LUND UNIVERSITY

Geometric Decompositions and Networks - Approximation Bounds and Algorithms

Gudmundsson, Joachim

2000

Link to publication

Citation for published version (APA):
Gudmundsson, J. (2000). *Geometric Decompositions and Networks - Approximation Bounds and Algorithms*. [Doctoral Thesis (monograph)]. Department of Computer Science, Lund University.

Total number of authors:
1

General rights
Unless other specific re-use rights are stated the following general rights apply:
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.
• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Read more about Creative commons licenses: https://creativecommons.org/licenses/

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

LUND UNIVERSITY

PO Box 117
221 00 Lund
+46 46-222 00 00

Download date: 25. Jun. 2024

Geometric Decompositions and Networks

Approximation Bounds and Algorithms

Joachim Gudmundsson
Department of Computer Science
Lund University

Department of Computer Science
Lund University
Box 118
S–221 00 Lund
Sweden

E-mail: Joachim.Gudmundsson@cs.lth.se

©2000 by Joachim Gudmundsson
Printed in Sweden

ISSN 1650-1268
ISBN 91–7874–098–3
Dissertation 16

Abstract

In this thesis we focus on four problems in computational geometry:

In the first four chapters we consider the problem of covering an arbitrary polygon with simpler polygons, i.e., rectangles. We present several approximation algorithms for this problem, and also some lower bounds on the number of rectangles needed in a covering of a hole-free polygon and on the time-complexity for this and related problems.

Then, we consider a generalization of the well-known Euclidean traveling salesman problem (TSP), namely the TSP with neighborhoods problem. In the TSP with neighborhoods problem we are given a collection of polygonal regions, and we seek the shortest tour that visits each neighborhood at least once. We give approximation algorithms for the problem and also show a result on the hardness of the problem.

Next we turn our attention to the problem of finding a t-spanner of a complete geometric graph. The aim is to produce a sparse graph with a small number of edges and with low total weight, that is almost as "good" as a complete graph. With good we mean that for every pair of points in the graph there exists a path in the spanner graph that is at most t times longer than the distance between the two points. We present several approximation algorithms for this problem.

In the final chapter of the thesis we introduce the concept of *higher-order Delaunay triangulations*. We give an algorithm to compute which edges can be included in a higher-order Delaunay triangulation. We show that for 1-order Delaunay triangulations, most of the criteria we study can be optimized in $O(n \log n)$ time, for example, minimizing the number of local minima, the number of local extrema, the maximum angle, area triangle, and degree of any vertex.

Contents

Preface

This thesis would never have been completed without the help of several people, and I am grateful to them all. First of all would like to thank my supervisor Christos Levcopoulos, who taught me how to do research using my knowledge and intuition. He has always encouraged and inspired me with his deep knowledge in computer science.

A special thank goes to Giri Narasimhan who motivated and encouraged me during his annual summer visits at our department.

The department provided a creative atmosphere and I will remember the lunch and coffee discussions with Anders, Andreas, Andrzej, 2×Anna, Bengt, Björn, Drago, Hans, Jesper, Klas, Kurt, Mikael, Per and Thore. I also like to express my appreciation to the algorithm group in Lund for providing a pleasant research environment. A special thanks goes to the Ph.D. students at our department who created a positive and scientific atmosphere while supporting each other in all weathers.

Numerous people have influenced my work. I would like to thank my coauthors during my Ph.D. studies Mikael Hammar, Herman Haverkort, Thore Husfeldt, Mark van Kreveld, Christos Levcopoulos, Giri Narasimhan and Mark Overmars. I wish to specifically thank Mark de Berg for inviting me to Utrecht and allowing me to collaborate with the dynamic research group in Utrecht. Also, I would like to thank Otfried Cheong for inviting me to Hong Kong.

Last, but definitely the most, I thank Anna Östlin for supporting me during this time.

This work was in part supported by the TFR projects dnr 221-1996-344 and dnr 221-1999-278.

Chapters 3-5 and 7-9 correspond each to an article. These articles are listed below in the order of the corresponding chapters.

- J. Gudmundsson and C. Levcopoulos, *Close approximations of minimum rectangular coverings*, Journal of Combinatorial Optimization 3 (4):437-452, December 1999.

- J. Gudmundsson and C. Levcopoulos, *A linear-time heuristic for minimum rectangular coverings.* In Proceedings of the 11th International Symposium on Fundamentals of Computation Theory, pages 305-316, LNCS 1279, 1997.

