ziegler [2004] and van Lint and Wilson [1992]; see also Section 2.3.4.6 in Knuth [1997].

For excellent surveys on the topic of Steiner minimum trees, see Du and Hwang [1995] and the book by Hwang, Richards, and Winter [1992] (and the references given therein). The Steiner ratio conjecture was posed by Gilbert and Pollak [1968]. After twenty years, the conjecture was finally settled by Du and Hwang [1990a]; see also Du and Hwang [1990b] and Du and Hwang [1992]. Garey, Graham, and Johnson [1977] showed that computing Steiner minimum trees is **NP**-hard.

A captivating history of the traveling salesperson problem can be found in the book by Lawler et al. [1985], especially in the survey article by Johnson and Papadimitriou [1985]. For points in an arbitrary metric space, the problem of finding an optimal tour was shown to be **NP**-hard by Karp [1972]. For the Euclidean case, this was proved a few years later by Garey, Graham, and Johnson [1977] and Papadimitriou [1977].

The problem of designing good heuristics and approximation algorithms for the traveling salesperson problem has had a long and interesting history. For the general case in which the input does not necessarily satisfy the triangle inequality, Sahni and Gonzalez [1976] showed that computing any approximation to the optimal tour is **NP**-hard. The case when the points are from Euclidean space $\mathbb{R}^d$ was less intractable. A 2-approximation algorithm based on "doubling a minimum spanning tree" was designed by Rosenkrantz, Stearns, and Lewis [1977]; see also Exercise 1.6. For $d = 2$, their algorithm can be implemented to run in $O(n \log n)$ time. However, if the input also includes a planar graph that contains a minimum spanning tree (such as the Delaunay triangulation), then a minimum spanning tree can be computed in linear time using the algorithm of Cheriton and Tarjan [1976]. A 3/2-approximation algorithm based on minimum spanning trees, Euler tours, and minimum weight matchings was invented by Christofides [1976]. Improving this result was a major open problem for over two decades. Many heuristics and local optimization methods did not help in topping Christofides' result. Many of these heuristics are surveyed in Bentley [1992]. A breakthrough was achieved when Arora [1998] designed the first randomized polynomial-time approximation scheme for the Euclidean traveling salesperson problem, thus achieving a $(1+\epsilon)$-approximation algorithm with a time complexity of $O(n(\log n)^{(O(\sqrt{d}/\epsilon))^{d-1}})$ for points in $\mathbb{R}^d$. A similar result was discovered independently at about the same time by Mitchell [1999] for points in the plane. Rao and Smith [1998] took this one step further by designing a deterministic $(1+\epsilon)$-approximation algorithm with a time complexity of $O(n \log n)$ for points in d-dimensional space. More information about the latter result will be given in Chapter 19.

The upper bound on the number of triangulations of any planar point set that was mentioned in Section 1.1.4 appears in Santos and Seidel [2003]. The question of whether or not the minimum weight triangulation problem is in **P** or **NP**-hard appears in the book by Garey and Johnson [1979]. Some recent results about this triangulation and the greedy triangulation can be found in Krznaric [1997]. Triangulations can have small stretch factors. For example, the Delaunay triangulation of a set of points in the plane is a t-

spanner for $t = \frac{2\pi}{3\cos\pi/6} \approx 2.42$, see Keil and Gutwin [1992]; it is not known if it is a t-spanner for smaller values of t. It is known that t cannot be smaller than $\pi/2$. See also Section 20.3.

A solution to Exercise 1.14 can be found in Grünewald et al. [2002].

There are many excellent and useful books on computational geometry. These include the following: Mehlhorn [1984a]; Edelsbrunner [1987]; Preparata and Shamos [1988]; Agarwal [1991]; Mulmuley [1994]; de Berg et al. [2000]; Boissonnat and Yvinec [1998]; O'Rourke [1998]; Goodman and O'Rourke [2004]; Sack and Urrutia [2000]; Matoušek [1999]; Chazelle [2000a].

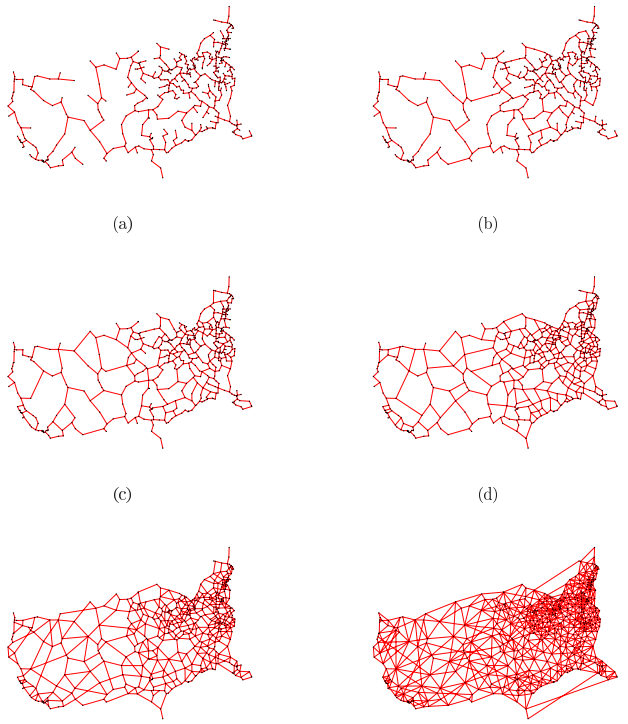


Figure 1.3: *Six geometric t-spanner networks on 532 US cities with stretch factors of (a)* 10*, (b)* 5*, (c)* 3*, (d)* 2*, (e)* 1.5*, and (f)* 1.2*, respectively.*

Chapter 2

Algorithms and graphs

> The Feynmann problem-solving algorithm: (1) write down the problem; (2) think very hard; (3) write down the answer.
>
> — Attributed to Murray Gell-mann.

2.1 Algorithms and data structures

Throughout this book, algorithms will be presented in either plain English or in *pseudocode*. Most of our algorithms will be designed for the *algebraic computation-tree model*. In this model, an algorithm works with arbitrary real numbers and can perform the operations of addition (+), subtraction (−), multiplication (∗), division (/), and the square root. Each of these operations takes one unit of time. Furthermore, an algorithm can test, again in one unit of time, whether or not any two real numbers, x and y, are equal, whether or not $x < y$, and whether or not $x \leq y$. Operations that are not allowed in our model of computation include the floor and ceiling functions, non-algebraic functions such as log, sin, cos, and tan, and bitwise operations on bit strings such as XOR. Unless stated otherwise, our algorithms will not use indirect addressing. A formal definition of the algebraic computation-tree model will be given in Chapter 3.

We will use standard data structures such as linked lists, heaps, Fibonacci heaps, balanced binary search trees, and skip lists, and we assume that the reader is familiar with these data structures.

The running time of an algorithm will be expressed as a function of the number n of input elements. This function counts the worst-case number of primitive operations made by the algorithm on any input of length n. Actually, the input will generally be a set of n points in $\mathbb{R}^d$ and, hence, it consists of a sequence of dn real numbers. Since we assume, however, that the dimension d is a constant, it is realistic to consider n to be the length of the input.

We will use the standard asymptotic notation to estimate the running times of algorithms. Let $f : \mathbb{N} \longrightarrow \mathbb{N}$ and $g : \mathbb{N} \longrightarrow \mathbb{N}$ be two functions.

1. $f(n) = O(g(n))$ if there are constants n_0 and C such that $f(n) \leq C \cdot g(n)$ for all $n \geq n_0$.
2. $f(n) = \Omega(g(n))$ if there are constants n_0 and C such that $f(n) \geq C \cdot g(n)$ for all $n \geq n_0$.
3. $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.
4. $f(n) = o(g(n))$ if $\lim_{n\to\infty} f(n)/g(n) = 0$. (Here, we assume that $g(n) \neq 0$ for all sufficiently large n.)

2.2 Some notions from graph theory

2.2.1 Graphs

An *undirected graph* is a pair $G = (V, E)$, where V is a set whose elements are called *vertices*, and E is a set consisting of pairs of vertices. Any element of E is called an *edge* and has the form $\{u, v\}$ for some distinct vertices u and v in V. We also say that u and v are *connected by an edge*. Observe that $\{u, v\}$ and $\{v, u\}$ denote the same edge of E. Throughout this book, we will only consider finite graphs, i.e., graphs whose vertex set is finite.

If u and v are two vertices of an undirected graph $G = (V, E)$, then a *path* between u and v is a sequence $u_0, u_1, u_2, \ldots, u_k$ of vertices of V for some $k \geq 0$ such that $u = u_0$, $u_k = v$, and $\{u_i, u_{i+1}\} \in E$, for all i with $0 \leq i < k$. The path is called *simple* if all its vertices are pairwise distinct. The path is called a *cycle* if all its vertices are pairwise distinct, except that $u = v$.

Two vertices u and v of an undirected graph $G = (V, E)$ are said to be *connected by a path* if there is a path in G between u and v. Observe that each vertex is connected to itself. We say that G is *connected* if each pair of vertices is connected by a path. A *connected component* of G is a maximal subset of V, all of whose elements are pairwise connected by paths.

The *degree* of a vertex u in an undirected graph $G = (V, E)$ is defined as the number of edges that contain u as a vertex. We denote the degree of u by $deg(u)$. The *degree* of G is the maximum degree of any of its vertices.

An undirected graph $G = (V, E)$ is called *weighted* if each of its edges e has a weight $wt(e)$, which is a real number. If $e = \{u, v\}$, then we write $wt(u, v)$ instead of $wt(\{u, v\})$. The weight of a path $u_0, u_1, u_2, \ldots, u_k$ is defined as $\sum_{i=0}^{k-1} wt(u_i, u_{i+1})$.

We say that an undirected weighted graph $G = (V, E)$ satisfies the *triangle inequality* if $wt(u, v) \leq wt(u, w) + wt(w, v)$ for any three edges $\{u, v\}$, $\{u, w\}$, and $\{v, w\}$ of E.

The *complete graph* on a set V is the undirected graph $G = (V, E)$ for which E is the set of all $\binom{|V|}{2}$ pairs of vertices of V.

A *directed graph* is a pair $G = (V, E)$, where, again, V is a finite set of vertices, but now, E is a set of directed edges of the form (u, v) for some distinct vertices u and v of V. We say that u is the *source* and v is the *sink* of the edge. The notions of a path and cycle are defined similarly as for undirected graphs, the only difference being that edges are considered "one-way streets". The *outdegree* of a vertex u is defined as the number of edges having u as a source, whereas the *indegree* of u is the number of edges having u as a sink. The *degree* of u is the sum of its indegree and outdegree.

An *embedding* of a directed or undirected graph $G = (V, E)$ is obtained by mapping each vertex of V to a point in the plane, and each edge of E to a straight-line segment joining the two vertices of the edge. We require that the vertices are mapped to pairwise distinct points. The embedding is called *plane* if no two edges in the embedding intersect, except possibly at their endpoints. The graph G is called *planar* if it admits a plane embedding. If G is planar then $|E| \leq 3|V| - 6$.

2.2.2 Geometric networks

The *Euclidean distance* $|pq|$ between any two points $p = (p_1, p_2, \ldots, p_d)$ and $q = (q_1, q_2, \ldots, q_d)$ in $\mathbb{R}^d$ is defined as

$$|pq| := \left((q_1 - p_1)^2 + (q_2 - p_2)^2 + \ldots + (q_d - p_d)^2\right)^{1/2}.$$

In this book, we will mainly consider *geometric networks* (or *Euclidean graphs*). Let S be a finite set of points in $\mathbb{R}^d$. A geometric network on S is a graph $G = (S, E)$, in which the weight of each edge is defined as the Euclidean distance between its vertices. The weight of a path in such a graph will also be referred to as its *length*. Hence, if $t > 1$ is a real number, then G is a t-spanner for S if any two points p and q of S are connected in G by a path of length at most $t|pq|$.

2.2.3 Trees

An undirected graph $T = (V, E)$ is called a *tree* if T is connected and has no cycles. The vertices of T are also called *nodes*. Observe that T has exactly $|V| - 1$ edges.

Let $T = (V, E)$ be a tree and let r be an arbitrary node which we call the *root* of T. For any node v of T with $v \neq r$, the *parent* of v is defined to be the first node different from v on the (unique) path in T from v to r. If u is a node of T and v is the parent of u, then we say that u is a *child* of v. If v does not have any children, then v is called a *leaf* of the rooted tree T; otherwise, v is called an *internal* node. Any node u on the path between v and r is called an *ancestor* of v; if $u \neq v$, then u is a *proper ancestor* of v. Similarly, any node u for which the path between u and r contains node v is called a *descendent* of v; if $u \neq v$, then u is a *proper descendent* of v. The *subtree* of v is the tree induced by all descendents of v.

A rooted tree T is called a *binary tree* if each node has at most two children. (Observe that a binary tree with more than one node has degree three.) In case a node v has two children, we call one of them the *left child* and the other one the *right child*.

2.2.4 Representing graphs

Let $G = (V, E)$ be a graph with n vertices and m edges. We assume for simplicity that G is undirected. We number the vertices of V arbitrarily as

$v_1, v_2, \ldots, v_n$. There are basically two different ways to store G in a data structure.

The first one uses an *adjacency matrix*. This is an $n \times n$ matrix M, where $M[i,j] = 1$ if $\{v_i, v_j\}$ is an edge of E, and $M[i,j] = 0$ otherwise. The advantage of this representation is that we can decide in constant time if any two vertices v_i and v_j are connected by an edge. (Observe that we need the indirect-addressing operation for this.) The disadvantages are that it takes $O(n)$ time to report all edges that are incident to a given vertex of V, and the amount of space used is always $\Theta(n^2)$, irrespective of the size m of E. Since we will mainly consider *sparse* graphs in this book, i.e., graphs having $m = O(n)$ edges, we are interested in data structures that need only $O(n+m)$ space.

In an *adjacency lists* representation, we store with each vertex v_i a list containing all vertices that are connected by an edge to v_i. The total amount of space needed is proportional to n plus the sum of the degrees of all vertices, i.e., it is $O(n+m)$. Using this representation, we can report all edges that are incident to a given vertex v_i in $O(deg(v_i))$ time. The same amount of time is needed to check if v_i is connected by an edge to any given vertex v_j.

Observe that both representations of undirected graphs can easily be extended to directed graphs.

2.3 Some algorithms on trees

2.3.1 Traversing a binary tree

In many algorithms that operate on trees, it is necessary to traverse the nodes of the tree. The order in which the nodes are visited during this traversal depends on the problem at hand. In this section, we introduce three different orderings. We will see in Section 2.3.2 how they can be used to solve some specific problems.

Let T be a rooted tree. We assume for simplicity that each internal node has exactly two children. We also assume that each internal node has pointers to its two children, and that each node (except the root) has a pointer to its parent.

The three traversals are defined recursively. If T contains only one node, then each of the three traversals is the node itself. Otherwise, the *postorder traversal* is the recursively defined postorder traversal of the left subtree of

T's root, followed by the recursively defined postorder traversal of the right subtree of T's root, followed by the root itself. The *inorder traversal* is the recursively defined inorder traversal of the left subtree of T's root, followed by the root, followed by the recursively defined inorder traversal of the right subtree of T's root. Finally, the *preorder traversal* is the root, followed by the recursively defined preorder traversal of the left subtree of T's root, followed by the recursively defined preorder traversal of the right subtree of T's root.

The amount of time needed by each of these three traversals is proportional to the number of nodes of T.

2.3.2 Lowest common ancestors

Let T be a rooted tree. As in the previous section, we assume for simplicity that each internal node has exactly two children. The *lowest common ancestor* of any two nodes u and v is the node that is an ancestor of both u and v, and that is farthest away from the root of T. Equivalently, it is the node in which the two paths from the root to nodes u and v diverge.

In this section and Section 2.3.3, we show how to represent the tree T in such a way that lowest common ancestor queries can be answered efficiently. In such a query, we are given two arbitrary nodes u and v of T, and have to compute their lowest common ancestor. Clearly, we may assume without loss of generality that $u \neq v$.

Let n denote the number of leaves of the tree T. For any node u of T, let $size(u)$ denote the number of leaves in the subtree of u, and let $\ell(u) := \lfloor \log size(u) \rfloor$. Observe that $\ell(u)$ is an integer in the range from zero to $\lfloor \log n \rfloor$. Moreover, on the path from any leaf to the root of T, the $\ell(u)$-values form a non-decreasing sequence.

We will use the values $\ell(u)$ to partition the tree T into a collection of pairwise disjoint paths. For any node u of T, let P_u be the subgraph of T consisting of all nodes v for which (i) $\ell(v) = \ell(u)$, and (ii) $\ell(w) = \ell(u)$ for all nodes w on the path in T between u and v. It is not difficult to prove that P_u is a path (see Exercise 2.1).

We extend the tree T by storing with each node u the value of $size(u)$, and a pointer to the node $gpar(u)$ of P_u that is closest to the root. We call $gpar(u)$ the *group parent* of u. We denote the parent of node u by $p(u)$. It is easy to prove that if two distinct nodes u and v have the same group parent, then either u is an ancestor of v, or v is an ancestor of u (see Exercise 2.2).

Let u be any node of T, and consider the sequence

$$u, gpar(u), p(gpar(u)), gpar(p(gpar(u))), p(gpar(p(gpar(u)))), \ldots$$

of nodes, which terminates at the root of T. This sequence consists of at most $2\lfloor \log n \rfloor + 1$ nodes. Hence, we can walk from any node to the root of T in $O(\log n)$ time. We generalize this approach to solve lowest common ancestor queries, as shown in algorithm LCA.

Algorithm LCA(u, v, T)

Comment: This algorithm takes as input two nodes u and v of T, and returns their lowest common ancestor. We assume for simplicity that $u \neq v$, and neither u nor v is the root of T.

Step 1: Compute the sequence $g_0^u, g_1^u, \ldots, g_i^u$ of nodes in T where $g_0^u := u$, $g_1^u := gpar(u)$, $g_k^u := gpar(p(g_{k-1}^u))$ for $2 \leq k \leq i$, and g_i^u is the first node in this sequence that is equal to the root of T. Since u is not the root, we have $i \geq 1$.

Step 2: Compute the sequence $g_0^v, g_1^v, \ldots, g_j^v$ of nodes in T where $g_0^v := v$, $g_1^v := gpar(v)$, $g_k^v := gpar(p(g_{k-1}^v))$ for $2 \leq k \leq j$, and g_j^v is the first node in this sequence that is equal to the root of T. As in Step 1, we have $j \geq 1$.

Step 3: Compute the integer k such that $g_i^u = g_j^v$, $g_{i-1}^u = g_{j-1}^v$, $\ldots$, $g_{i-k+1}^u = g_{j-k+1}^v$, and $g_{i-k}^u \neq g_{j-k}^v$. Since $g_i^u = g_j^v$, we have $k \geq 1$. Obviously, $k \leq \min(i, j)$.

Step 4: There are four possible cases to consider:

Case 4.1: $i = j = k$.
Since $g_1^u = g_1^v$, we have $gpar(u) = gpar(v)$ and, hence, one of u and v is an ancestor of the other. Therefore, the algorithm returns u if $size(u) \geq size(v)$, and it returns v otherwise.

Case 4.2: $i \neq j$ and $k = i$.
Since $u = g_0^u \neq g_{j-k}^v$ and $gpar(u) = g_1^u = g_{j-k+1}^v$, one of the nodes u and $p(g_{j-k}^v)$ (both are on the path P_u) is the lowest common ancestor of u and v. Therefore, the algorithm returns u if $size(u) \geq size(p(g_{j-k}^v))$, and it returns $p(g_{j-k}^v)$ otherwise.

Case 4.3: $i \neq j$ and $k = j$.
This case is symmetric to Case 4.2. The algorithm returns v if $size(v) \geq size(p(g_{i-k}^u))$, and it returns $p(g_{i-k}^u)$ otherwise.

Case 4.4: $k \neq i$ and $k \neq j$.
Since $g_{i-k+1}^u = g_{j-k+1}^v$ and $g_{i-k}^u \neq g_{j-k}^v$, one of the nodes $p(g_{i-k}^u)$ and $p(g_{j-k}^v)$ is the lowest common ancestor of u and v. Therefore, the algorithm returns $p(g_{i-k}^u)$ if $size(p(g_{i-k}^u)) \geq size(p(g_{j-k}^v))$, and it returns $p(g_{j-k}^v)$ otherwise.

This completes the description of algorithm LCA for computing the lowest common ancestor of any two nodes of T. It is not difficult to see that the running time is $O(\log n)$. It remains to show how to preprocess the tree T. That is, we have to show how to compute the values $size(u)$ and $\ell(u)$, and the group parents $gpar(u)$.

We start with the computation of the values $size(u)$ and $\ell(u)$. The values $size(u)$ can be computed by traversing the tree T in postorder. From these, we can compute $\ell(u)$ as $\ell(u) = \lfloor \log size(u) \rfloor$. The disadvantage of this approach is that we need the floor and logarithm functions for this. The following observation implies that we can avoid these functions.

Observation 2.3.1 *Let u be an internal node of T and let v and w be the two children of u. Assume that $size(v) \leq size(w)$.*

1. *If $size(v) + size(w) \geq 2^{\ell(w)+1}$, then $\ell(u) = \ell(w) + 1$.*

2. *If $size(v) + size(w) < 2^{\ell(w)+1}$, then $\ell(u) = \ell(w)$.*

Algorithm SizeAndLValues(u) takes as input a node u of T. It traverses the subtree rooted at u in postorder, and returns the value $2^{\ell(u)}$. After this algorithm has terminated, the values $size(v)$ and $\ell(v)$ for all nodes v that are in the subtree of u have been computed. Hence, we obtain the values $size(v)$ and $\ell(v)$ for all nodes v of T, by calling SizeAndLValues(u) with u being the root. The running time for this call is $O(n)$.

Algorithm SizeAndLValues(u)

Comment: This algorithm computes the values $size(v)$ and $\ell(v)$ for all nodes v that are in the subtree of u, and it returns the value $2^{\ell(u)}$.

```
if u is a leaf
then size(u) := 1;
     ℓ(u) := 0;
     x := 1;
     return x
else v := left child of u;
     w := right child of u;
     x := SizeAndLValues(v);
     y := SizeAndLValues(w);
     (* x = 2^ℓ(v) and y = 2^ℓ(w) *)
     size(u) := size(v) + size(w);
     if size(w) < size(v)
     then swap v and w, and swap x and y
     endif;
     (* size(v) ≤ size(w) *)
     if size(u) ≥ 2y
     then ℓ(u) := ℓ(w) + 1;
          z := 2y;
          return z
     else ℓ(u) := ℓ(w);
          return y
     endif
endif
```

The algorithm that computes the group parents is based on a preorder traversal of the tree T. This algorithm, denoted by GroupParents(u, x), takes as input two nodes u and x such that $gpar(u) = x$. It computes the group parents of all nodes in the subtree rooted at u. If we denote the root of T by r, then a call to GroupParents(r, r) computes the group parents of all nodes of the tree. The time for this call is $O(n)$.

Algorithm GROUPPARENTS(u, x)

Comment: This algorithm takes as input two nodes u and x of the tree T, such that $gpar(u) = x$. It computes the group parents of all nodes in the subtree rooted at u.

```
gpar(u) := x;
if u is not a leaf
then v := left child of u;
     w := right child of u;
     if ℓ(u) = ℓ(v)
     then GROUPPARENTS(v, x)
     else GROUPPARENTS(v, v)
     endif;
     if ℓ(u) = ℓ(w)
     then GROUPPARENTS(w, x)
     else GROUPPARENTS(w, w)
     endif
endif
```

We have described all algorithms for trees in which each internal node has exactly two children. Observe that such a tree with n leaves has exactly $2n - 1$ nodes. We leave it to the reader to generalize the algorithms to arbitrary rooted trees. This gives the following result:

Theorem 2.3.2 *Let T be a rooted tree with n nodes. We can preprocess T in $O(n)$ time such that lowest common ancestor queries can be answered in $O(\log n)$ time.*

2.3.3 A faster algorithm for lowest common ancestor queries

The algorithm in Section 2.3.2 works in the algebraic computation-tree model. In this section, we add the *indirect-addressing* operation as a unit-time operation to this model. We will show that in this more powerful model, lowest common ancestor queries can be answered in $O(1)$ time, after an $O(n)$–time

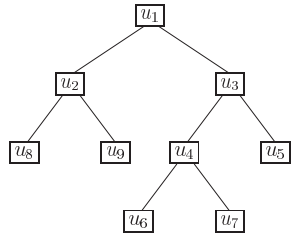
preprocessing step. This algorithm can in fact be implemented in the algebraic computation-tree model so that after an $O(n)$–time preprocessing, the lowest common ancestor queries can be answered in $O(\log \log n)$ time.

We start by reducing the lowest common ancestor problem on a tree with n nodes to the so-called *range minimum problem* on an array of $O(n)$ real numbers. Then we show how range minimum queries can be answered in $O(1)$ time, after an $O(n)$–time preprocessing of the array.

Reduction to the range minimum problem

Let T be a rooted tree with n nodes. We again assume for simplicity that each internal node has exactly two children. The *level* of any node u of T is the number of edges on the path from the root to u.

We number the nodes of T arbitrarily as $u_1, u_2, \ldots, u_n$. Consider an *Euler tour* of the "double-tree" obtained by doubling each edge of T. This tour starts at the root, visits each edge of T exactly twice, and returns to the root. Let $E[1 \,..\, 2n-1]$ be the array that stores the nodes of T in the order in which they occur in the Euler tour. To give an example, consider the following tree T with root u_1.



The corresponding array E is shown below with the array indices in the top row and the array entries in the bottom row:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
u_1	u_2	u_8	u_2	u_9	u_2	u_1	u_3	u_4	u_6	u_4	u_7	u_4	u_3	u_5	u_3	u_1

Since we assume that each node has zero or two children, each internal node of T occurs exactly three times in E, whereas each leaf occurs exactly once. Let $L[1\,..\,2n-1]$ be the array where $L[i]$ stores the level of node $E[i]$, $1 \leq i \leq 2n-1$. For our example tree, we obtain the following array L, again, with the array indices in the top row:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	1	2	1	2	1	0	1	2	3	2	3	2	1	2	1	0

Finally, let $R[1\,..\,n]$ be the array in which $R[i]$ is equal to the smallest integer ℓ for which $E[\ell] = u_i$, $1 \leq i \leq n$. In other words, $R[i]$ is equal to the first "time" that node u_i is visited during the Euler tour of T. For our example tree, we get the following array R:

1	2	3	4	5	6	7	8	9
1	2	8	9	15	10	12	3	5

The following lemma explains how the arrays E, L, and R can be used to obtain the lowest common ancestor of any two distinct nodes of the tree T.

Lemma 2.3.3 *Let i and j be distinct integers with $1 \leq i \leq n$ and $1 \leq j \leq n$, and assume that $R[i] < R[j]$. Let k be the integer such that $R[i] \leq k \leq R[j]$ and $L[k]$ is minimum. The lowest common ancestor of the nodes u_i and u_j is equal to $E[k]$.*

Proof. The lowest common ancestor of u_i and u_j is visited during the portion of the Euler tour that starts at the first visit to u_i and ends at the first visit to u_j. Among all nodes visited during this portion, it is that node that is closest to the root of T. This portion of the Euler tour is stored in $E[R[i]\,..\,R[j]]$, and the levels of the nodes of this portion are stored in $L[R[i]\,..\,R[j]]$. ∎

Hence, we can answer lowest common ancestor queries by answering a so-called range minimum query in the array L. We will show later how such queries can be answered in $O(1)$ time after an $O(n)$–time preprocessing step, by using the following property of L.

Observation 2.3.4 *The array L satisfies the* (±1)-property*, i.e., $L[i+1]$ is equal to either $L[i]+1$ or $L[i]-1$ for each i with $1 \leq i \leq 2n-2$.*

Given the tree T, the arrays E, L, and R can be computed in $O(n)$ time. Moreover, the reduction given above can easily be extended to arbitrary rooted trees. Therefore, we obtain the following result:

Theorem 2.3.5 *Let T be a rooted tree with n nodes. We can preprocess T in $O(n)$ time such that lowest common ancestor queries can be answered in $O(1)$ time. The algorithms work in the algebraic computation-tree model, extended with indirect-addressing.*

The $O(n)$–time preprocessing algorithm can be implemented in the algebraic computation-tree model (i.e., without using indirect-addressing). In this weaker model, the query time becomes $O(\log \log n)$. This will imply the following result:

Theorem 2.3.6 *Let T be a rooted tree with n nodes. We can preprocess T in $O(n)$ time such that the lowest common ancestor queries can be answered in $O(\log \log n)$ time. The algorithms work in the algebraic computation-tree model.*

Solving the range minimum problem

In the *range minimum problem*, we are given an array $A[1\,..\,n]$ of real numbers. We want to preprocess A such that for any two given integers i and j with $1 \leq i < j \leq n$, we can efficiently compute an integer k with $i \leq k \leq j$, for which $A[k]$ is minimum.

In the application to the lowest common ancestor problem, we have to solve this problem for the array $L[1\,..\,2n-1]$. Observe that for this application, we need the *index* of the minimal element rather than the minimal element itself.

We start by giving two simple solutions to this problem, both of which do not assume any restriction on the array A. These solutions use $O(n^2)$ and $O(n \log n)$ preprocessing time (and space), respectively. Afterwards, we show how these solutions can be combined to obtain an $O(n)$–time preprocessing algorithm for arrays that satisfy the (±1)-property.

A first solution: In this solution to the range minimum problem, we store all $\binom{n}{2}$ possible answers. That is, for each i with $1 \leq i < n$, there is an array $X_i[i+1\,..\,n]$, where for each j with $i+1 \leq j \leq n$, the value of $X_i[j]$ is equal

to the integer k with $i \leq k \leq j$, for which $A[k]$ is minimum. It is clear that these arrays can be used to answer a range minimum query in $O(1)$ time. Using the relations

$$X_i[i+1] = \begin{cases} i & \text{if } A[i] < A[i+1], \\ i+1 & \text{otherwise,} \end{cases}$$

for $1 \leq i < n$, and

$$X_i[j] = \begin{cases} X_i[j-1] & \text{if } A[X_i[j-1]] < A[j], \\ j & \text{otherwise,} \end{cases}$$

for $1 \leq i < n-1$ and $i+2 \leq j \leq n$, all arrays X_i, $1 \leq i < n$, can be computed in $O(n^2)$ time.

A second solution: We improve upon the previous solution, by storing for each i with $1 \leq i < n$, the answers to only $O(\log n)$ different range minimum queries. To be more precise, for each i with $1 \leq i < n$, there is an array $Y_i[1 \,..\, \lfloor \log(n-i+1) \rfloor]$, in which for each ℓ with $1 \leq \ell \leq \lfloor \log(n-i+1) \rfloor$, the value of $Y_i[\ell]$ is equal to the integer k with $i \leq k \leq i + 2^\ell - 1$, for which $A[k]$ is minimum.

Let us see how these arrays can be used to answer a range minimum query. Let i and j be two integers with $1 \leq i < j \leq n$ and let $h := \lfloor \log(j-i) \rfloor$. If $h = 0$, then $j = i+1$ and $Y_i[1]$ is the answer to the query. Assume that $h \geq 1$. We observe that $j - 2^h + 1 \leq i + 2^h$ and, therefore,

$$\{i, i+1, \ldots, i+2^h-1\} \cup \{j-2^h+1, j-2^h+2, \ldots, j\} = \{i, i+1, \ldots, j\}.$$

Let $k := Y_i[h]$ and $k' := Y_{j-2^h+1}[h]$. (Observe that h is in the range of both Y_i and Y_{j-2^h+1}.) If $A[k] < A[k']$, then $A[k]$ is a minimal element in the subarray $A[i \,..\, j]$ and, therefore, k is the answer to the query. Otherwise, k' is the answer. Hence, assuming that the value of h can be computed in $O(1)$ time, we can answer a range minimum query in $O(1)$ time.

If we initialize an array $Z[1 \,..\, n]$, where $Z[m] = \lfloor \log m \rfloor$, $1 \leq m \leq n$, then h is obtained in $O(1)$ time by looking up the value $Z[j-i]$. This array can be computed in $O(n)$ time, without using the floor and logarithm functions.

The relations

$$Y_i[1] = \begin{cases} i & \text{if } A[i] < A[i+1], \\ i+1 & \text{otherwise,} \end{cases}$$

for $1 \leq i < n$, and

$$Y_i[\ell] = \begin{cases} Y_i[\ell-1] & \text{if } A[Y_i[\ell-1]] < A[Y_{i+2^{\ell-1}}[\ell-1]], \\ Y_{i+2^{\ell-1}}[\ell-1] & \text{otherwise,} \end{cases}$$

for $1 \leq i < n-1$ and $2 \leq \ell \leq \lfloor \log(n-i+1) \rfloor$, imply that all arrays Y_i, $1 \leq i < n$, can be computed in $O(n \log n)$ time.

Hence, we have shown that the array A can be preprocessed in $O(n \log n)$ time, such that range minimum queries can be answered in $O(1)$ time.

An optimal solution: We now assume that the array $A[1 \,..\, n]$ satisfies the (± 1)-property. Furthermore, we assume for simplicity that $\log n$ is an even integer.

We partition A into *blocks* of length $(1/2) \log n$, where the h-th block is the subarray

$$A[((h-1)/2) \log n + 1, ((h-1)/2) \log n + 2, \ldots, (h/2) \log n],$$

for $1 \leq h \leq \lceil 2n / \log n \rceil$.

Let $B[1 \,..\, \lceil 2n/\log n \rceil]$ be the array in which $B[h]$ is equal to the minimal element of the h-th block for $1 \leq h \leq \lceil 2n/\log n \rceil$. The array B can be computed in $O(n)$ time. We use our second solution to preprocess B for range minimum queries. This takes $O(n)$ time, after which queries in B can be answered in $O(1)$ time.

We also initialize an array $C[1 \,..\, n]$ in which $C[i] = \lfloor 2i / \log n \rfloor$ for $1 \leq i \leq n$. We will use this array to compute, given any i with $1 \leq i \leq n$, the number of the block that contains $A[i]$. The array C can be computed in $O(n)$ time, without using the floor function.