- J. Gudmundsson, T. Husfeldt and C. Levcopoulos, *Lower bounds for approximate polygon covering and minimum gap*, Technical report LU-CS-TR:2000-221, Department of Computer Science, Lund University, Sweden, 2000.

- J. Gudmundsson, C. Levcopoulos and G. Narasimhan, *Approximating minimum Manhattan networks.* In Proceedings of the 2nd International Workshop on Approximation Algorithms for Combinatorial Optimization Problems, pages 28-38, LNCS 1671, 1999.

- J. Gudmundsson, C. Levcopoulos and G. Narasimhan, *Improved greedy algorithms for constructing sparse geometric spanners*, In Proceedings of the 7th Scandinavian Workshop on Algorithm Theory, pages 314-327, LNCS 1851, 2000. Invited to SWAT'00 special issue of *Nordic Journal of Computing.*

- J. Gudmundsson, M. Hammar and M. van Kreveld, *Higher-order Delaunay triangulations*, In Proceedings of the 8th Annual European Symposium on Algorithms, pages 232-243, LNCS 1879, 2000.

Chapter 6 is based on two articles.

- J. Gudmundsson and C. Levcopoulos, *A fast approximation algorithm for TSP with neighborhoods*, Nordic Journal of Computing 6:469-488, 1999.

- J. Gudmundsson and C. Levcopoulos, *Hardness result on TSP with neighborhoods*, Technical report LU-CS-TR:2000-216, Department of Computer Science, Lund University, Sweden, 2000.

Chapter 1

Introduction

Twenty-five years ago Shamos and Hoey [88] introduced Voronoi diagrams to computer science. This was the start of systematic studies of geometric algorithms and the field computational geometry was born. Of course, Voronoi diagrams had been known long before, cf. Descartes's *Principia Philosophiae* 1644, but the notion of efficiently computing them was something new. As a consequence researchers started to identify geometrical problems of great importance. The first annual conference in the subject appeared in 1985. One year later the first journal in the field published its first issue. Now there are three journals devoted only to computational geometry and several annual conferences and workshops.

The success of the field can among other things be explained by the many applications of geometrical problems, for example, in computer graphics, chemistry, pattern recognition, geographic information systems, robotics, and other important application areas. Another reason for the success of the field is the beauty of geometry and the ease of understanding and explaining geometrical problems. In computational geometry classes, a large group of students is motivated and interested in the subject because the problems are often natural and occur in everyday life, for example shortest paths, Voronoi diagrams and point-location. When posing a geometrical problem to a student one doesn't need any complicated definitions or notations, with the result that the student naturally starts to work on the problem.

An obvious question is why we study geometrical problems explicitly and not their non-geometric counterpart. The answer is that a geometric input is more constrained than its non-geometric counterpart, which implies that the geometric problems are easier to handle and solve.

As an example, let us consider the Euclidean traveling salesman problem.

We are given a set of n points (cities) in the plane and we want to compute the shortest tour that visits all of the points exactly once. The geometric problem can be solved exactly in time $n^{O(\sqrt{n})}$, and approximated in polynomial time within a factor of $1+\epsilon$. While the general problem can be solved exactly in time $2^{O(n)}$, and cannot be approximated within any reasonable factor in polynomial time. But most significantly the constrained nature of the geometric problems gives us a lot of information that we may use to obtain efficient algorithms.

1.1 Geometric preliminaries

In this section we define some geometric objects that are used later on. Many standard definitions and notations are not included. Therefore, it is convenient to already have some knowledge in computational geometry. Examples of excellent introductory books on computational geometry are these of de Berg, van Kreveld, Overmars, and Schwarzkopf [32], O'Rourke [78] and, Preparata and Shamos [81].

Let $\mathbb{R}^d$ be the d-dimensional real space. A d-tuple $(x_1, \ldots, x_d)$ denotes a *point* of $\mathbb{R}^d$. We use $|xy|$ to denote the distance between points $x=(x_1, \ldots, x_d)$ and $y = (y_1, y_2, \ldots, y_d)$. In our case, distances are measured according to some *L_t-metric* $(1 \leq t \leq \infty)$,

$$|xy| = \left(\sum_{i=1}^{d} |x_i - y_i|^t \right)^{1/t}.$$

Unless otherwise stated, we use the L_2-metric (the standard Euclidean metric). A *straight-line segment* connecting two points x and y is the set

$$\overline{xy} = \{\alpha x + (1-\alpha)y : \alpha \in \mathbb{R} \text{ and } 0 \leq \alpha \leq 1\}.$$

Convex set. A domain D in $\mathbb{R}^d$ is said to be convex if, for any two points x and y in D, the segment $\overline{xy}$ lies entirely within D. The *convex hull* of a set of points in S in $\mathbb{R}^d$ is the boundary of the smallest convex set in $\mathbb{R}^d$ containing S.