Let i and j be two integers with $1 \leq i < j \leq n$, let $h := \lceil 2i / \log n \rceil$, and let $h' := \lceil 2j / \log n \rceil$. Hence, $A[i]$ and $A[j]$ are contained in the h-th and h'-th blocks of A, respectively. Let us first consider the case when $h < h'$. The minimal value in the subarray $A[i \,..\, j]$ is equal to the minimum of the following three numbers:

1. the minimum in the portion of the h-th block that starts at position i and ends at the end of this block,

2. the minimum in the portion of the h'-th block that starts at the beginning of this block and ends at position j, and

3. the minimum in the subarray $B[h+1\,..\,h'-1]$. (Let us say that this minimum is ∞ if $h' = h+1$.)

The third minimum can be computed in $O(1)$ time using the data structure for the array B. In order to compute the first and second minima, we have to preprocess the blocks so that *in-block* range minimum queries can be answered. If $h = h'$, then $A[i]$ and $A[j]$ are contained in the same block and the query on A reduces to an in-block query.

Let us see how the blocks can be preprocessed for in-block queries. Consider the *normalized* blocks obtained by taking each block and subtracting the first element of the block from each element of the block. The critical point is that even though there are $\lceil 2n/\log n\rceil$ blocks, we will show that the number of distinct normalized blocks is only $O(\sqrt{n})$. To see this, we observe that there are $\lceil 2n/\log n\rceil$ normalized blocks, each

- having length $(1/2)\log n$,
- starting with a zero, and
- satisfying the (± 1)-property.

A normalized block can be uniquely encoded by a string of length $(1/2)\log n - 1$ over the alphabet $\{+1, -1\}$. We observe that relative to the start of a block, the location of the minimal item within that block is dependent only on the string encoding its normalized block and is independent of the value of its first item. Since the number of such strings is only $2^{(1/2)\log n - 1} = (1/2)\sqrt{n}$, it follows that we only have to preprocess up to $(1/2)\sqrt{n}$ many "distinct" blocks.

Our preprocessing proceeds as follows. For each h with $1 \le h \le \lceil 2n/\log n\rceil$, let $s^h := s_1 s_2 \ldots s_{(1/2)\log n - 1}$ be the (± 1)-string corresponding to the h-th block of A, and let the integer x^h be defined by

$$x^h := \sum_{i=1}^{(1/2)\log n - 1} \frac{s_i + 1}{2} 2^i.$$

That is, we replace in s^h each occurrence of -1 by 0, and each occurrence of $+1$ by 1. The resulting binary string is the binary representation of the integer x^h. Each of the $\lceil 2n/\log n\rceil$ integers x^h can be computed in $O(\log n)$ time. Also, we can sort all these integers in $O(n)$ time. After this sorting

step, we know the "distinct" blocks of A. For each distinct value of x^h, we preprocess the corresponding normalized block for range minimum queries using our first solution. Since each normalized block has length $O(\log n)$, and since we do this for only $O(\sqrt{n})$ blocks, this part of the preprocessing takes $O(\sqrt{n}\log^2 n) = O(n)$ time.

Finally, for each h with $1 \le h \le \lceil 2n/\log n\rceil$, we store with the h-th block of the array A a pointer to the data structure for the normalized block that corresponds to the integer x^h.

We now present the query algorithm. Given two integers i and j with $1 \le i < j \le n$, we compute $h := \lceil 2i/\log n\rceil$ and $h' := \lceil 2j/\log n\rceil$. If $h < h'$, then we do the following:

1. We follow the pointer to the data structure for the normalized block that corresponds to x^h and compute the index of the minimal element in the portion that starts at position i and ends at the end of this normalized block.
2. We follow the pointer to the data structure for the normalized block that corresponds to $x^{h'}$ and compute the index of the minimal element in the portion that starts at the beginning of this normalized block and ends at position j.
3. If $h < h' - 1$, then we use the data structure for B to compute the index of the minimal element in the subarray $B[h+1\,..\,h'-1]$.

It should be clear that, given this information, the index of the minimal element in the subarray $A[i\,..\,j]$ can be computed in $O(1)$ time. If $h = h'$, then we follow the pointer to the data structure for the normalized block that corresponds to x^h, and answer the query within this normalized block. The answer to this query allows us to find, in $O(1)$ time, the index of the minimal element in the subarray $A[i\,..\,j]$. We have proved the following result:

Theorem 2.3.7 *Let $A[1\,..\,n]$ be an array of real numbers satisfying the (± 1)-property. We can preprocess A in $O(n)$ time such that range minimum queries can be answered in $O(1)$ time. The algorithms work in the algebraic computation-tree model, extended with indirect-addressing.*

In the algorithms given above, indirect-addressing is a crucial operation. We claim that these algorithms can be implemented without using indirect-addressing, at the expense of an increase in the query time:

Theorem 2.3.8 *Let $A[1\,..\,n]$ be an array of real numbers satisfying the (± 1)-property. We can preprocess A in $O(n)$ time such that range minimum queries can be answered in $O(\log\log n)$ time. The algorithms work in the algebraic computation-tree model.*

2.3.4 Centroids and separators in trees

Divide-and-conquer is a standard technique in algorithm design. When applying this technique to trees, we have to partition the tree into a small number of pieces such that each piece contains at most a constant fraction of the nodes. In this section, we will introduce the notion of centroid node, centroid edge, and separator node. Each of these can be used to achieve such a partition.

Before we can define centroid nodes, we have to introduce some notation. We denote the number of nodes of any tree T by $\#(T)$. We say that two nodes u and v of T are *neighbors*, if they are connected by an edge.

Consider any node u of T. Removing node u (and its incident edges) from T results in a collection of subtrees with one subtree for each neighbor of u. We denote by T_{uv} the subtree that contains neighbor v. Observe that $T_{uw} \neq T_{vu}$. In fact, we have $\#(T_{uw}) + \#(T_{vu}) = \#(T)$.

A node u of the tree T is called a *centroid node* if T_w contains at most $\#(T)/2$ nodes for each neighbor v of u. The following theorem states that such a node always exists and that it can be computed efficiently.

Theorem 2.3.9 *Let $n \geq 2$. Any tree having n nodes contains a centroid node, which can be computed in $O(n)$ time.*

Proof. For each node v of T, we define

$$m_v := \max\{\#(T_{vw}) : \ w \text{ is a neighbor of } v\}.$$

Let u be a node of T for which m_u is minimum. We will show that u is a centroid node of T.

Let v be a neighbor of u such that the tree T_w contains m_u vertices. Let $u, w_1, w_2, \ldots, w_k$ be the neighbors of v in T. Observe that, by our choice of u, we have $m_v \geq m_u$. Let x be a neighbor of v such that the tree T_{vx} contains m_v nodes. Then $x \in \{u, w_1, w_2, \ldots, w_k\}$. We claim that $x = u$. To prove this, assume there is an index i with $1 \leq i \leq k$, such that $x = w_i$.

Then $\#(T_{vw_i}) < \#(T_{uw})$, which is a contradiction, because $\#(T_{vw_i}) = m_v$ and $\#(T_{uw}) = m_u$. It follows that

$$n = \#(T_{uw}) + \#(T_{vu}) = \#(T_{uw}) + \#(T_{vx}) = m_u + m_v \geq 2m_u,$$

i.e., we have $m_u \leq n/2$. This proves that u is a centroid node. We leave the computation of the centroid node as an exercise (see Exercise 2.12). ∎

A similar notion is that of a *centroid edge*, i.e., an edge whose removal from the tree T gives two trees having approximately the same size. If the degree of T is bounded by a constant, then such a centroid edge always exists and can be computed in $O(n)$ time. The proof of this claim is left as an exercise (see Exercise 2.14).

Let T be a tree with n nodes. A *separator node* is a node in T whose removal results in two graphs G_1 and G_2 (that are forests), each having at most $2n/3$ nodes. In fact, a centroid node is also a separator node. Given this node, the two graphs G_1 and G_2 can be computed in $O(n)$ time. (The proof of this claim is left as an exercise; see Exercise 2.16.)

2.4 Coloring graphs of bounded degree

For the rest of this chapter, we will discuss several graph algorithms. We start with a simple graph coloring problem. In Chapter 11, we will need an algorithm that gives each vertex of a given graph a color such that any two vertices that are connected by an edge have different colors. In this section, we show that this is always possible using $D+1$ colors, if D is the maximum degree of any vertex.

Let G be an undirected graph with n vertices, let D be the maximum degree of any vertex of G, and let C be a set of $D+1$ colors. We present a greedy algorithm that colors the vertices of G using the colors of C, such that any two adjacent vertices have different colors.

Let $v_1, v_2, \ldots, v_n$ be the sequence of vertices of G. Our algorithm visits the vertices in this order. For any k with $1 \leq k \leq n$, if the algorithm has visited $v_1, v_2, \ldots, v_k$, then each of these vertices will have a color of C, such that any two adjacent vertices v_i and v_j, with $1 \leq i < j \leq k$, have different colors.

Initially, $k = 1$, and the algorithm colors v_1 with an arbitrary element of C. Let $1 \leq k < n$, and assume that $v_1, v_2, \ldots, v_k$ have been colored already.

The algorithm computes the set C_k of all colors $c \in C$ for which there is an index i, such that $i \leq k$, v_i has color c, and $\{v_i, v_{k+1}\}$ is an edge of G. Then the algorithm colors v_{k+1} with an arbitrary element of the set $C \setminus C_k$. Observe that $C \setminus C_k$ is non-empty, because C_k contains at most D elements.

It is clear that this algorithm computes a valid coloring of the vertices of G. Furthermore, the algorithm can be implemented so that its running time is $O(Dn)$. Hence, we have proved the following result:

Theorem 2.4.1 *Let G be an undirected graph with n vertices and let D be a positive integer. Assume that the degree of each vertex of G is less than or equal to D. The vertices of G can be colored using $D+1$ colors, such that any two adjacent vertices have different colors. Moreover, such a coloring can be computed in $O(Dn)$ time.*

2.5 Dijkstra's shortest paths algorithm

Let $G = (V, E)$ be a weighted undirected graph. Each edge $\{u, v\}$ in E has a weight which we denote by $wt(u, v)$. We assume that these weights are positive real numbers. Since the graph G is undirected, we have $wt(u, v) = wt(v, u)$ for any edge $\{u, v\}$. Recall that the weight of a path in G is defined as the sum of the weights of the edges on this path. For any two vertices u and v of V, we denote by $\delta(u, v)$ the minimum weight of any path in G between u and v. If there is no path between u and v, then $\delta(u, v) := \infty$. Observe that δ satisfies the *triangle inequality*: We have $\delta(u, v) \leq \delta(u, w) + \delta(w, v)$ for any three vertices u, v, and w of V.

In this section, we give an algorithm that solves the following problem:

Problem 2.5.1 Given a weighted undirected graph $G = (V, E)$ in which each edge has a positive real weight, given a vertex $s \in V$, and given a real number $R > 0$, compute all vertices $v \in V$ for which $\delta(s, v) \leq R$, and for each such vertex v, compute the value of $\delta(s, v)$.

If $R = \infty$, then the standard algorithm for solving this problem is *Dijkstra's algorithm*. The idea of this algorithm is as follows. For each vertex v, we maintain a variable $d(v)$, whose value is the smallest weight of any path between s and v encountered so far. Hence, during the algorithm, we always have $\delta(s, v) \leq d(v)$ for all vertices $v \in V$. We also maintain a subset A of V, such that for each $v \in A$, the value of $\delta(s, v)$ has been computed already.

At the start of the algorithm, we set $d(s) := 0$, $d(v) := \infty$ for all vertices $v \in V \setminus \{s\}$, and initialize A to be the empty set. Then we repeatedly select a vertex $u \in V \setminus A$ for which $d(u)$ is minimum. As we will see later, this vertex u has the property that $d(u) = \delta(s, u)$. Therefore, we add u to the set A and update the values $d(v)$ for all vertices $v \in V$ that are connected by an edge to u by assigning $d(v) := \min(d(v), d(u) + wt(u, v))$.

The vertices $v \in V \setminus A$ are maintained in a priority queue PQ with the priority being the value $d(v)$. This priority queue can be implemented, for example, using a binary heap or a Fibonacci heap. In this way, the minimum value $d(u)$ can be selected and deleted efficiently, and the values $d(v)$ that change can be updated efficiently.

It is clear that we can also run this algorithm if R is finite. The disadvantage of this approach is that the running time is at least linear in the number of vertices and edges of the graph G. Our goal is an algorithm whose running time only depends on the number of vertices v for which $\delta(s, v) \leq R$ and the degrees of these vertices.

2.5.1 Algorithm SingleSource

In our variant of Dijkstra's algorithm, the priority queue PQ only stores vertices v for which $d(v) \leq R$. For every vertex $v \in V$, the algorithm maintains two Boolean variables *added_to_PQ*(v) and *reported*(v). Initially, all these variables have the value *false*. The algorithm also maintains a set A which is implemented as a linked list. During the algorithm, the following two properties will be maintained for each vertex v:

1. *added_to_PQ*(v) = *true* if and only if v has ever been inserted into PQ.

2. *reported*(v) = *true* if and only if v is contained in the set A.

As shown later, the set A consists of all vertices v with $\delta(s, v) \leq R$ for which we have computed $\delta(s, v)$. The algorithm terminates as soon as PQ is empty or the current minimum value $d(u)$ in PQ is larger than R.

Algorithm SINGLESOURCE(G, s, R)

Comment: This algorithm takes as input an undirected graph G in which every edge has a positive weight, a vertex s of G, and a real number $R > 0$. It returns the set A of all vertices v for which $\delta(s, v) \leq R$.

Step 1: Initialize $d(s) := 0$, PQ as the priority queue containing s, $added\_to\_PQ(s) := true$, A as the empty list, and $reported(s) := false$.

Step 2: If PQ is empty, then go to Step 3. Otherwise, let u be a vertex of PQ for which $d(u)$ is minimum. If $d(u) > R$, then go to Step 3. Otherwise, $d(u) \leq R$, in which case the following Steps 2.1 and 2.2 are made:

Step 2.1: Add u to the list A, set $reported(u) := true$, and delete u from PQ.

Step 2.2: For each vertex v for which $\{u,v\}$ is an edge of G and $reported(v) = false$, distinguish the following two cases:

Case 2.2.1: If $added\_to\_PQ(v) = false$, then set $d(v) := d(u) + wt(u, v)$ and, if $d(v) \leq R$, insert v into PQ and set $added\_to\_PQ(v) := true$.

Case 2.2.2: If $added\_to\_PQ(v) = true$ and $d(u) + wt(u,v) < d(v)$, then set $d(v) := d(u) + wt(u, v)$ and update the priority of v in PQ.

After Steps 2.1 and 2.2 have been made, repeat Step 2.

Step 3: Reset all variables $added\_to\_PQ(v)$ that are *true* to *false*, and reset all variables $reported(v)$ with $v \in A$ to *false*.

Step 4: Return the list A.

Remark 2.5.2 In Chapter 15, algorithm SINGLESOURCE(G, s, R) will be run repeatedly on a fixed graph G, for different vertices s. Prior to one call to this algorithm, the values of all variables $added\_to\_PQ(v)$ and $reported(v)$ are assumed to be *false*. Because of this, all these variables are reset to *false* in Step 3 of the algorithm. There is no need to reset the values of the variables $d(v)$. Prior to one call to SINGLESOURCE, these variables can have arbitrary values. The proof of this last claim is left as an exercise (see Exercise 2.17).

In Section 2.5.2, we prove that the list A returned in Step 4 indeed consists of all vertices v for which $\delta(s, v) \leq R$. In Section 2.5.3, we analyze the running

time of the algorithm.

2.5.2 The correctness proof of algorithm SINGLESOURCE

We start by showing that for each vertex v stored in the priority queue PQ, the value of $d(v)$ is an upper bound on the weight of a shortest path between s and v.

Lemma 2.5.3 *Algorithm* SINGLESOURCE(G, s, R) *maintains the invariant that* $\delta(s, v) \leq d(v)$ *for all vertices* v *for which* $added\_to\_PQ(v) = true$.

Proof. The proof is by induction on the number of times Step 2 has been executed. Immediately after Step 1 (i.e., when Step 2 has not been executed yet), only the variable $added\_to\_PQ(s)$ has the value *true*. At this moment, the variable $d(s)$ has value zero, which is equal to $\delta(s, s)$. Hence, the invariant holds immediately before Step 2 is executed for the first time.

Consider one execution of Step 2, and assume that PQ is not empty at the start of this execution. Consider the vertex u in PQ for which $d(u)$ is minimum and that is chosen in this execution, and assume that $d(u) \leq R$.

Let v be any vertex for which $added\_to\_PQ(v) = true$ at the start of this execution of Step 2. If $d(v)$ does not change during this execution, then we clearly have $\delta(s, v) \leq d(v)$ afterwards. Assume that $d(v)$ does change. Then at the end of this execution, we have $d(v) = d(u) + wt(u, v)$. Since $d(u)$ does not change during this execution, we have $\delta(s, u) \leq d(u)$. By the triangle inequality, we have $\delta(s, v) \leq \delta(s, u) + wt(u, v)$. This implies that at the end of this execution of Step 2, we have $\delta(s, v) \leq d(v)$.