Polygon. In $\mathbb{R}^2$ a polygon is defined by a finite set of segments such that every segment endpoint is shared by exactly two edges. The segments are the *edges* and their extremes are the *vertices* of the polygon. A *simple polygon* divides the plane into two parts, the *interior* (bounded) and the *exterior* (unbounded). By a polygon we often mean the union of its boundary and its interior. A *convex polygon* is a simple polygon such that all its vertices lie on their convex hull.

For simplicity we will throughout this text assume that no three points are co-linear and that no four points are co-circular.

1.2 Model of computation

From a practical point of view, the effectiveness of an algorithm can be measured in terms of how much time and memory is required by the algorithm. Both the running time and the space used by the algorithm depend on the particular computer and on the implementation. In order to compare different algorithms we need to agree on a standard model of computation. The model specifies the primitive operations that may be executed and their respective cost, space and time. The running time complexity of an algorithm is the sum of the primitive operations performed by the algorithm times their cost. Similarly, the space complexity describes how many memory units that are needed to store all the data required for the execution of the algorithm. The unit of space specifies what types of variables that can be used in the model, so-called elementary variables.

The model of computation that will be used by the algorithms presented in this thesis are designed for a model of computation called *real RAM* [81]. A real RAM can store an arbitrary real number in a memory location, and perform the following operations in unit time:

1. The arithmetic operations $(+, -, \times, /)$.
2. Comparisons between two real numbers $(<, \leq, =, \neq, \geq, >)$.
3. Indirect addressing of memory (integer addresses only).

And, optionally

4. Standard analytic functions on a real number, such as kth root, trigonometric functions, and logarithms.

The above model reflects the standard high-level languages like PASCAL, et cetera, in which many algorithms treat real variables as having infinite precision.

Some of our algorithms are shown to be optimal in the augmented decision tree model in which algebraic computations are allowed (excluding, for example, the non-algebraic floor function). This model is called *algebraic decision tree model* [81].

1.3 Notions of complexity

In this section we present standard complexity notions which can be found in most introductory books on algorithms and complexity. The reader already familiar to these concepts can jump to Section 1.4.

Each instance of a problem is specified by a set of data called the input to the problem. The size of the input is the number of memory units needed to represent the input. For geometrical problems, this is usually the number of

points and segments in the input. For most applications we are not primarily interested in the exact computation complexity of an algorithm for all possible values of input. Instead we are usually more interested in the behavior of these complexities as the input size grows. However, the running time may also depend on the input itself, for example, how points and segments are placed in space, how long the segments are and so on. In order to specify the performance of an algorithm on input of size n, one determines the complexity by running the algorithm on the worst case instance of size n. The worst case instance is the instance with highest complexity. More formally:

An algorithm, denoted A, performs a sequence of operations on its input. If I denotes the input, we let $t_A(I)$ denote the time algorithm A needs to compute the final result, with I as input. We can associate a *time complexity function* $T_A(n) : \mathbb{N} \rightarrow \mathbb{N}$ to every algorithm A, defined as

$$T_A(n) = \max_{|I|=n} \{t_A(I)\}.$$

As described above, this is the worst-case time complexity over all possible input instances of size n. Similarly, one can associate a *space complexity function* $S_A(N)$ to every algorithm. The worst-case complexity is a pessimistic estimation of the complexity of an algorithm. For many cases in computational geometry the upper bound is only reached for extreme inputs that very seldom occur if at all.

We are mainly interested in how fast the computation complexity grows with respect to the input size. In algorithmic analysis one usually expresses the complexity, space and time, as a function of the input size and removing the multiplicative constant.

Knuth [55] introduced a notation that distinguishes between upper and lower bounds, which we will use in this thesis.

$O(f(N))$ denotes the set of all functions $g(N)$ such that there exist positive constants C and N_0 with $g(N) \leq C \cdot f(N)$ for all $N \geq N_0$.

$\Omega(f(N))$ denotes the set of all functions $g(N)$ such that there exist positive constants C and N_0 with $g(N) \geq C \cdot f(N)$ for all $N \geq N_0$.

$\Theta(f(N))$ denotes the set of all functions $g(N)$ such that there exist positive constants C_1, C_2 and N_0 with $C_1 \cdot f(N) \leq g(N) \leq C_2 \cdot f(N)$ for all $N \geq N_0$.