Finally, let v be any vertex for which the value of $added\_to\_PQ(v)$ is set to *true* during this execution of Step 2. Then, $d(v) = d(u) + wt(u, v)$. As in the previous case, we conclude that $\delta(s, v) \leq d(v)$ at the end of this execution. ■

Next we prove that after the value of a variable $d(v)$ becomes equal to $\delta(s, v)$, it does not change during the rest of the algorithm.

Lemma 2.5.4 *Let* v *be any vertex of* V. *Assume that at some moment during the algorithm,* $added\_to\_PQ(v) = true$ *and the value of* $d(v)$ *becomes equal to* $\delta(s, v)$. *Then the value of* $d(v)$ *does not change during the rest of the algorithm.*

Proof. Since *added_to_PQ*(v) = *true*, we have, by Lemma 2.5.3, $\delta(s,v) \leq d(v)$. It follows from the algorithm that, after *added_to_PQ*(v) has been set to *true*, the value of $d(v)$ does not increase. ∎

Lemma 2.5.5 *The minimum value $d(u)$ stored in the priority queue PQ does not decrease during the algorithm.*

Proof. Consider any execution of Step 2, and let u be the vertex that is inserted into A during this execution. Then $d(u)$ is minimum over all vertices that are stored in PQ. Let v be any vertex such that (i) v is already in PQ and $d(v)$ is decreased during this execution, or (ii) v is inserted into PQ during this execution. Then $d(v) = d(u) + wt(u,v)$, which is larger than $d(u)$. Hence, after deleting u from PQ, the new minimum stored in PQ is not smaller than $d(u)$. ∎

The next lemma is the basis for the correctness proof of algorithm SINGLESOURCE.

Lemma 2.5.6 *Let v be any vertex of V such that $v \neq s$ and $\delta(s,v) \leq R$, and let P be a shortest path between s and v. Let $\{u,v\}$ be the last edge on P. Assume that u is inserted into A during some execution of Step 2, and assume that $d(u) = \delta(s,u)$ at that moment. Then $d(v) = \delta(s,v)$ at any moment after this execution.*

Proof. During the execution of Step 2 in which u is inserted into A, the algorithm considers the edge $\{u,v\}$. We claim that v is not an element of A at that moment, i.e., *reported*(v) = *false*. To prove this, assume that *reported*(v) = *true* at the moment when u is inserted into A. It follows from the algorithm that *added_to_PQ*(v) = *true* at this moment. Also, by Lemma 2.5.5, we have $d(v) \leq d(u)$ at this moment. Combining this with Lemma 2.5.3 and the assumption of the lemma, it follows that

$$\delta(s,v) \leq d(v) \leq d(u) = \delta(s,u). \tag{2.1}$$

On the other hand, since P is a shortest path between s and v, the subpath of P between s and u is a shortest path between s and u. Therefore, $\delta(s,v) = \delta(s,u) + wt(u,v)$. Combining this with (2.1) implies that $wt(u,v) \leq 0$, which is a contradiction because $u \neq v$ and the edge weights are positive.

Hence, when the algorithm considers the edge $\{u,v\}$, we have *reported*(v) = *false*. At the end of the execution of Step 2 in which u is inserted into A, we

have $d(v) \leq d(u) + wt(u,v)$. Then the assumption of the lemma implies that $d(v) \leq \delta(s,u) + wt(u,v)$ at the end of this execution. We have seen already that $\delta(s,u) + wt(u,v) = \delta(s,v)$. Therefore, we have $d(v) \leq \delta(s,v)$ at the end of this execution. Then Lemmas 2.5.3 and 2.5.4 imply that $d(v) = \delta(s,v)$ at any moment after the execution in which u is inserted into A. ∎

The next lemma states that for each vertex u that is contained in the list A that is returned in Step 4, the algorithm has computed the correct value of $\delta(s,u)$.

Lemma 2.5.7 *Let A be the list that is returned by algorithm* SINGLESOURCE(G,s,R). *We have $d(u) = \delta(s,u)$ for each vertex u in this list.*

Proof. We claim that for each vertex u of the final list A, we have $d(u) = \delta(s,u)$ at the moment when u is inserted into this list. Since $d(u)$ does not change afterwards, this will prove the lemma.

The proof of the claim is by induction. The vertex s is the first vertex that is inserted into A. For this vertex, the claim clearly holds.

Let u be any vertex of the final list A such that $u \neq s$. Consider the execution of Step 2 in which u is inserted into A. We denote the list A at the beginning of this execution by A'. The inductive hypothesis is that the claim holds for any vertex $x \in A'$. We will show that the claim then also holds for u.

Let P be a shortest path between s and u. (Since $\delta(s,u) \leq d(u) \leq R$, the path P exists.) Observe that $s \in A'$ and $u \notin A'$. Let y be the first[1] vertex on P that is not contained in A' and let x be the predecessor of y. Then $x \in A'$. By the induction hypothesis, we had $d(x) = \delta(s,x)$ at the moment when x was inserted into A. Hence, by Lemma 2.5.6, $d(y) = \delta(s,y)$ at any moment after the execution of Step 2 in which x is inserted into A.

We claim that $y = u$. If this is true, then $d(u) = \delta(s,u)$ at the moment when u is inserted into A, and the proof is complete.

Assume that $y \neq u$. Then $\delta(s,y) < \delta(s,u)$. Hence, at the moment when the algorithm inserts u into A, we have $d(y) < \delta(s,u)$. Vertex u is inserted into A because, at that moment, $d(u)$ is minimum in the priority queue. That is, when u is inserted into A, we have $d(u) \leq d(y)$. Hence, we have $d(u) < \delta(s,u)$ at that moment. This is a contradiction to Lemma 2.5.3 and, therefore, we have shown that $y = u$. ∎

[1] when following the path P from s to u

The previous lemma implies that algorithm SingleSource(G, s, R) computes a subset of the set of all vertices u for which $\delta(s, u) \leq R$. It remains to show that the algorithm computes all such vertices u.

Lemma 2.5.8 *Let A be the list that is returned by algorithm* SingleSource(G, s, R). *The list A contains all vertices u of V for which $\delta(s, u) \leq R$.*

Proof. By Lemma 2.5.5, the minimum value of $d(u)$ stored in the priority queue does not decrease over time. This implies that the list A stores the vertices u in non-decreasing order of the value $d(u)$. By Lemma 2.5.7, we have $d(u) = \delta(s, u)$ for each $u \in A$. Therefore, the algorithm can terminate as soon as the minimum value $d(u)$ in the priority queue is larger than R. ■

2.5.3 The running time of algorithm SingleSource

Let n denote the number of vertices of G, and consider the list A that is returned by the algorithm. The running time is dominated by the total time needed to process the following priority queue operations:

1. DeleteMin: This operation finds and deletes the vertex u in PQ for which $d(u)$ is minimum. It is performed at most $|A| + 1$ times.
2. Insert(v): This operation inserts the vertex v into PQ, with priority $d(v)$. Since the algorithm only inserts v in case $d(v) \leq R$, this operation is performed $|A|$ times.
3. DecreaseKey$(v, d'(v))$: This operation replaces the priority $d(v)$ of vertex v by the smaller priority $d'(v)$. It is performed at most $|A| + \sum_{v \in A} deg(v)$ times.

We denote the number of vertices stored in the priority queue by $|PQ|$. If we implement PQ using a Fibonacci heap, then one operation DeleteMin takes $O(\log |PQ|)$ amortized time, one operation Insert takes $O(1)$ amortized time, and one operation DecreaseKey takes $O(1)$ amortized time. Since $|PQ| \leq |A|$ at any moment during the algorithm, we have proved the following result:

Theorem 2.5.9 *Let G be a weighted undirected graph with n vertices, whose edges have positive weights; let s be any vertex of G, and let $R > 0$ be any real number. Algorithm* SingleSource(G, s, R) *computes*

1. *the set A of all vertices v of G for which $\delta(s, v) \leq R$, and*
2. *for each vertex $v \in A$ the value of $\delta(s, v)$,*

in $O\left(|A| \log |A| + \sum_{v \in A} deg(v)\right)$ time.

If we run algorithm SingleSource(G, s, R) with $R = \infty$, then we get for each vertex v the weight $\delta(s, v)$ of a shortest path between s and v. In this case, we have the classical Dijkstra algorithm, which solves the single-source shortest paths problem. Since $|A| \leq n$, and $\sum_{v \in A} deg(v)$ is equal to twice the number of edges of G, we get the following result:

Corollary 2.5.10 *Let G be a weighted undirected graph with n vertices and m edges, and assume that these edges have positive weights. Let s be any vertex of G. Algorithm* SingleSource(G, s, ∞) *computes for each vertex v of G the value $\delta(s, v)$ in total time $O(n \log n + m)$.*

2.6 Minimum spanning trees

Let $G = (V, E)$ be a weighted undirected connected graph. Each edge e in E has a weight $wt(e)$, which is a positive real number. The weight of any subgraph $G' = (V, E')$ of G is defined as the sum of the weights of the edges of E'.

In this section, we consider the problem of computing a connected subgraph G' of G having minimum weight. It is not difficult to see that G' must be a tree. Therefore, G' is called a *minimum spanning tree* of G.

We will present two well-known algorithms for computing a minimum spanning tree. Both use a *greedy* strategy to build the tree, and their correctness proofs are based on the following lemma, whose proof is left as an exercise (see Exercise 2.18). We say that two sets A and B of vertices form a partition of the vertex set V if $A \cup B = V$, $A \cap B = \emptyset$, $A \neq \emptyset$, and $B \neq \emptyset$.

Lemma 2.6.1 *Let A and B be two sets of vertices that form a partition of V. Let a and b be vertices of A and B, respectively, such that $\{a, b\}$ is an edge of E and the weight $wt(a, b)$ is minimum. Then the graph G contains a minimum spanning tree in which $\{a, b\}$ is an edge.*

2.6.1 Kruskal's algorithm

Our first minimum spanning tree algorithm is known as Kruskal's algorithm. Let the graph $G = (V, E)$ have n vertices and m edges. Kruskal's algorithm maintains a *forest*, which is a collection of trees. It repeatedly adds an edge of minimum weight that does not create a cycle. To be more precise, the algorithm starts with a forest consisting of n trees, each consisting of a single vertex of V. Then the algorithm combines two trees in the forest, using an edge of minimum weight, and repeats this, until the forest consists of one single tree. This final tree is a minimum spanning tree of the graph G.

The algorithm, which we denote by KRUSKAL(G), is given below. During this algorithm, every tree in the forest has a unique vertex that is chosen as a "leader". When two trees of the forest are combined, one of the two leaders of the two trees is chosen as the leader of the combined tree.

Algorithm KRUSKAL(G)

Comment: This algorithm takes as input a connected weighted undirected graph $G = (V, E)$. It returns the edge set of a minimum spanning tree of G.

```
sort the edges of E in non-decreasing order of their weights;
let {u_k, v_k}, 1 ≤ k ≤ |E|, be the sorted sequence of edges;
for each u ∈ V
do V_u := {u};
   E_u := ∅
endfor;
for k := 1 to |E|
do (* test if the edge {u_k, v_k} has to be included *)
   x := index such that u_k ∈ V_x;
   y := index such that v_k ∈ V_y;
   if x ≠ y
   then (* adding edge {u_k, v_k} does not introduce a cycle *)
        V_x := V_x ∪ V_y;
        V_y := ∅;
        E_x := E_x ∪ E_y ∪ {{u_k, v_k}};
        E_y := ∅
   endif
endfor;
x := index such that V_x ≠ ∅;
return E_x
```

We claim that after all edges $\{u_k, v_k\}$ have been tested for inclusion, there is exactly one index x for which $V_x \neq \emptyset$. The corresponding set E_x is the edge set of a minimum spanning tree of the graph G. The correctness proof is based on the following claims, whose proofs are left as an exercise (see Exercise 2.19):

1. At any moment after the edge $\{u_k, v_k\}$ has been tested for inclusion, the vertices u_k and v_k are connected by a path in the graph (V, E'), where $E' := \bigcup_{x \in V} E_x$.
2. During the second for-loop, the following invariant is maintained:

(a) The non-empty sets V_x form a partition of the vertex set V.

(b) There is a minimum spanning tree of G that contains the edge set $E' := \bigcup_{x \in V} E_x$.

(c) For any two indices x and y, such that $x \neq y$ and V_x and V_y are both non-empty, no vertex of V_x is connected by a path in the graph (V, E') to any vertex of V_y.

To implement Kruskal's algorithm, we have to maintain the non-empty vertex sets V_x under the following three operations:

1. MakeSet(u): Given a vertex u, this operation initializes a new set $V_u := \{u\}$. It has to be processed for each vertex u of V.

2. Find(u): This operation finds the index x such that vertex u is an element of the set V_x. It has to be processed $2m$ times.

3. Union(V_x, V_y): This operation assigns $V_x := V_x \cup V_y$ and $V_y := \emptyset$. It has to be processed $n - 1$ times.

The problem of designing a data structure that supports these three operations is called the *union-find* problem. In Exercise 2.20, you are asked to design a data structure in which each MakeSet and Union operation takes $O(1)$ time and each Find operation takes $O(\log n)$ time. Therefore, we obtain the following result:

Theorem 2.6.2 *Given a connected weighted undirected graph G with n vertices and m edges, algorithm* Kruskal(G) *computes a minimum spanning tree of G in $O(m \log n)$ time.*

2.6.2 Prim's algorithm

The second algorithm for computing a minimum spanning tree is known as Prim's algorithm. As before, we denote the number of vertices and edges of the graph $G = (V, E)$ by n and m, respectively. Prim's algorithm starts with a set A containing an arbitrary vertex of G, and an empty set E' of edges. Then, it repeatedly adds to the edge set E' an edge of minimum weight between a vertex of A and a vertex of $V \setminus A$. If $\{u, v\}$ is such an edge, with $u \in A$, then the vertex v "moves" to the set A. The algorithm terminates as

soon as $A = V$. At that moment, the graph (V, E') is a minimum spanning tree of G.

Algorithm Prim(G)

Comment: This algorithm takes as input a connected weighted undirected graph $G = (V, E)$. It returns the edge set of a minimum spanning tree of G.

$r :=$ arbitrary vertex of G;
$A := \{r\}$;
$E' := \emptyset$;
while $A \neq V$
do find a vertex $u \in A$ and a vertex $v \in V \setminus A$ such that
  $\{u, v\} \in E$ and $wt(u, v)$ is minimum;
  $A := A \cup \{v\}$;
  $E' := E' \cup \{\{u, v\}\}$
endwhile;
return E'

The correctness proof of this minimum spanning tree algorithm is left as an exercise (see Exercise 2.21). If we implement the algorithm using a Fibonacci heap, then we get the following result:

Theorem 2.6.3 *Given a connected weighted undirected graph G with n vertices and m edges, algorithm* Prim(G) *computes a minimum spanning tree of G in $O(n \log n + m)$ time.*

Exercises

2.1 Prove that the subgraph P_u in Section 2.3.2 is a path.

2.2 In Section 2.3.2, we defined the notion of a *group parent*. Prove that if two distinct nodes u and v have the same group parent, then either u is an ancestor of v, or v is an ancestor of u.

2.3 Prove the correctness of algorithm LCA in Section 2.3.2.

2.4 Prove Observation 2.3.1.

2.5 Algorithm GroupParents(u, x) in Section 2.3.2 is based on a preorder traversal of the tree T. Design an algorithm that is based on a postorder traversal and that computes the group parents of all nodes of T.

2.6 Let T be a rooted binary tree with n nodes. Assume that each internal node of T stores pointers to its left and right children, but no node stores a pointer to its parent. Design an $O(n)$–time algorithm that stores with each node of T (except for the root) a pointer to its parent.

2.7 We have proved Theorem 2.3.2 for binary trees. Extend the proof to arbitrary rooted trees.

2.8 Let $T = (V, E_V)$ be a tree with n nodes, and let $G = (V, E)$ be an undirected graph that contains T, i.e., $E_V \subseteq E$. Let u and v be any two distinct nodes of V, and let $P = (u = x_0, x_1, x_2, \ldots, x_\ell = v)$ be the unique path in T between u and v. A path $Q = (u = x_{i_0}, x_{i_1}, x_{i_2}, \ldots, x_{i_k} = v)$ in the graph G between u and v is called a *T-monotone path*, if $0 = i_0 < i_1 < i_2 < \ldots < i_k = \ell$. The *$T$-monotone diameter* of G is defined as the smallest integer k, such that for any two distinct nodes u and v of V, there is a T-monotone path in G between u and v that contains at most k edges.

Design an $O(n)$–time algorithm that computes, when given any tree T with n nodes as input, a graph having $O(n)$ edges and whose T-monotone diameter is $O(\log n)$. (*Hint:* Use the paths P_u that were defined in Section 2.3.2.)

2.9 Prove Theorems 2.3.6 and 2.3.8.

2.10 Let T be a rooted tree with n nodes. For any two distinct nodes u and v of T, let w be their lowest common ancestor, and let w_u and w_v be the children of w that contain u and v in their subtrees, respectively. Extend the results of Section 2.3.3 such that, when given u and v, not only their lowest common ancestor w, but also w_u and w_v are returned.

2.11 We have proved Theorems 2.3.7 and 2.3.8 for arrays that satisfy the (± 1)-property. Prove that these theorems hold for arbitrary arrays. (*Hint:* Reduce the range minimum problem for an arbitrary array to the lowest common ancestor problem.)

2.12 Design an algorithm that computes, in $O(n)$ time, a centroid node of any given tree with n nodes.

2.13 Can a tree have more than one centroid node? Can it have more than two centroid nodes?

2.14 Let n and D be integers such that $n \geq 2$ and $2 \leq D < n$. Let T be a tree with n nodes in which each node has degree at most D. Prove that T contains a *centroid edge*, i.e., an edge e, such that removing e from T gives two trees, each of which contains at least $(n-1)/D$, and at most $((D-1)n+1)/D$ nodes. Design an $O(n)$–time algorithm that computes such a centroid edge.

2.15 Let P be a simple polygon with n vertices. A *diagonal* is a line segment joining two non-adjacent vertices of P that is completely contained in P. Such a diagonal cuts P into two disjoint simple polygons P_1 and P_2. Prove that there is a diagonal such that both P_1 and P_2 have at least $1 + n/3$ and at most $1 + 2n/3$ vertices. (*Hint:* The dual of any triangulation of P is a tree.)

2.16 Let $n \geq 2$ and let T be a tree with n vertices. Prove that G contains a separator node v, i.e., a node whose removal gives two graphs G_1 and G_2—that are not necessarily trees—each having at most $2n/3$ vertices. Moreover, design an algorithm that computes such a vertex v and the corresponding graphs G_1 and G_2 in $O(n)$ time.

2.17 Consider algorithm SingleSource(G, s, R) of Section 2.5.1. Assume that, prior to running this algorithm, the variable $d(v)$ of each vertex v of G has some arbitrary value. Prove that the algorithm still correctly solves Problem 2.5.1.

2.18 Prove Lemma 2.6.1.

2.19 Use Lemma 2.6.1 to prove the correctness of Kruskal's minimum spanning tree algorithm.

2.20 Design a data structure for the union-find problem on a set of n elements (see Section 2.6.1), in which each MakeSet and Union operation takes $O(1)$ time, and each Find operation takes $O(\log n)$ time.

2.21 Use Lemma 2.6.1 to prove the correctness of Prim's minimum spanning tree algorithm.

2.22 Prove that the vertices of any planar graph can be colored using six colors, such that no two adjacent vertices have the same color. (*Hint:* First prove that in any planar graph, there is a vertex whose degree is less than or equal to five.)

Bibliographic notes

Good and thorough introductions to algorithms can be found in the books Manber [1989] and Cormen et al. [2001]. Much deeper treatments of graph theory can be found in the books Bollobás [1998], Diestel [2000], Even [1979], and Harary [1972].

The technique that we used in Section 2.3.2 to partition a tree T into pairwise disjoint paths is due to Cole and Vishkin [1988].

Harel and Tarjan were the first to show that lowest common ancestor queries in a tree with n nodes can be answered in $O(1)$ time (using indirect-addressing) after an $O(n)$–time preprocessing, see Harel and Tarjan [1984]. They also proved an $\Omega(\log\log n)$ lower bound for pointer machine algorithms. The algorithm of Harel and Tarjan was simplified in Schieber and Vishkin [1988] (see also Chapter 8 of the book Gusfield [1997]). The simple algorithm given in Section 2.3.3 is from Bender and Farach-Colton [2000]; this paper also contains a solution to Exercise 2.11.

Jordan [1869] was the first to show that every tree contains a centroid node; see Theorem 2.3.9. The proof given in Section 2.3.4 is from Bodlaender, Tel, and Santoro [1994]. The existence of a separator node in a tree is a special case of Lipton and Tarjan's separator theorem for planar graphs, see Lipton and Tarjan [1979, 1980].

Dijkstra's single-source shortest paths algorithm appeared in Dijkstra [1959]. Fibonacci heaps are due to Fredman and Tarjan [1987]. Kruskal's minimum spanning tree algorithm appeared in Kruskal [1956], whereas Prim's algorithm was discovered independently by Jarník [1930], Prim [1957], and Dijkstra [1959]. Good descriptions of the algorithms of Dijkstra, Kruskal, and Prim, as well as Fibonacci heaps and data structures for the union-find problem can be found in the book Cormen et al. [2001]. The currently

fastest known algorithm for computing a minimum spanning tree appears in Chazelle [2000b].

The *Euclidean minimum spanning tree* $MST(S)$ of a set S of n points in $\mathbb{R}^d$ is a minimum spanning tree of the complete Euclidean graph whose vertex set is S. Prim's algorithm computes this tree in $O(n^2)$ time. By exploiting geometry, we can do better. Let us assume that the dimension d is equal to two. It is known that $MST(S)$ is a subgraph of the Delaunay triangulation $DT(S)$ of S. That is, the minimum spanning tree of $DT(S)$ is a Euclidean minimum spanning tree of the point set S. Since $DT(S)$ is a planar graph with n vertices, it has at most $3n-6$ edges. Therefore, given $DT(S)$, we can use either Kruskal's or Prim's algorithm to compute $MST(S)$ in $O(n\log n)$ time. In fact, Cheriton and Tarjan [1976] have shown that $MST(S)$ can be computed in $O(n)$ time from $DT(S)$. Several algorithms are know for computing $DT(S)$ in $O(n\log n)$ time (see Shamos and Hoey [1975], Lee and Schachter [1980], or any of the books on computational geometry mentioned in the bibliographic notes at the end of Chapter 1).

In dimensions $d \geq 3$, the Euclidean minimum spanning tree problem becomes considerably harder. For example, if $d = 3$, a Euclidean minimum spanning tree can be computed in a time that is roughly proportional to $n^{4/3}$ (see Agarwal et al. [1991] and Callahan and Kosaraju [1993]). In Chapter 10, we will show how spanners can be used to compute an approximate Euclidean minimum spanning tree of any set of n points in $\mathbb{R}^d$ in $O(n\log n)$ time.

Chapter 3

The algebraic computation-tree model

> My intelligence is bewildered by Your equivocal instructions. Therefore, please tell me decisively which will be most beneficial for me.
>
> — *The Bhagavad Gita.*

In this chapter, we formalize the model of computation that will be used throughout most of this book. This model, called the *algebraic computation-tree model* captures algorithms that perform exact arithmetic on arbitrary real numbers, and that use the following primitive operations: comparison of two real numbers, and the arithmetic operations of addition ($+$), subtraction ($-$), multiplication ($\times$), division ($/$), and the square root. Algorithms in this model take sequences of real numbers as input. The worst-case running time of such an algorithm depends only on the *number* of elements in the input sequence. Hence, it does *not* depend on the number of bits needed to specify the input.

We start in Section 3.1 with a formal definition of computation problems, and algebraic computation-trees as devices for solving such problems. In Section 3.2, we introduce a restricted type of algebraic computation-trees, called *algebraic decision-trees*, that solve decision problems (i.e., their output is either YES or NO). In Section 3.3, we present a general technique for

proving lower bounds on the time complexity for solving decision problems. Since a given computation problem often "contains" a related decision problem, this also gives a general approach for proving lower bounds on the time complexity for solving computation problems. In Section 3.4, we apply this approach and prove that the time complexity of computing a t-spanner of a set of n points in $\mathbb{R}^d$ is $\Omega(n \log n)$.

3.1 Algebraic computation-trees

Informally, an algorithm is a procedure that transforms an input consisting of n real numbers (i.e., a point in $\mathbb{R}^n$) to an output (for example, a real number or a combinatorial structure) belonging to some *solution space* $\mathcal{S}$. In other words, an algorithm solves a *computation problem*, which is a function

$$\mathcal{P} : \mathbb{R}^n \to \mathcal{S}.$$

An example of a computation problem is the *sorting problem*, in which $\mathcal{S}$ is the set of all non-decreasing sequences of n real numbers.

Definition 3.1.1 Let n be a positive integer, and let $\mathcal{S}$ be a solution space. An *algebraic computation-tree* on a sequence $s_1, s_2, \ldots, s_n$ of n variables is a finite tree T in which each node has at most two children, and that satisfies the following three conditions:

1. Each leaf is labeled with the combinatorial description, in terms of the variables $s_1, s_2, \ldots, s_n$, of an element in $\mathcal{S}$.
2. Each node u having one child is labeled with a variable $Z(u)$ and an assignment of the form
   (a) $Z(u) := A_1 \& A_2$, where $\& \in \{+, -, *, /\}$, or
   (b) $Z(u) := \sqrt{A_1}$,

   where, for $i = 1, 2$, (i) $A_i = Z(u')$ for some proper ancestor u' of u in T, (ii) $A_i \in \{s_1, s_2, \ldots, s_n\}$, or (iii) A_i is a real number constant.
3. Each node u having two children is labeled with a comparison of the form $A \bowtie 0$, where (i) $A = Z(u')$ for some proper ancestor u' of u in T, or (ii) $A \in \{s_1, s_2, \ldots, s_n\}$. The two outgoing edges leading to the left and right children of u are labeled with $\leq$ and $>$, respectively.

An algebraic computation-tree T corresponds to an algorithm $\mathcal{A}_T$ that solves a computation problem $\mathcal{P} : \mathbb{R}^n \to \mathcal{S}$. Given an input sequence $s_1, s_2, \ldots, s_n$ of n real numbers, algorithm $\mathcal{A}_T$ traverses a path down the tree T starting at the root. If the current node u has one child, then $\mathcal{A}_T$ performs the corresponding arithmetic operation, assigns the result to the variable $Z(u)$, and proceeds to the child of u. If the current node u has two children, then the corresponding comparison is made and, depending on the outcome, $\mathcal{A}_T$ proceeds to the left or right child of u. When $\mathcal{A}_T$ reaches a leaf, say w, it takes the label stored at w, replaces all variables in this label by the values of the real numbers $s_1, s_2, \ldots, s_n$, returns the resulting label and terminates. The returned label is the solution to the problem $\mathcal{P}$ on input sequence $s_1, s_2, \ldots, s_n$, i.e., the value of $\mathcal{P}(s_1, s_2, \ldots, s_n)$.

We require that no computation leads to a division by zero or taking the square root of a negative number. Hence, for any sequence of n real numbers, algorithm $\mathcal{A}_T$ is well-defined. We also require that for each leaf w of T, there exists an input sequence of length n, on which algorithm $\mathcal{A}_T$ terminates in w.

Thus, a specific algebraic computation-tree can only represent an algorithm on all inputs of a specific length. In this sense, the algebraic computation-tree represents all possible behaviors of an algorithm over all input sequences consisting of n real numbers. We remark that an algebraic computation-tree is only used to *analyze* an algorithm; the tree is *not* explicitly constructed. As we will see, algebraic computation-trees are a convenient tool to prove lower bounds for computation problems.

An algorithm $\mathcal{A}$ that only makes comparisons and the arithmetic operations $+, -, *, /$, and $\sqrt{}$, is called an *algebraic computation-tree algorithm*, if there is an algebraic computation-tree T such that $\mathcal{A} = \mathcal{A}_T$. In Exercise 3.6, you will be asked to prove that *not* every such algorithm $\mathcal{A}$ is an algebraic computation-tree algorithm.

Definition 3.1.2 Let $\mathcal{P} : \mathbb{R}^n \to \mathcal{S}$ be a computation problem. We say that $\mathcal{P}$ is *solvable in the algebraic computation-tree model*, if there exists an algebraic computation-tree T such that, for any $(s_1, s_2, \ldots, s_n) \in \mathbb{R}^n$, the corresponding algorithm $\mathcal{A}_T$ returns the value of $\mathcal{P}(s_1, s_2, \ldots, s_n)$. We say then that T *solves* the computation problem $\mathcal{P}$.

Intuitively, the worst-case running time of an algebraic computation-tree algorithm $\mathcal{A}_T$ is the maximum number of comparisons and arithmetic operations that are made on any input sequence of length n. Hence, each

comparison and each of the elementary operations $+, -, *, /$, and $\sqrt{}$ takes unit time. The following definition expresses this in terms of the tree T.

Definition 3.1.3 Let T be an algebraic computation-tree. The *time complexity* of the corresponding algorithm $\mathcal{A}_T$ is defined as the height of the tree T.

Having defined the time complexity of an algorithm, we can define the time complexity of a computation problem:

Definition 3.1.4 Let $\mathcal{P} : \mathbb{R}^n \to \mathcal{S}$ be a computation problem that is solvable in the algebraic computation-tree model. The *time complexity* of $\mathcal{P}$ is defined as the minimum height of any algebraic computation-tree that solves $\mathcal{P}$.

3.2 Algebraic decision-trees

A computation problem $\mathcal{P} : \mathbb{R}^n \to \mathcal{S}$ is called a *decision problem*, if $\mathcal{S} = \{\text{YES,NO}\}$. An example is the *element uniqueness problem*, in which we are given a sequence $s_1, s_2, \ldots, s_n$ of n real numbers and have to decide if these elements are pairwise distinct. In this case, we have

$$\mathcal{P}(s_1, s_2, \ldots, s_n) = \begin{cases} \text{YES} & \text{if } s_i \neq s_j \text{ for all } i \neq j, \\ \text{NO} & \text{otherwise.} \end{cases}$$

Throughout this book, we will mainly consider computation problems. As we will see, however, lower bounds for decision problems are generally easier to prove than lower bounds for computation problems. Fortunately, it is often the case that a computation problem implicitly "contains" a related decision problem. For example, any algorithm that computes the minimum distance in a set of n points also solves the element uniqueness problem. As a result, any lower bound for the latter decision problem immediately implies a lower bound for the corresponding computation problem.

An algebraic computation-tree that solves a decision problem is called a *algebraic decision-tree*. This restricted type of decision tree is formally defined by replacing Condition 1. in Definition 3.1.1 by the following condition:

- Each leaf is labeled with either YES or NO.

Let $\mathcal{P} : \mathbb{R}^n \to \{\text{YES,NO}\}$ be a decision problem. A point $(s_1, s_2, \ldots, s_n)$ in $\mathbb{R}^n$ is called a *YES-instance* for $\mathcal{P}$, if the value of $\mathcal{P}(s_1, s_2, \ldots, s_n)$ is YES. We associate with $\mathcal{P}$ the subset $V_{\mathcal{P}}$ of $\mathbb{R}^n$ consisting of all YES-instances. Conversely, for any subset V of $\mathbb{R}^n$, there is a decision problem $\mathcal{P}$ for which $V_{\mathcal{P}} = V$. Hence, we can identify decision problems with subsets of $\mathbb{R}^n$. As an example, if $\mathcal{P}$ is the element uniqueness problem, then we have

$$V_{\mathcal{P}} = \left\{ (s_1, s_2, \ldots, s_n) \in \mathbb{R}^n : \prod_{1 \le i < j \le n} (s_i - s_j) \neq 0 \right\}.$$

A decision problem $\mathcal{P} : \mathbb{R}^n \to \{\text{YES,NO}\}$ is called *decidable in the algebraic decision-tree model*, if there exists an algebraic decision-tree T such that, for any $(s_1, s_2, \ldots, s_n) \in \mathbb{R}^n$, the corresponding algorithm $\mathcal{A}_T$ returns YES if $(s_1, s_2, \ldots, s_n) \in V_{\mathcal{P}}$, and NO if $(s_1, s_2, \ldots, s_n) \notin V_{\mathcal{P}}$. We say then that T *decides* the decision problem $\mathcal{P}$.

Definition 3.2.1 Let V be a subset of $\mathbb{R}^n$, and let $\mathcal{P}$ be the corresponding decision problem. If $\mathcal{P}$ is decidable in the algebraic decision-tree model, then we define the *time complexity* of V as the minimum height of any algebraic decision-tree that decides $\mathcal{P}$.

3.3 Lower bounds for algebraic decision-tree algorithms

In this section, we will show that the topological structure of a set $V \subseteq \mathbb{R}^n$ yields a lower bound on the time complexity of V.

Basic Idea: A lower bound on the time complexity of a decision problem $\mathcal{P}$ can be obtained by inspecting the topological structure of $V_{\mathcal{P}}$. If the number of connected components of $V_{\mathcal{P}}$ is $CC(V_{\mathcal{P}})$, then the following is true: In the linear decision-tree model, its time complexity is lower bounded by $\log(CC(V_{\mathcal{P}}))$; in the algebraic decision-tree model, its time complexity is lower bounded by $\frac{\log(CC(V_{\mathcal{P}})) - n\log 3}{1+2\log 3}$.

3.3.1 Linear decision-trees

In order to introduce the basic approach, we first consider a restricted class of algorithms. We consider algorithms that can only perform the following unit-time operations: First, quantities that can be added or substracted are either constants, input elements, or values that have been previously computed. Second, any input element and any previously computed value can be multiplied by a constant. Finally, any input element and any previously computed value can be compared with zero with an outcome of either "$\le$" or "$>$".