We say that an algorithm A has polynomial time-complexity if $T_A(n) = O(n^c)$, where c is some constant. Algorithms that have polynomial time complexity are said to be efficient, so when we try to obtain efficient algorithms for problems we look for algorithms with polynomial time-complexity. Unfortunately, for many problems no polynomial time algorithms have been found. One

class of problems that has not been found to admit any efficient algorithms, is known as the class of NP-complete problems. It is commonly believed that no efficient algorithms even exist for this class of problems, but no one has, so far, been able to prove it.

Many problems of practical significance are NP-complete but are too important to abandon. If a problem is NP-complete, we are unlikely to find a polynomial-time algorithm solving the problem exactly. But, it may still be possible to find a near-optimal solution in polynomial time. In practice, near optimal might be good enough. An algorithm that returns a near-optimal solution is called an *approximation algorithm*.

Assume that we have an optimization problem where each possible solution has a cost, and we want to find a near-optimal solution. An approximation algorithm A is said to be an $\alpha(n)$-approximation algorithm if for any input of size n, the cost C_A of the solution produced by A is within a factor of $\alpha(n)$ of the optimal cost C_{opt}:

$$\alpha(n) \geq \max(\frac{C_A}{C_{opt}}, \frac{C_{opt}}{C_A}).$$

Note that this definition holds for both minimization and maximization problems.

An approximation scheme for an optimization problem is an approximation algorithm that takes as input not only an instance of the problem, but also a value $\epsilon > 0$ such that for any fixed ϵ, the scheme is an $(1 + \epsilon)$-approximation algorithm.

The reader seeking a more detailed discussion on the notions of complexity in this chapter is referred to one of the classical textbook on algorithm such as those by Aho, Hopcroft, and Ullman [4], and Cormen, Leiserson, and Rivest [27].

1.4 Outline of the thesis

In this thesis we focus on four problems: covering polygons with simpler polygons, finding a minimum tour visiting a set of regions, constructing sparse spanners and triangulating a set of points. We present several approximation algorithms for these problems and also new lower bounds.

Covering problems. The following five chapters of this thesis consider the problem of covering arbitrary polygons with a minimum number of rectangles or squares. The problem of covering polygons with various types of simpler polygons has a number of important practical applications [46, 53] and has received considerable attention from a theoretical perspective. One application for this problem is the fabrication of VLSI chips. According to [46], a common method

for fabricating VLSI chips is the optical method of the automatic blockflasher, which exposes rectangles of, in practice, almost any size and any orientation. In order to minimize the cost of the fabrication it is desirable to cover the polygonal area of each layer of the circuit with as few rectangles as possible. Also when solving other problems for geometric figures, a common method is to decompose the figure into simpler parts, solve the problem on each component using some specialized algorithm, and then combine the partial solutions to solve the problem for increasingly larger parts of the polygon.

In Chapter 3 we present the first polynomial time $O(\log n)$-approximation algorithm under the assumption that the vertices of the polygon are given as polynomially bounded integer coordinates. The idea is to reduce the original problem to the set-covering problem, by constructing a superset of rectangles. Using the same approach we can also find a covering which is within a constant factor of the optimal in exponential time.

In Chapter 4 we show that in the case when the input polygons are hole-free one can in linear time obtain a much better approximation factor. The approximation ratio is $O(\alpha(n))$ where $\alpha(n)$ is the extremely slowly growing inverse of Ackermann's function. The idea of this result is very simple. Our claim is that for hole-free polygons there cannot be a large set of "long" rectangles that covers a major part of the polygon. Hence, one can replace these "long" rectangles by short rectangles, which implies that a good algorithm that covers the polygon locally performs quite good.

In the final chapter on the covering problem we notice that there are several covering algorithms that run in time $O(n \log n)$ for general polygons but can be improved to have linear time-complexity in the case when the input polygon is hole-free. This led us to believe that the trivial linear lower bound can be improved in the case when the given input polygon contains holes. We show that any approximation algorithm for covering an arbitrary polygon with holes, with a finite number of polygons, has a lower bound on the time-complexity of $\Omega(n \log n)$ in the algebraic decision tree model.

Traveling salesman problem with neighborhoods (TSPN). The second problem we consider is a variant of the well-known Euclidean traveling salesman problem. A salesman wants to meet some potential buyers. Each buyer specifies a region in the plane, his *neighborhood*, within which he is willing to meet. The salesman wants to find a tour of shortest length that visits all of the buyers' neighborhoods and finally returns to his initial departure point. The number of simple polygons, called *neighborhoods*, is denoted k and the total number of vertices in the plane is denoted n.

The main result, of Chapter 6, is an algorithm that, given an arbitrary real

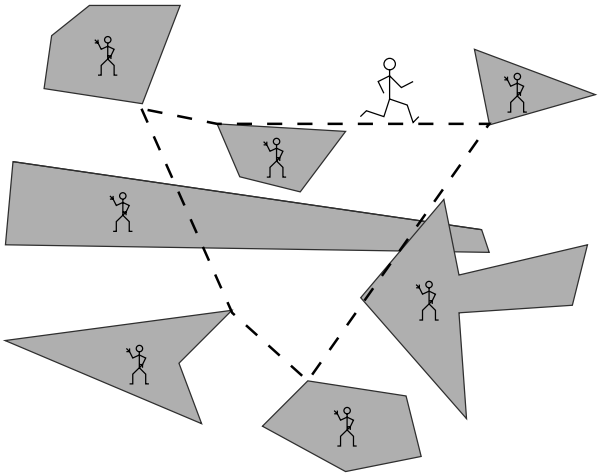

Figure 1.1: An example of TSP with neighborhoods.

constant ϵ as an optional parameter, performs at least one of the following two tasks (depending on the instance):

(1) It outputs in time $O(n \log n)$ a TSPN tour guaranteeing that it is of length $O(\log k)$ times the optimum.

(2) It outputs a TSPN tour guaranteeing that it is of length less than $(1+\epsilon)$ times the optimum in time $O(n^3)$.

The first part of our method builds upon an idea by Mata and Mitchell [70], in that our logarithmic approximation algorithm produces a guillotine subdivision. However, we produce a quite different guillotine partition and show that it has some nice "sparseness" properties, which guarantee the $O(\log k)$ approximation bound. The described method can also be applied to other problems as suggested in [70]. In the same chapter we also give the first hardness result for TSPN. We show that TSPN is APX-hard and cannot be approximated within a factor of 1.000374 unless P=NP. The reduction is done using the well-known Min Vertex Cover-problem, which is known to be APX-hard and not approximable within a factor of 1.0029 in the case when the degree is bounded by 5 [16].

Constructing spanners. The third problem we consider is how to construct sparse spanners. The problem of constructing spanners has been thoroughly investigated by many researchers the last decade [11, 20, 31, 52, 85, 93]. Spanners have applications in the design of geometric networks and they are also useful in designing efficient approximation schemes for geometric problems. Consider

a set S of n points in $\mathbb{R}^d$, where the dimension d is a constant. A geometric network on S can be modeled as an undirected graph G with vertex set S and with edges $e = (a, b)$ of weight $wt(e)$ defined as the Euclidean distance $d(a, b)$ between its two endpoints a and b. Let p and q be two points of S, and let P be a pq-path in G, i.e., a path in G between p and q. The weight of the path P is denoted by $wt(P)$ and is defined as the sum of the weights of the edges of P. Let $t > 1$ be a real number. We say that G is a *t-spanner* for S, if for each pair of points $p, q \in S$, there exists a pq-path in G of weight at most t times the Euclidean distance between p and q. A sparse spanner is a spanner with low weight and a linear number of edges.

In Chapter 7 we consider the problem of finding a minimum Manhattan network. For a given set S of points in the plane, we define a *Manhattan Network* on S as a rectilinear geometric network G with the property that for every pair of points $p, q \in S$, the network G contains a minimum rectilinear pq-path connecting them. A *Minimum Manhattan Network* on S is a Manhattan network of minimum possible length. Many VLSI circuit design applications require that a given set of terminals in the plane be connected by networks of small total length. Rectilinear Steiner minimum trees were studied in this context. Manhattan networks impose additional constraints on the distance between the terminals in the network. The concept of Manhattan networks seems to be a very natural concept; it is surprising that this concept has not been previously studied. Manhattan networks are also closely connected to the concept of spanners. In this connection, a minimum Manhattan network can be thought of as a sparsest 1-spanner for S for the L_1-norm, assuming that Steiner points are allowed to be added. 1-spanners are also interesting since they represent the network with the most stringent distance constraints.