This restricted class of algorithms is formally defined by replacing Condition 2. in Definition 3.1.1 by the following condition:

- Each node u having one child is labeled with a variable $Z(u)$ and an assignment $Z(u) := A_1 \& A_2$, having one of the following two forms:
  1. $\& \in \{+, -\}$ and, for $i = 1, 2$, (i) $A_i = Z(u')$ for some proper ancestor u' of u in T, (ii) $A_i \in \{s_1, s_2, \ldots, s_n\}$, or (iii) A_i is a real constant.
  2. $\& \in \{*, /\}$ and
     (a) $A_1 = Z(u')$ for some proper ancestor u' of u in T, or $A_1 \in \{s_1, s_2, \ldots, s_n\}$, or A_1 is a real constant, whereas
     (b) A_2 is a real constant.

Observe that two input elements (or previously computed values) cannot be multiplied or divided, thus avoiding non-linear functions of the input elements. Hence, each variable that occurs during the execution of the corresponding algorithm is a linear function of the input variables, such as $4 + 3s_1 + s_2/7 - 8s_3 + 2s_4 - s_5$. Therefore, we call these algorithms *linear decision-tree algorithms*.

Consider a decision problem $\mathcal{P}$ that is decidable in this linear model, and let $V := V_{\mathcal{P}} \subseteq \mathbb{R}^n$ be the corresponding set of YES-instances. Let T be an arbitrary linear decision-tree that decides $\mathcal{P}$. Hence, the algorithm $\mathcal{A}_T$ corresponding to T returns YES if and only if it gets an element of V as input.

For any leaf w of T, we denote by $R(w)$ the set of those inputs on which algorithm $\mathcal{A}_T$ terminates in leaf w. To give a formal definition of this set, consider the path $u_1, u_2, \ldots, u_\ell$ from the root u_1 to $u_\ell = w$. Let $v_1, v_2, \ldots, v_k$

be all nodes on this path that have two children. For each i with $1 \leq i \leq k$, algorithm $\mathcal{A}_T$ makes a comparison in node v_i, which can be written as

$$F_i(s_1, s_2, \ldots, s_n) \bowtie 0,$$

where F_i is a linear function of the n input variables $s_1, s_2, \ldots, s_n$. Then $R(w)$ is the set consisting of all points $(s_1, s_2, \ldots, s_n) \in \mathbb{R}^n$, such that for all i with $1 \leq i \leq k$,

1. $F_i(s_1, s_2, \ldots, s_n) \leq 0$ if the path in T to w proceeds from v_i to its left child, and
2. $F_i(s_1, s_2, \ldots, s_n) > 0$ if the path in T to w proceeds from v_i to its right child.

The following lemma states the main property of the sets $R(w)$ that we will use to prove a lower bound on the time complexity of the decision problem $\mathcal{P}$.

Lemma 3.3.1 *The set $R(w)$ is connected, i.e., for any two points p and q in $R(w)$, there is a continuous curve in $\mathbb{R}^n$ between p and q that is completely contained in $R(w)$.*

Proof. The inequality at node v_i, given by $F_i(s_1, s_2, \ldots, s_n) \bowtie 0$, defines a closed or open halfspace in $\mathbb{R}^n$. The set $R(w)$ is the intersection of the halfspaces defined by the inequalities at nodes $v_1, \ldots, v_k$. Hence, since each halfspace is convex, $R(w)$ is also convex and, therefore, connected. ∎

The set V consists of one or more *connected components*. Let A and B be two distinct connected components of V, and let $p = (s_1, s_2, \ldots, s_n)$ and $q = (t_1, t_2, \ldots, t_n)$ be points in A and B, respectively. Let w_p be the leaf of T in which algorithm $\mathcal{A}_T$ terminates on input p. Define w_q similarly with respect to input q.

Lemma 3.3.2 *The leaves w_p and w_q are distinct.*

Proof. Assume that $w_p = w_q$, and denote this leaf by w. Observe that p and q are both contained in the set $R(w)$. We saw in the proof of Lemma 3.3.1 that $R(w)$ is convex. Therefore, the line segment pq is completely contained in $R(w)$. Hence, for each point $x = (x_1, x_2, \ldots, x_n)$ that is on pq, algorithm

$\mathcal{A}_T$, when given x as input, terminates in w. Since $p \in V$, leaf w is labeled with YES. This proves that each point x on pq belongs to the set V. As a result, the points p and q are connected by a continuous curve (*viz.* the line segment pq) that is completely inside V. This is a contradiction because these points are in different connected components of V. ∎

Lemma 3.3.2 immediately implies the following lower bound. For any subset V of $\mathbb{R}^n$, we denote the number of its connected components by $CC(V)$.

Theorem 3.3.3 *Let $\mathcal{P}$ be a decision problem that is decidable in the linear decision-tree model and let $V_{\mathcal{P}} \subseteq \mathbb{R}^n$ be the corresponding set of YES-instances. The time complexity of $V_{\mathcal{P}}$ in the linear decision-tree model is greater than or equal to $\log(CC(V_{\mathcal{P}}))$.*

Proof. Let T be an arbitrary linear decision-tree that decides $V_{\mathcal{P}}$. By Lemma 3.3.2, T has at least $CC(V_{\mathcal{P}})$ leaves. Hence, the height of this tree is greater than or equal to $\log(CC(V_{\mathcal{P}}))$. ∎

Let us apply this result to the element uniqueness problem, in which case we have

$$V_{\mathcal{P}} = \left\{ (s_1, s_2, \ldots, s_n) \in \mathbb{R}^n : \prod_{1 \leq i < j \leq n} (s_i - s_j) \neq 0 \right\}.$$

In order to get a lower bound on this problem for linear decision-tree algorithms, we need to estimate the number of connected components of $V_{\mathcal{P}}$.

Let π and ρ be two distinct permutations of $1, 2, \ldots, n$, and consider the points $p := (\pi(1), \pi(2), \ldots, \pi(n))$ and $r := (\rho(1), \rho(2), \ldots, \rho(n))$ in $\mathbb{R}^n$. Clearly, both p and r belong to the set $V_{\mathcal{P}}$. We will show that these two points belong to different connected components of $V_{\mathcal{P}}$. This will prove that $CC(V_{\mathcal{P}}) \geq n!$.

Because π and ρ are distinct, there are indices i and j with $1 \leq i \leq n$ and $1 \leq j \leq n$, such that $\pi(i) < \pi(j)$ and $\rho(i) > \rho(j)$. Hence, the points p and r are on different sides of the hyperplane $x_i = x_j$. Any continuous curve in $\mathbb{R}^n$ between p and r must pass through this hyperplane. That is, any such curve contains a point $q := (q_1, q_2, \ldots, q_n)$ for which $q_i = q_j$. As a result, this curve is not completely contained in the set $V_{\mathcal{P}}$. Therefore, p and r belong to different connected components of $V_{\mathcal{P}}$.

Theorem 3.3.3 immediately implies the following lower bound:

Theorem 3.3.4 *The time complexity of the element uniqueness problem for n real numbers in the linear decision-tree model is greater than or equal to $\lceil \log n! \rceil = n \log n - O(n)$.*

This theorem states that using comparisons, the arithmetic operations of addition and subtraction, and multiplication and division by constants, it is not possible to solve the element uniqueness problem in $o(n \log n)$ time. It does not rule out, however, $o(n \log n)$–time algorithms that can multiply and divide input elements and take square roots. In the next section, we will show that these operations do not significantly reduce the time complexity of the element uniqueness problem.

3.3.2 The general lower bound

The proof of Theorem 3.3.3 heavily depends on the fact that the set $R(w)$ associated with a leaf w is connected. For an algebraic decision-tree, this set is, in general, not connected. For example, for $n = 2$, let $R(w)$ be defined by the two inequalities $s_1^2 + s_2^2 - 1 \leq 0$ and $-8s_1^2 + s_2 + 2 \leq 0$. That is, $R(w)$ is the set of those points that are inside the unit-circle and below the parabola $s_2 = 8s_1^2 - 2$. This set clearly consists of two connected components.

In this section, we will prove that the arguments of Section 3.3.1 can, nevertheless, be generalized. As we will see, the number of connected components of the set $V_{\mathcal{P}}$ of YES-instances still gives a lower bound on the time complexity of the (decidable) decision problem $\mathcal{P}$. The proof will be based on the following result from algebraic topology.

Theorem 3.3.5 *Let k and g be positive integers, and let $F_1, F_2, \ldots, F_k$ be polynomials in n variables, each having degree less than or equal to g. Let*

$$W := \{(x_1, x_2, \ldots, x_n) \in \mathbb{R}^n : F_i(x_1, x_2, \ldots, x_n) = 0 \text{ for all } 1 \leq i \leq k\}.$$

The set W has at most $g(2g-1)^{n-1}$ connected components.

For a proof of this theorem, which is highly non-trivial, the reader is referred to the references given in the bibliographic notes at the end of this chapter. Observe that the upper bound on the number of connected components of the set W only depends on the number of variables and the degrees of the polynomials; it does not depend on the number of polynomials.

Consider an arbitrary algebraic decision-tree T, and let w be any leaf of T. Later in this section, we will show that the set $R(w) \subseteq \mathbb{R}^n$ of all inputs on which algorithm $\mathcal{A}_T$ terminates in w can be described by a system of polynomial equations and inequalities, each having degree less than or equal to two. Our goal is to derive an upper bound on the number of connected components of $R(w)$. This will be done by transforming the system of equations and inequalities that describe $R(w)$ into a system containing polynomial equations only and then applying Theorem 3.3.5. The details of this transformation are given in the following theorem.

Theorem 3.3.6 *Let a, b and c be non-negative integers, and let $E_1, \ldots, E_a$, $N_1, \ldots, N_b$, $P_1, \ldots, P_c$ be polynomials in n variables, each having degree less than or equal to two. Let W be the set of all points $(x_1, \ldots, x_n)$ in $\mathbb{R}^n$ such that the following is true:*

1. *$E_i(x_1, \ldots, x_n) = 0$ for all i with $1 \leq i \leq a$,*
2. *$N_i(x_1, \ldots, x_n) \leq 0$ for all i with $1 \leq i \leq b$, and*
3. *$P_i(x_1, \ldots, x_n) > 0$ for all i with $1 \leq i \leq c$.*

The set W has at most 3^{n+b+c} connected components.

Proof. It can be shown that the number $CC(W)$ of connected components of W is finite. Let $d := CC(W)$. For each j with $1 \leq j \leq d$, let $p_j \in \mathbb{R}^n$ be an arbitrary point in the j-th connected component of W. Define

$$\epsilon := \min\{P_i(p_j) : 1 \leq i \leq c, 1 \leq j \leq d\}.$$

Clearly, $\epsilon > 0$. Let W_ϵ be the set of all points $(x_1, \ldots, x_n) \in \mathbb{R}^n$, such that

- $E_i(x_1, \ldots, x_n) = 0$ for all i with $1 \leq i \leq a$,
- $N_i(x_1, \ldots, x_n) \leq 0$ for all i with $1 \leq i \leq b$, and
- $P_i(x_1, \ldots, x_n) \geq \epsilon$ for all i with $1 \leq i \leq c$.

Then, $W_\epsilon \subseteq W$ and W_ϵ contains the points $p_1, p_2, \ldots, p_d$.

We transform the equations and inequalities that define W_ϵ into a system of polynomial equations by introducing $b + c$ new variables $x_{n+1}, \ldots, x_{n+b+c}$. Let W' be the set of all points $(x_1, \ldots, x_{n+b+c}) \in \mathbb{R}^{n+b+c}$ such that

- $E_i(x_1,\ldots,x_n)=0$ for all i with $1\le i\le a$,
- $N_i(x_1,\ldots,x_n)+x_{n+i}^2=0$ for all i with $1\le i\le b$, and
- $P_i(x_1,\ldots,x_n)-x_{n+b+i}^2-\epsilon=0$ for all i with $1\le i\le c$.

The projection of W' onto the first n coordinates is exactly the set W_ϵ, i.e.,

$$W_\epsilon=\{(x_1,\ldots,x_n):\exists x_{n+1},\ldots,x_{n+b+c}\in\mathbb{R},(x_1,\ldots,x_{n+b+c})\in W'\}.$$

For each j with $1\le j\le d$, let p'_j be a point in W' such that its projection onto the first n coordinates is the point p_j. Since the points $p_1,p_2,\ldots,p_d$ are in pairwise distinct connected components of W and since $W_\epsilon\subseteq W$, it follows that the points $p'_1,p'_2,\ldots,p'_d$ are in pairwise distinct connected components of W'. Hence, $CC(W')\ge d$.

The set W' is defined by polynomial equations in $n+b+c$ variables, each having degree less than or equal to two. Therefore, by Theorem 3.3.5, we have

$$CC(W')\le 2\cdot 3^{n+b+c-1}\le 3^{n+b+c}.$$

This completes the proof. ∎

Now we are ready to prove the lower bound for algebraic decision-tree algorithms.

Theorem 3.3.7 *Let $\mathcal{P}$ be a decision problem that is decidable in the algebraic decision-tree model, and let $V_\mathcal{P}\subseteq\mathbb{R}^n$ be the corresponding set of YES-instances. The time complexity of $V_\mathcal{P}$ in the algebraic decision-tree model is greater than or equal to*

$$\frac{\log(CC(V_\mathcal{P}))-n\log 3}{1+2\log 3}.$$

Proof. Let T be an arbitrary algebraic decision-tree that decides $V_\mathcal{P}$, and let w be any leaf of T. Let $R(w)$ be the set of all points $(s_1,s_2,\ldots,s_n)\in\mathbb{R}^n$, such that the computation in T on input $s_1,s_2,\ldots,s_n$ terminates in w. We will derive an upper bound on the number of connected components of $R(w)$.

Consider the path $u_1,u_2,\ldots,u_{k+1}$ in T from the root u_1 to $u_{k+1}=w$. Let r be the number of nodes on this path that have exactly one child, and let s be the number of such nodes that are labeled with a $\sqrt{}$-assignment. We will define a system of $k+s$ polynomial equations and inequalities in the variables

$x_1,\ldots,x_{n+k}$. The variables $x_1,\ldots,x_n$ represent the input variables $s_1,\ldots,s_n$, whereas the variables $x_{n+1},\ldots,x_{n+k}$ represent the program variables of the nodes $u_1,\ldots,u_k$.

Let i be any integer with $1\le i\le k$, and consider node u_i. There are two possible cases.

Case 1: u_i has one child, i.e., u_i is a computation node.

Node u_i is labeled with an assignment of the form $Z(u_i):=A_1\&A_2$ or $Z(u_i):=\sqrt{A_1}$, where $\&\in\{+,-,*,/\}$, and, for $m=1,2$, (i) $A_m=Z(u_j)$ for some index j with $1\le j<i$, (ii) $A_m\in\{s_1,s_2,\ldots,s_n\}$, or (iii) $A_m=c$ for some real constant c. (See Definition 3.1.1.)

We add one equation to our system for this computation node u_i. Furthermore, depending on the form of the assignment, we may add one $\le$-inequality to our system. In Table 3.1, all possibilities are listed. For example, if the assignment is $Z(u_i):=s_a/Z(u_\ell)$, then we add the equation $x_{n+i}x_{n+\ell}-x_a=0$. Here, x_{n+i} represents the program variable $Z(u_i)$, $x_{n+\ell}$ represents $Z(u_\ell)$, and x_a represents the input variable s_a. If the assignment is $Z(u_i):=\sqrt{s_a}$, then we add the equation $x_{n+i}^2-x_a=0$ and the inequality $-x_{n+i}\le 0$. (Observe that $x_{n+i}=\sqrt{x_a}$ if and only if $x_{n+i}^2=x_a$ and $x_{n+i}\ge 0$.)

Case 2: u_i has two children, i.e., u_i is a comparison node.

Node u_i is labeled with a comparison of the form $A\bowtie 0$, where (i) $A=Z(u_j)$ for some index j with $1\le j<i$, or (ii) $A\in\{s_1,s_2,\ldots,s_n\}$.

In case the path in T to w proceeds from u_i to its left child, we do the following: If the comparison in u_i has the form $Z(u_j)\bowtie 0$, then we add the inequality $x_{n+j}\le 0$ to our system. Otherwise, the comparison in u_i has the form $s_a\bowtie 0$, in which case we add the inequality $x_a\le 0$.

In case the path to w proceeds from u_i to its right child, we do the following: If the comparison in u_i has the form $Z(u_j)\bowtie 0$, then we add the inequality $x_{n+j}>0$. Otherwise, the comparison in u_i has the form $s_a\bowtie 0$, in which case we add the inequality $x_a>0$.