In Chapter 8, we show the first algorithm that constructs a sparse spanner in time $O(n \log n)$. This solves a critical open problem, since in a startling development, Rao and Smith [83] showed an optimal $O(n \log n)$-time approximation scheme for the well-known Euclidean *traveling salesperson problem*, assuming that it is possible to compute sparse spanners in time $O(n \log n)$. Our algorithm is inspired by the algorithm due to Das and Narasimhan [31]. They showed how to use clustering in order to speed up shortest path queries. However, their algorithm was not efficient enough because they were unable to maintain the clusters efficiently and the algorithm had to frequently rebuild the clusters. We retain the general framework of that algorithm. Our main contribution is in developing techniques to efficiently perform clustering. We believe that the techniques that we have developed are likely to be useful in designing other greedy-style dynamic algorithms, i.e., in situations where only insertions take place and particularly in increasing order of length.

Figure 1.2: Example of a t-spanner of the complete Euclidean graph with $t = 1.18$.

Higher-order Delaunay triangulations. One of the most well-known and useful structures studied in computational geometry is the Delaunay triangulation [32, 38, 75]. It has applications in spatial interpolation between points with measurements, because it defines a piecewise linear interpolation function. One specific use of the Delaunay triangulation for interpolation is to model elevation in Geographic Information Systems. The so-called *Triangulated Irregular Network,* or *TIN,* is one of the most common ways to model elevation. Elevation is used for hydrological and geomorphological studies, for site planning, for visibility impact studies, for natural hazard modeling, and more.

Because a TIN is a piecewise linear, continuous function which is generally not differentiable at the edges, these edges play a special role. In elevation modeling, one usually tries to make the edges of the TIN coincide with the ridges and valleys of the terrain. Then the rivers that can be predicted from the elevation model are a subset of the edges of the TIN. When one obtains a TIN using the Delaunay triangulation of a set of points, the ridges and valleys in the actual terrain will not always be as they appear in the TIN. The so-called 'artificial dam' in valleys is a well-known artifact in elevation models, Fig. 1.3. It appears when a Delaunay edge crosses a valley from the one hillside to the other hillside, creating a local minimum in the terrain model slightly higher up in the valley. It is known that in real terrains such local minima are quite rare [48]. These artifacts need to be repaired, if the TIN is to be used for realistic terrain modeling [86], in particular for hydrological purposes [67, 68, 91]. If the valley and ridge lines are known, these can be incorporated by using the constrained Delaunay triangulation [26, 36, 66]. The cause of problems like the one mentioned above may be that the Delaunay triangulation is a structure defined for a planar set of points, and doesn't take into account the third dimension at all. One would like to define a triangulation that is both well-shaped and has some other properties as well, like avoiding artificial dams. This led us to define *higher-order Delaunay (HOD) triangulations,* a class of triangulations for any point set P that allows some flexibility in which triangles are actually used. The Delaunay triangulation of P has the property that for each triangle, the circle

through its vertices has no points of P inside. A k-order Delaunay (k-OD) triangulation has the relaxed property that at most k points are inside the circle. The idea is then to develop algorithms that compute some HOD triangulation that optimizes some other criterion as well. Such criteria might be minimizing the number of local minima, and minimizing the number of local extrema. The former criterion deals with the artificial dam problem, and the latter criterion also deals with interrupted ridge lines. For finite element method applications, criteria like minimizing the maximum angle, area triangle, and degree of any vertex may be of use [17, 18, 19].

We define HOD triangulations and analyse their properties. We also give an algorithm to compute which edges can be included in a k-OD triangulation. The algorithm runs in $O(nk \log n + n \log^3 n)$ expected time. For 1-OD triangulations we show some useful results, i.e., that most of the criteria we study can be optimized in $O(n \log n)$ time.

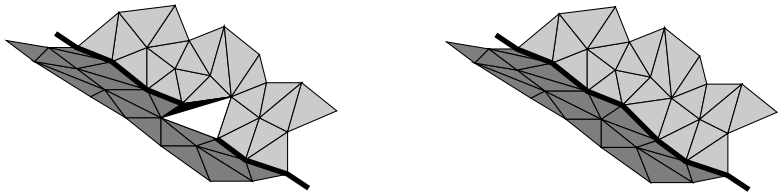

Figure 1.3: Artificial dam that interrupts a valley line (left), and a correct version obtained after one flip (right).

Chapter 2

Covering polygons

The problem of covering polygons with various types of simpler polygons has a number of important practical applications [46, 53] and has received considerable attention from a theoretical perspective. One application for this problem is the fabrication of VLSI chips. According to Hegedüs [46], a common method for fabricating VLSI chips is the optical method of the automatic blockflasher, which exposes rectangles of, in practice, almost any size and any orientation. In order to minimize the cost of the fabrication it is desirable to cover the polygonal area of each layer of the circuit with as few rectangles as possible. For more information about the VLSI fabrication process see [46]. Also when solving other problems for geometric figures, a common method is to decompose the figure into simpler parts, solve the problem on each component using some specialized algorithm, and then combine the partial solutions to solve the problem for increasingly larger parts of the polygon.