Recall that r denotes the number of computation nodes on the path to w, and s denotes the number of computation nodes on this path that are labeled with a $\sqrt{}$-assignment. Let t be the number of times this path proceeds from a comparison node to its left child. Then we have obtained a system of r polynomial equations, $s+t$ polynomial $\le$-inequalities, and $k-r-t$ polynomial $>$-inequalities, in the variables $x_1,\ldots,x_{n+k}$. Each of these polynomials has degree less than or equal to two. Let $W\subseteq\mathbb{R}^{n+k}$ be the set of all points that

assignment	equation/inequality
$Z(u_i) := Z(u_j) + Z(u_\ell)$	$x_{n+i} - x_{n+j} - x_{n+\ell} = 0$
$Z(u_i) := Z(u_j) - Z(u_\ell)$	$x_{n+i} - x_{n+j} + x_{n+\ell} = 0$
$Z(u_i) := Z(u_j) * Z(u_\ell)$	$x_{n+i} - x_{n+j}x_{n+\ell} = 0$
$Z(u_i) := Z(u_j)/Z(u_\ell)$	$x_{n+i}x_{n+\ell} - x_{n+j} = 0$
$Z(u_i) := \sqrt{Z(u_j)}$	$x_{n+i}^2 - x_{n+j} = 0$ and $-x_{n+i} \leq 0$
$Z(u_i) := s_a + Z(u_\ell)$	$x_{n+i} - x_a - x_{n+\ell} = 0$
$Z(u_i) := s_a - Z(u_\ell)$	$x_{n+i} - x_a + x_{n+\ell} = 0$
$Z(u_i) := Z(u_\ell) - s_a$	$x_{n+i} - x_{n+\ell} + x_a = 0$
$Z(u_i) := s_a * Z(u_\ell)$	$x_{n+i} - x_a x_{n+\ell} = 0$
$Z(u_i) := s_a/Z(u_\ell)$	$x_{n+i}x_{n+\ell} - x_a = 0$
$Z(u_i) := Z(u_\ell)/s_a$	$x_{n+i}x_a - x_{n+\ell} = 0$
$Z(u_i) := s_a + s_b$	$x_{n+i} - x_a - x_b = 0$
$Z(u_i) := s_a - s_b$	$x_{n+i} - x_a + x_b = 0$
$Z(u_i) := s_a * s_b$	$x_{n+i} - x_a x_b = 0$
$Z(u_i) := s_a/s_b$	$x_{n+i}x_b - x_a = 0$
$Z(u_i) := \sqrt{s_a}$	$x_{n+i}^2 - x_a = 0$ and $-x_{n+i} \leq 0$
$Z(u_i) := c + Z(u_\ell)$	$x_{n+i} - c - x_{n+\ell} = 0$
$Z(u_i) := c - Z(u_\ell)$	$x_{n+i} - c + x_{n+\ell} = 0$
$Z(u_i) := Z(u_\ell) - c$	$x_{n+i} - x_{n+\ell} + c = 0$
$Z(u_i) := c * Z(u_\ell)$	$x_{n+i} - cx_{n+\ell} = 0$
$Z(u_i) := c/Z(u_\ell)$	$x_{n+i}x_{n+\ell} - c = 0$
$Z(u_i) := Z(u_\ell)/c$	$cx_{n+i} - x_{n+\ell} = 0$
$Z(u_i) := c + s_b$	$x_{n+i} - c - x_b = 0$
$Z(u_i) := c - s_b$	$x_{n+i} - c + x_b = 0$
$Z(u_i) := s_b - c$	$x_{n+i} - x_b + c = 0$
$Z(u_i) := c * s_b$	$x_{n+i} - cx_b = 0$
$Z(u_i) := c/s_b$	$x_{n+i}x_b - c = 0$
$Z(u_i) := s_b/c$	$cx_{n+i} - x_b = 0$
$Z(u_i) := c + d$	$x_{n+i} - c + d = 0$
$Z(u_i) := c - d$	$x_{n+i} - c + d = 0$
$Z(u_i) := c * d$	$x_{n+i} - cd = 0$
$Z(u_i) := c/d$	$dx_{n+i} - c = 0$
$Z(u_i) := \sqrt{c}$	$x_{n+i}^2 - c = 0$ and $-x_{n+i} \leq 0$

Table 3.1: *The equations and inequalities corresponding to all possible assignments of computation node u_i. The indices j and ℓ satisfy $1 \leq j < i$ and $1 \leq \ell < i$; the indices a and b satisfy $1 \leq a \leq n$ and $1 \leq b \leq n$; and c and d are real constants.*

satisfy these equations and inequalities. Then, by Theorem 3.3.6, W has at most $3^{n+2k+s-r}$ connected components.

The projection of W onto the first n coordinates is equal to the set $R(w)$. This implies that $CC(R(w)) \leq CC(W)$ and, hence, $CC(R(w)) \leq 3^{n+2k+s-r}$. Let h be the height of our algebraic decision-tree T. Then, since $k \leq h$ and $s \leq r$, we have proved that $CC(R(w)) \leq 3^{n+2h}$.

Now we can complete the proof of the theorem. Recall that $V_{\mathcal{P}} \subseteq \mathbb{R}^n$ is the set of YES-instances for the decision problem $\mathcal{P}$. A leaf w of T is called a *YES-leaf*, if its label is YES. Since

$$V_{\mathcal{P}} = \bigcup_{w:\text{YES-leaf of } T} R(w),$$

we have

$$CC(V_{\mathcal{P}}) \leq \sum_{w:\text{YES-leaf of } T} CC(R(w)).$$

Hence, the number of connected components of $V_{\mathcal{P}}$ is less than or equal to 3^{n+2h} times the number of YES-leaves of T. Since T has height h, it has at most 2^h leaves. Therefore,

$$CC(V_{\mathcal{P}}) \leq 3^{n+2h} \cdot 2^h.$$

Taking logarithms and rewriting this inequality, we obtain

$$h \geq \frac{\log(CC(V_{\mathcal{P}})) - n\log 3}{1 + 2\log 3},$$

which is exactly what we wanted to show. ∎

Remark 3.3.8 Let $\mathcal{P}$ be a decision problem that is decidable in the algebraic decision-tree model, and let $V \subseteq \mathbb{R}^n$ be the corresponding set of YES-instances. It follows from the proof of Theorem 3.3.7 that the number of connected components of V is finite.

3.3.3 Some applications

Let us again consider the element uniqueness problem. As we have seen already in Section 3.3.1, the corresponding subset V of $\mathbb{R}^n$ has at least $n!$ connected components. Hence, Theorem 3.3.7 gives a lower bound of

$$\frac{\log n! - n\log 3}{1 + 2\log 3} = \Omega(n \log n)$$

on the time complexity of this problem.

Theorem 3.3.9 *The time complexity of the element uniqueness problem for n real numbers in the algebraic decision-tree model is $\Omega(n \log n)$.*

Using simple reductions, this theorem implies other interesting lower bounds:

Corollary 3.3.10 *The following two problems have time complexity $\Omega(n \log n)$ in the algebraic computation-tree model:*

1. *the sorting problem for n real numbers, and*
2. *the closest pair problem, i.e., given a set S of n points in $\mathbb{R}^d$, compute two distinct points of S whose distance is minimum.*

Proof. Let $\mathcal{A}$ be an arbitrary algebraic computation-tree algorithm that solves the sorting problem, and let $T(n)$ be its time complexity. The following algebraic decision-tree algorithm $\mathcal{B}$ solves the element uniqueness problem: On an input consisting of n real numbers $s_1, s_2, \ldots, s_n$, $\mathcal{B}$ first uses algorithm $\mathcal{A}$ to sort them. Then, $\mathcal{B}$ compares all pairs of elements that are neighbors in the sorted sequence. Algorithm $\mathcal{B}$ returns YES if no two equal elements are encountered; otherwise, it returns NO.

Algorithm $\mathcal{B}$ has time complexity $T(n) + O(n)$, which, by Theorem 3.3.9, must be $\Omega(n \log n)$. It follows that $T(n) = \Omega(n \log n)$.

The lower bound for the closest pair problem follows immediately from Theorem 3.3.9 because the input sequence contains two equal elements if and only if the distance of the closest pair is zero. ∎

We now give an example of a problem that is not decidable in the algebraic decision-tree model. Consider a decision problem $\mathcal{P}$, and let $V \subseteq \mathbb{R}^n$ be the corresponding set of YES-instances. If V has an infinite number of connected components, then it follows from Remark 3.3.8 that $\mathcal{P}$ is not decidable in the algebraic decision-tree model. Hence, by taking $V := \mathbb{N} = \{0, 1, 2, \ldots\}$, we obtain the following result:

Theorem 3.3.11 *There is no algebraic decision-tree algorithm that, when given an arbitrary real number x as input, returns YES if $x \in \mathbb{N}$, and NO if $x \notin \mathbb{N}$.*

3.4 A lower bound for constructing spanners

In this section, we will use Theorem 3.3.7 to prove an $\Omega(n \log n)$ lower bound for constructing t-spanners. We try to make this lower bound as strong as possible, i.e., it should hold for a very general class of spanner graphs. To be more precise, the lower bound that we will prove holds for any value of $t > 1$, i.e., t may even depend on the number n of input points. Also, it holds for spanners that include additional vertices that were not part of the input. We formally define these spanners as follows:

Let S be a multiset of n points in $\mathbb{R}^d$. We consider graphs $G = (V, E)$, such that V is a finite multiset of points in $\mathbb{R}^d$ that contains all points of S. Let $t > 1$ be a real number. Consider a graph $G = (V, E)$ that satisfies these two conditions. Assume that for any two points p and q of S, there is a path in G between p and q whose length is less than or equal to $t|pq|$. Hence, if $V = S$, then G is a t-spanner for S. If the size of V is larger than that of S, then we call G a *Steiner t-spanner* for S. In this case, the points of V that are not in S are called the *Steiner points* of G.

To be as general as possible, the graph G may have multiple vertices that represent the same point: There may be points u and v in S that are distinct as elements of S, but that represent the same point in $\mathbb{R}^d$. Similarly, there may be a point u of S and a Steiner point v that represent the same point in $\mathbb{R}^d$. Finally, there may be Steiner points u and v that are distinct as vertices of G, but that represent the same point. Hence, the graph G may have edges of length zero.

Throughout this section, we consider algebraic computation-tree algorithms that construct Steiner t-spanners with $o(n \log n)$ edges. (Clearly, any algorithm that constructs Steiner t-spanners with $\Omega(n \log n)$ edges takes $\Omega(n \log n)$ time.) Moreover, we will focus on algorithms that construct Steiner t-spanners for one-dimensional multisets, i.e., multisets of real numbers. We will prove that even this one-dimensional case has an $\Omega(n \log n)$ lower bound. Of course, this implies the same lower bound for any dimension $d \geq 1$.

3.4.1 A reduction from the element uniqueness problem

Let $s_1, s_2, \ldots, s_n, t$ be a sequence of $n+1$ real numbers, such that $t > 1$. The main observation is encapsulated in the following key idea:

Key Idea: If $s_i = s_j$ for some i and j with $i \neq j$, then any Steiner t-spanner for $s_1, s_2, \ldots, s_n$ contains a path between s_i and s_j whose length is less than or equal to $t|s_i - s_j| = 0$. In particular, each edge on this path has length zero.

We have to be careful in formalizing this, however, because the spanner may contain Steiner points.

Let $\mathcal{A}$ be an arbitrary algebraic computation-tree algorithm that, when given a sequence of n real numbers $s_1, s_2, \ldots, s_n$ and a real number $t > 1$, constructs a Steiner t-spanner for the multiset $S = \{s_1, s_2, \ldots, s_n\}$ of n points on the one-dimensional real line. We may assume that each vertex of the spanner graph constructed by $\mathcal{A}$ is labeled as either being an element of S or being a Steiner point.

We will show how algorithm $\mathcal{A}$ can be used to solve the element uniqueness problem. Consider the following algorithm that takes as input a sequence $S = (s_1, s_2, \ldots, s_n)$ of n real numbers:

Step 1: Choose an arbitrary real number $t > 1$, and run algorithm $\mathcal{A}$ on the input sequence $s_1, s_2, \ldots, s_n, t$. Let G be the resulting Steiner t-spanner.

Step 2: Construct the subgraph G' of G such that G' contains the same vertices as G, and G' contains all edges of G having length zero.

Step 3: Compute the connected components of the graph G'.

Step 4: For each connected component of G', check if it contains two or more distinct elements (i.e., elements having distinct indices) of S among its vertices. If this is the case for some connected component, return NO. Otherwise, return YES.

It is not difficult to see that this algorithm correctly solves the element uniqueness problem. Hence, given the Steiner t-spanner G, we can solve the element uniqueness problem in a time that is proportional to the number of edges of G, which we assumed to be $o(n \log n)$. Therefore, it follows from Theorem 3.3.9 that the worst-case running time of algorithm $\mathcal{A}$ is $\Omega(n \log n)$.

Theorem 3.4.1 *Let $d \geq 1$ be an integer constant. In the algebraic computation-tree model, any algorithm that, when given a multiset S of n points in $\mathbb{R}^d$, and a real number $t > 1$, constructs a Steiner t-spanner for S, takes $\Omega(n \log n)$ time in the worst case.*

This lower bound proof is unsatisfying, in the sense that most existing algorithms to construct t-spanners, assume implicitly that all input points are pairwise distinct. When the inputs are thus constrained, the above proof does not work: If the points are known to be pairwise distinct, then the element uniqueness problem can be solved in $O(1)$ time, because the output is always YES. In the next section, we will consider algorithms that construct Steiner spanners for inputs consisting of pairwise distinct points.

3.4.2 A lower bound for a set of pairwise distinct points

The main result of this section is encapsulated below.

Main Result: In the algebraic computation-tree model, the lower bound of $\Omega(n \log n)$ for the Steiner t-spanner construction problem holds even if the input is known to consist of pairwise distinct points. The proof effectively uses a lower bound of $\Omega(n \log n)$ for the *mingap* problem.

As in the previous section, we will only consider algorithms that compute Steiner t-spanners for one-dimensional point sets, i.e., sets of real numbers.

Throughout this section, we fix an integer n. Moreover, $\mathcal{A}$ denotes an arbitrary algebraic computation-tree algorithm that, when given a set S of n pairwise distinct real numbers, and a real number $t > 1$, constructs a Steiner t-spanner for S with $o(n \log n)$ edges. Hence, the output of $\mathcal{A}$ is a graph having as its vertices the elements of S and (possibly) some additional Steiner points. We assume that each vertex of this graph is labeled as either being an element of S or being a Steiner point. If two input elements of S are equal, then algorithm $\mathcal{A}$ is not defined.

Our goal is to prove that the worst-case running time of $\mathcal{A}$ is $\Omega(n \log n)$. First observe that $\mathcal{A}$ solves a computation problem. Therefore, in order to apply Theorem 3.3.7, we have to define an appropriate algorithm $\mathcal{D}$, such that

1. $\mathcal{D}$ solves a decision problem, i.e., it returns YES or NO,
2. $\mathcal{D}$ has a running time that is within a constant factor of $\mathcal{A}$'s running time, and
3. the set of YES-instances of $\mathcal{D}$, considered as a subset of $\mathbb{R}^n$, consists of many (at least $n!$, in our case) connected components.

Roughly stated, algorithm $\mathcal{D}$ takes the input to algorithm $\mathcal{A}$ and returns YES if the length of the shortest edge in the spanner returned by $\mathcal{A}$ is greater than a specified length. Consequently, proving a lower bound of $\Omega(n \log n)$ for algorithm $\mathcal{D}$ implies the same lower bound for algorithm $\mathcal{A}$.

However, the above strategy faces the following hurdle: Since algorithm $\mathcal{A}$ (and, consequently, algorithm $\mathcal{D}$) only accepts pairwise distinct real numbers, it is not defined on all points in $\mathbb{R}^n$. In fact, the subset of $\mathbb{R}^n$ on which it is defined trivially contains at least $n!$ connected components. Thus we cannot apply Theorem 3.3.7 to these algorithms.

To emphasize the point that Theorem 3.3.7 cannot be applied here, consider, for example, an algorithm $\mathcal{X}$ that takes as input any sequence of n pairwise distinct real numbers, and simply returns YES (therefore, it runs in $O(1)$ time). Since algorithm $\mathcal{X}$ is not defined if two input elements are equal, the subset of $\mathbb{R}^n$ accepted by this algorithm has at least $n!$ connected components.

Therefore, in order to apply Theorem 3.3.7, we carefully define algorithm $\mathcal{D}$ so that it can take *any* point of $\mathbb{R}^n$ as input and that its set of YES-instances still has $\Omega(n!)$ connected components. We will define algorithm $\mathcal{D}$ in three steps.

1. First, we define an algorithm $\mathcal{B}$ that takes pairwise distinct real numbers as input. This algorithm runs algorithm $\mathcal{A}$ on this input, and returns the length L of a shortest edge of non-zero length in the graph that $\mathcal{A}$ computes.

2. Next, we use algorithm $\mathcal{B}$ to define a positive real number L^*. Algorithm $\mathcal{C}$ takes pairwise distinct real numbers as input. It runs algorithm $\mathcal{B}$ on this input, and returns YES if and only if the ouput L of $\mathcal{B}$ is greater than or equal to L^*.

3. Finally, we change algorithm $\mathcal{C}$, such that it is well-defined on *any* input sequence of real numbers. The resulting algorithm is the algebraic decision-tree algorithm $\mathcal{D}$ we are looking for.

In the rest of this section, we will fill in the details.

Algorithm $\mathcal{B}$

Algorithm $\mathcal{B}$ does the following on an input consisting of n pairwise distinct real numbers $s_1, s_2, \ldots, s_n$ and a real number $t > 1$: It first runs algorithm $\mathcal{A}$

on the input $s_1, s_2, \ldots, s_n, t$. Let G be the Steiner t-spanner that is computed by $\mathcal{A}$. By considering all edges of G, algorithm $\mathcal{B}$ selects a shortest edge of non-zero length and returns the length L of this edge.

The following lemma relates the output L of algorithm $\mathcal{B}$ to the minimum distance of its input sequence. For real numbers $s_1, s_2, \ldots, s_n$, we define

$$\mathit{mingap}(s_1, s_2, \ldots, s_n) := \min\{|s_i - s_j| : 1 \leq i < j \leq n\}.$$

Lemma 3.4.2 *The real number L that is returned by algorithm $\mathcal{B}$ satisfies*

$$0 < L \leq t \cdot \mathit{mingap}(s_1, s_2, \ldots, s_n).$$

Proof. Let i and j be two indices, such that $|s_i - s_j| = \mathit{mingap}(s_1, s_2, \ldots, s_n)$. Observe that since the input elements are pairwise distinct, we have $|s_i - s_j| > 0$. The graph G constructed by algorithm $\mathcal{A}$ contains a path between s_i and s_j whose length is less than or equal to $t|s_i - s_j|$. The length of each edge on this path is obviously less than or equal to $t|s_i - s_j|$. The claim follows because this path contains at least one edge of non-zero length. ∎

Let $T_{\mathcal{A}}(n, t)$ and $T_{\mathcal{B}}(n, t)$ denote the worst-case running times of algorithms $\mathcal{A}$ and $\mathcal{B}$, respectively. Then, the fact that the graph G has $o(n \log n)$ edges implies that

$$T_{\mathcal{B}}(n, t) \leq T_{\mathcal{A}}(n, t) + o(n \log n).$$

Algorithm $\mathcal{C}$

We fix a real number $t > 1$. Before we define algorithm $\mathcal{C}$, we use algorithm $\mathcal{B}$ to define a positive real number L^* as follows: For each permutation π of the integers $1, 2, \ldots, n$, let L_π be the output of algorithm $\mathcal{B}$ when given as input the sequence $\pi(1), \pi(2), \ldots, \pi(n), t$. Among all these $n!$ outputs, let L^* be one that has the minimum value.

Now we can define algorithm $\mathcal{C}$. It only takes input sequences of our fixed length n, consisting of n pairwise distinct real numbers. On input $s_1, s_2, \ldots, s_n$, algorithm $\mathcal{C}$ does the following: It first runs algorithm $\mathcal{B}$ on the input sequence $s_1, s_2, \ldots, s_n, t$. Let L be the output of $\mathcal{B}$. Algorithm $\mathcal{C}$ returns YES if $L \geq L^*$, and NO otherwise.

We remark that it is not necessary to compute L^*, which would take a lot of time. For our proof, it is sufficient that algorithm $\mathcal{C}$ *exists*. In other

words, since algorithm $\mathcal{C}$ only takes inputs of our fixed length n, and since we also fixed t, we may assume that it "knows" the value L^*.

It is clear that the running time of algorithm $\mathcal{C}$ is within a constant factor of $\mathcal{B}$'s running time.

Algorithm $\mathcal{D}$

Algorithm $\mathcal{C}$ is only defined for inputs consisting of n pairwise distinct real numbers. As a result, $\mathcal{C}$ can safely perform operations such as $z := x/(s_i - s_j)$ and $z := \sqrt{y}$, without having to worry whether the denominator $s_i - s_j$ is zero, or whether $y \geq 0$. Our final algorithm $\mathcal{D}$ will take any sequence $s_1, s_2, \ldots, s_n$ of n real numbers as input. On such an input, $\mathcal{D}$ performs the same computation as $\mathcal{C}$ does on the same input, except that each operation of the form $z := x/y$ is performed by $\mathcal{D}$ as

if $y \neq 0$
then $z := x/y$
else return YES and terminate
endif

and each operation of the form $z := \sqrt{y}$ is performed by $\mathcal{D}$ as

if $y \geq 0$
then $z := \sqrt{y}$
else return YES and terminate
endif

Since $\mathcal{C}$ is a well-defined algorithm for inputs consisting of n pairwise distinct real numbers, it will always be the case that $y \neq 0$ when the operation $z := x/y$ is performed. When two input elements are equal, it may still be true that $y \neq 0$, although this is not necessarily the case. Similarly, if the input elements are pairwise distinct, it will always be the case that $y \geq 0$ when the operation $z := \sqrt{y}$ is performed. When two input elements are equal, y may still be non-negative, although, again, this is not necessarily the case.

It is clear that $\mathcal{C}$ and $\mathcal{D}$ give the same output when given, as input, the same sequence of n pairwise distinct real numbers. If these numbers are not pairwise distinct, then $\mathcal{C}$ is not defined, whereas $\mathcal{D}$ is, although its output may not have a meaning at all. Finally, observe that the running time of $\mathcal{D}$ is within a constant factor of that of $\mathcal{C}$.

Analysis of algorithm $\mathcal{D}$

We now prove that the worst-case running time of algorithm $\mathcal{D}$ is $\Omega(n \log n)$. This will imply the same lower bound on the running time of our target algorithm $\mathcal{A}$.

Let W be the set of all points $(s_1, s_2, \ldots, s_n) \in \mathbb{R}^n$ such that algorithm $\mathcal{D}$ returns YES on the input sequence $s_1, s_2, \ldots, s_n$. The lower bound will follow from the following lemma:

Lemma 3.4.3 *The set W has at least $n!$ connected components.*

Proof. Let π and ρ be two distinct permutations of $1, 2, \ldots, n$. We will show that the points

$$p := (\pi(1), \pi(2), \ldots, \pi(n))$$

and

$$r := (\rho(1), \rho(2), \ldots, \rho(n))$$

belong to different connected components of W. (Observe that both these points are elements of W.) This will prove the lemma.

Since π and ρ are distinct permutations, there are indices i and j with $1 \leq i \leq n$ and $1 \leq j \leq n$, such that $\pi(i) < \pi(j)$ and $\rho(i) > \rho(j)$.

Consider an arbitrary continuous curve C in $\mathbb{R}^n$ that connects p and r. We will show that C contains a point q that does not belong to the set W. From this, it will follow that p and r are in different connected components of W. In order to guarantee that $q \notin W$, the point $q = (q_1, q_2, \ldots, q_n)$ must have the property that $L < L^*$, where L is the output of algorithm $\mathcal{B}$ on input $q_1, q_2, \ldots, q_n, t$. Moreover, we have to take care that the coordinates of q are pairwise distinct.

Since the curve C passes through the hyperplane $x_i = x_j$, it contains points for which the absolute difference between the i-th and j-th coordinates is positive but arbitrarily small. However, for such points $q = (q_1, q_2, \ldots, q_n)$, there may be two distinct indices k and ℓ, such that $q_k = q_\ell$. We do not have any control over algorithm $\mathcal{D}$ when given such a point q as input. Therefore, we proceed as follows: We take for q the first point on the curve C, such that

$$\mathit{mingap}(q_1, q_2, \ldots, q_n) \leq L^*/(2t).$$

We will see below that the coordinates of q are pairwise distinct. If we run algorithm $\mathcal{B}$ on input $q_1, q_2, \ldots, q_n, t$, then, by Lemma 3.4.2, the output L

satisfies $L \leq t \cdot L^*/(2t) < L^*$. Hence, point q is not contained in the set W. In the rest of the proof, we will formalize this.

We parameterize the curve C as $C(\tau)$, $0 \leq \tau \leq 1$, where $C(0) = p$ and $C(1) = r$. For each k with $1 \leq k \leq n$, we write the k-th coordinate of the point $C(\tau)$ as $C(\tau)_k$. We define

$$\tau_0 := \min\{0 \leq \tau \leq 1 : \mathit{mingap}(C(\tau)_1, C(\tau)_2, \ldots, C(\tau)_n) \leq L^*/(2t)\}.$$

Observe that τ_0 exists because the curve C passes through the hyperplane $x_i = x_j$ and the function *mingap* is continuous along C.

Let $q := C(\tau_0)$. We write this point as $q = (q_1, q_2, \ldots, q_n)$. Clearly, we have

$$\mathit{mingap}(q_1, q_2, \ldots, q_n) \leq L^*/(2t).$$

Also, by Lemma 3.4.2, and since $C(0) = p \in W$, we have

$$\mathit{mingap}(C(0)_1, C(0)_2, \ldots, C(0)_n) \geq L^*/t > L^*/(2t).$$

The value of τ_0 is the first "time" at which the *mingap*-function is less than or equal to $L^*/(2t)$. Since this function is continuous along C, we have $\mathit{mingap}(q_1, q_2, \ldots, q_n) > 0$. Hence, $(q_1, q_2, \ldots, q_n)$ is a sequence of n pairwise distinct real numbers.

Consider what happens when we run algorithm $\mathcal{D}$ on the input sequence $q_1, q_2, \ldots, q_n$. First, algorithm $\mathcal{B}$ is run on the input sequence $q_1, q_2, \ldots, q_n, t$. Let L be the output of $\mathcal{B}$. By Lemma 3.4.2, we have

$$L \leq t \cdot \mathit{mingap}(q_1, q_2, \ldots, q_n).$$

Hence, $L \leq t \cdot L^*/(2t) < L^*$ and, therefore, algorithm $\mathcal{D}$ returns NO. This implies that point q does not belong to the set W. This completes the proof. ■

Recall that we denote the number of connected components of the set W by $CC(W)$. Lemma 3.4.3 and Theorem 3.3.7 imply that the running time of any algebraic decision-tree algorithm that decides the set W is bounded from below by

$$\Omega(\log(CC(W)) - n) = \Omega(n \log n).$$

Since $\mathcal{D}$ is such an algorithm, it follows that for our fixed values of n and t, the worst-case running time of $\mathcal{D}$ is greater than or equal to $cn \log n$, where

c is a positive constant independent of n and t. This, in turn, implies that there is an input on which algorithm $\mathcal{A}$ takes time at least $c'n \log n$, for some constant $c' > 0$. Since c' does not depend on n and t, this implies that the lower bound holds for all values of n and t. Hence, we have proved the following theorem.

Theorem 3.4.4 *Let $d \geq 1$ be an integer constant. In the algebraic computation-tree model, any algorithm that, when given a set S of n pairwise distinct points in $\mathbb{R}^d$ and a real number $t > 1$, constructs a Steiner t-spanner for S, takes $\Omega(n \log n)$ time in the worst case.*

Our lower bound proof of Theorem 3.4.4 holds for inputs consisting of pairwise distinct points. In computational geometry, we often make stronger assumptions on the input, e.g., no three points are on a line, no four points are in a two-dimensional plane, etc. We say that a set S of points in $\mathbb{R}^d$ is in *general position*, if for each k with $3 \leq k \leq d+1$, no k points of S are contained in a $(k-2)$-dimensional subspace of $\mathbb{R}^d$. For such general position inputs, our lower bound proof does not hold; the proof heavily uses the fact that the points are on a line.

Open Problem: Let $d \geq 2$ be an integer constant. Prove that, in the algebraic computation-tree model, any algorithm that, when given a set S of n points in $\mathbb{R}^d$ that are in general position and a real number $t > 1$, constructs a Steiner t-spanner for S, takes $\Omega(n \log n)$ time in the worst case.

Exercises

3.1 Prove that $\log n! = n \log n - O(n)$.

3.2 Prove Theorem 3.3.5 for $n = 1$.

3.3 In the proof of Theorem 3.3.7, it is mentioned that the projection of the set W onto the first n coordinates is equal to the set $R(w)$. Prove this claim.

3.4 Prove that, in the algebraic computation-tree model, the following problems have time complexity $\Omega(n \log n)$:

```
Algorithm NaturalNumber(x)
(* x is a real number *)
if x < 0
then return NO
else k := 0;
     while k ≤ x
     do k := k + 1
     endwhile;
     ℓ := k − 1;
     if x > ℓ
     then return NO
     else return YES
     endif
endif
```

Figure 3.1: *An algorithm that decides the set* $\mathbb{N} = \{0, 1, 2, \ldots\}$.

- Constructing the convex hull of a set of n points in the plane. (The convex hull vertices should be reported in clockwise or counter clockwise order.)
- Constructing the Voronoi diagram of a set of n points in the plane.
- Constructing an arbitrary triangulation of a set of n points in the plane.

3.5 It is known that the sorting problem for a set of n arbitrary *integers* has an $\Omega(n \log n)$ lower bound in the algebraic computation-tree model. Use this fact to prove that the sorting problem for n pairwise distinct integers has the same lower bound in this model.

3.6 In Figure 3.1, an algorithm is given that takes an arbitrary real number x as input, and returns YES if and only if $x \in \mathbb{N}$. Does this contradict Theorem 3.3.11?

3.7 Prove that there is no algebraic computation-tree algorithm that, on an arbitrary input $x \in \mathbb{R}$, computes the value $\lfloor x \rfloor$. (*Hint:* Use Theorem 3.3.11.)

3.8 Prove that there is no algebraic computation-tree algorithm that, on an arbitrary input $x \in \mathbb{R}$, computes the value $\sin x$.

3.9 We have seen in Remark 3.3.8 that any set $V \subseteq \mathbb{R}^n$, having an infinite number of connected components, is not decidable in the algebraic decision-tree model. Give an example of a set $V \subseteq \mathbb{R}^2$ with a finite number of connected components that is not decidable in this model. (In fact, there is such a set V consisting of only one connected component.)

3.10 In the algebraic computation-tree model, square roots can be computed in unit time. Prove that Theorem 3.3.7 also holds (with different constant factors) if k-th roots can be computed in unit time, where k is any element from a finite set of positive integers. (The size of this set is a constant that does not depend on n.)

3.11 Let $A[1 \,..\, n]$ and $B[1 \,..\, m]$ be arrays, and assume that $B[i] \in \{1, 2, \ldots, n\}$ for all i with $1 \leq i \leq m$. An assignment of the form $x := A[B[i]]$ is called an *indirect-addressing assignment*. Algorithms that use such assignments as unit-cost operations do not work in the algebraic computation-tree model. Explain why this is the case. Observe that this implies that the lower bound proofs that have been presented in this chapter are not valid for such algorithms.

3.12 For any sequence $s_1, s_2, \ldots, s_n$ of real numbers, we define a *gap* to be the absolute value of the difference of two elements s_i and s_j (with $i \neq j$) that are consecutive in the sorted sequence. The *uniformgap*-problem is defined as follows: Given a sequence $s_1, s_2, \ldots, s_n$ of real numbers, and given a real number g, decide whether all $n - 1$ gaps are equal to g.

Prove an $\Omega(n \log n)$ lower bound on the time complexity of the *uniformgap*-problem in the algebraic decision-tree model.

3.13 The *maxgap*-problem is defined as follows: Given a sequence $s_1, s_2, \ldots, s_n$ of real numbers, compute the *maximum gap* in this sequence, which is the largest absolute value of the difference of any two elements s_i and s_j (with $i \neq j$) that are consecutive in the sorted sequence.

Use Exercise 3.12 to prove an $\Omega(n \log n)$ lower bound on the time complexity of the *maxgap*-problem in the algebraic computation-tree model.

3.14 Consider the algebraic computation-tree model in which, additionally, any indirect-addressing operation takes unit time, and in which the floor-function can be computed in unit time. Prove that in this model, the *maxgap*-problem can be solved in $O(n)$ time.

Bibliographic notes

Sections 3.1, 3.2, and 3.3 are based on the books by Mehlhorn [1984b] and Preparata and Shamos [1988].

Ford and Johnson [1959] introduced decision-trees as a model to study comparison-based sorting algorithms. In 1966, Rabin generalized decision-trees to the more general class of algebraic computation-trees; see Rabin [1972] and Reingold [1972]. Dobkin and Lipton [1979] introduced the linear decision-tree model and proved Theorem 3.3.3. Theorems 3.3.6 and 3.3.7 are due to Ben-Or [1983], who extended earlier work by Steele and Yao [1982]. Theorem 3.3.5 was proved independently by Milnor [1964] and Thom [1965]. In the proof of Theorem 3.3.6, it is mentioned that the set W has a finite number of connected components. This was proved by Milnor [1968]. A wealth of information about algebraic algorithms can be found in the book by Bürgisser, Clausen, and Shokrollahi [1997].

The lower bound for constructing spanners that is presented in Section 3.4 is due to Chen, Das, and Smid [2001].

The $\Omega(n \log n)$ lower bound for sorting n integers in the algebraic computation-tree model that is mentioned in Exercise 3.5 is due to Yao [1991]. A solution to this exercise can be found in Chen, Das, and Smid [2001].

The $\Omega(n \log n)$ lower bounds for the *uniformgap*- and *maxgap*-problems in Exercises 3.12 and 3.13 are due to Lee and Wu [1986]. Earlier, these lower bounds were proved in Manber and Tompa [1985] for the linear decision-tree model. The $O(n)$–time algorithm in the more powerful model in Exercise 3.14 is due to Gonzalez [1975]; see also Preparata and Shamos [1988].

We have only considered deterministic algebraic algorithms. In Yao [1977], a general technique for proving lower bounds on the expected running time of *randomized* algorithms is presented. This technique is also described in the books by Mehlhorn [1984b] and Motwani and Raghavan [1995].