O'Rourke and Supowit [79] showed that the problems of covering polygons with a minimum number of convex polygons, star-shaped polygons or spiral polygons are all NP-hard, if the polygon contains holes. The rectilinear case, i.e., when the polygons, as well as the rectangles, have sides that are vertical or horizontal, has been treated in several papers [13, 39, 53, 56, 73, 87, 95]. Masek [69] proved that this case is NP-complete, but again the proof required the polygon to contain holes. Later Culberson and Reckhow [28] showed this case to be NP-hard even when the given polygon was hole-free. Berman and DasGupta [15] showed that if the rectilinear polygon has holes, the problem is also MAXSNP-hard.

Kumar and Ramesh [56] recently presented a polynomial-time approximation algorithm that produces a covering which is within an $O(\sqrt{\log n})$ factor of the optimal. When the polygon is hole-free, Franzblau [40] showed a factor 2

approximation guarantee. When the polygon is both vertically and horizontally convex, Chaiken, Kleitman, Saks and Shearer [21] gave a polynomial time algorithm for computing the minimum number of rectangles required, exactly. This was improved upon by Franzblau and Kleitman [39], who achieved the same result under the weaker restriction that the hole-free polygon is just vertically convex.

For the related *rectilinear square covering* problem some of the previous work used a bit-map representation for the input polygon [5, 12, 73, 87]. A *bit-map* representation of a polygon is a boolean (zero-one) matrix, where one (1) represents a point inside the polygon and a zero (0) - a point outside it. When using this representation, complexity is measured in terms of the number of points in the matrix, denoted p. Note that $p{>}\Omega(n)$, and for most practical applications $p{\gg}n$, where n is the number of edges of P. Bar-Yehuda and Ben-Hanoch argues that the representation of the polygon should be made as segment representation, not only theoretically, but also for most practical applications [13].

Scott and Iyenger [87] present an algorithm to find, in $O(n \log n)$ time, the maximal squares in the polygon, and then divide the rectangular portions of the polygon to cover them minimally. However, their algorithm does not yield a globally minimum cover. It was shown by Aupperle, Conn, Keil and O'Rourke [12] that the rectilinear case is NP-hard for polygons containing holes. In the case where the image is hole-free, they provide an $O(p^{1.5})$ algorithm. Recently, Bar-Yehuda and Ben-Hanoch [13] presented a linear time algorithm for covering simple rectilinear polygons with squares. Morita [73] developed a parallel algorithm, which finds a *minimal* (not minimum[1]) square cover for bit-maps which may contain holes. The sequential running time of this algorithm is $O(p)$. Gudmundsson and Levcopoulos [44] presented an approximation algorithm for the general, non-rectilinear, square covering problem that guaranteed a constant approximation factor 14 running in time $O(n^2 + \mu(P))$, where $\mu(P)$ is the number of squares in an optimal covering.

In the general rectangle covering problem it is not known whether the optimal solution can be computed even in exponential time. Using the proof technique from [77] (see also [24] and [72]), Tarski's decidability results can be applied to prove that the decision version of the problem is computable, but no upper bound better than $2^{2^{O(n)}}$ was known for its execution time. Hegedüs [46] implemented a program (the so-called GENCOV-algorithm) which covers general polygons with rectangles and presented some empirical results concerning its performance. The results show that the algorithm is rather good in practice, but no theoretical analysis is presented in [46]. Another approximation

[1] A square cover is called *minimal* if it has no smaller subset that forms a cover. A square cover is called *minimum* if there is no smaller set that forms a cover.

algorithm was presented by Levcopoulos in [58], and with refined analysis in [60], which produces $O(n \log n + \mu(P))$ rectangles to cover an arbitrary coverable polygon P in time $O(n \log n + \mu(P))$, where n is the number of vertices of P and $\mu(P)$ is the minimum number of rectangles required to cover P. (A polygon can be covered by rectangles iff it has no acute interior angles.) In [60] it was shown that $\mu(P) = O(n \log \frac{\min(b,l)}{s})$, where s is the shortest distance between two non-incident edges of P, l is the length of the longest edge of P and b is the diameter of the largest circle which can fit within P. It was also shown that this upper bound cannot be further refined with respect to n, b, l and s.

In the next three chapters we show results concerning the problem of covering a general polygon with rectangles.

A problem with the algorithm by Levcopoulos [60] is that, even when $\mu(P) = O(n^{0.5+\epsilon})$, where ϵ is any fixed constant greater than zero, the algorithm may produce a covering with $\Omega(n \log n)$ rectangles, which is very far from being optimal, as is shown in [60]. To get a reasonable approximation in the case when $\mu(P) \ll n$ we use a different approach in Chapter 3. The idea is to reduce the original problem to the set-covering problem. By constructing a superset of rectangles, we can find a covering which is within a constant factor of the optimal in exponential time. We also present the first polynomial time algorithm, guaranteeing an $O(\log n)$ approximation factor, provided that the n vertices of the polygon are given as polynomially bounded integer coordinates.

In Chapter 4 we establish that, for hole-free polygons, the algorithm presented by Levcopoulos in [58] produces $O(min[n + \mu(P), \alpha(n) \cdot \mu(P)])$ rectangles, where $\alpha(n)$ is the extremely slowly growing inverse of Ackermann's function. For proving this result we develop new techniques which we believe are interesting themselves, and can be used, e.g., for showing properties of other types of coverings.

In the final chapter, on the problem of covering polygons, we show that any approximation algorithm for covering an arbitrary polygon, with holes, with a finite number of polygons has a lower bound on the time-complexity of $\Omega(n \log n)$ in the algebraic decision tree model.

Chapter 3

Close approximations of minimum rectangular coverings

A problem with the algorithm by Levcopoulos [60], as mentioned in Chapter 2, is that, even when $\mu(P) = O(n^{0.5+\epsilon})$, where ϵ is any fixed constant greater than zero, the algorithm may produce a covering with $\Omega(n \log n)$ rectangles, which is very far from being optimal, as is shown in [60]. To get a reasonable approximation in the case when $\mu(P) \ll n$ we need a different approach. In order to find a better covering in the rectilinear case one has used the approach of finding a superset of rectangles which includes the rectangles in an optimal covering. If we could find such a set for the general case we could easily translate the polygon-covering problem to the set-covering problem. The main difference between the rectilinear case and the general case is that there are only $O(n^2)$ possible maximal rectangles in the rectilinear case versus infinite many possible maximal rectangles in the general case. Instead we will in this paper construct a set of rectangles, denoted C_R, and prove that C_R includes a covering which is within a constant factor of the optimal. In fact we show that C_R possesses an even stronger property, namely that every possible rectangle within a polygon P can be covered by a constant number of rectangles in C_R (Lemmas 3.5-3.7).

By reducing the original problem to the set-covering problem, by rectangles from C_R, we can find a covering which is within a constant factor of the optimal in exponential time. We also present the first polynomial time algorithm, guaranteeing an $O(\log n)$ approximation factor, provided that the n vertices of the polygon are given as polynomially bounded integer coordinates.

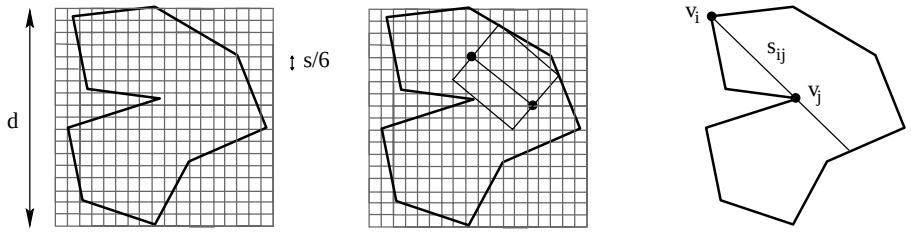

Figure 3.1: (a) The polygon P within the grid. (b) Two of the rectangles in C_R with endpoints at two marked points in C_p. (c) The maximal segment within P that passes through, or ends at, the two vertices v_i and v_j.

3.1 Defining the pool of rectangles

Let P be an arbitrary polygon, possibly with polygonal holes. The shortest distance between two edges, e_1 and e_2, is defined as the shortest distance between any two points on e_1 and e_2. Let $s(P)$ be the minimum of the length of the shortest edge of P and the shortest distance within P between two non-incident edges. The largest distance between two points in P is denoted $d(P)$. When it is clear from the context which polygon we refer to, we shall simply write s and d instead of $s(P)$ and $d(P)$. Construct a $(\frac{6d}{s} \times \frac{6d}{s})$ grid such that every square in the grid has length and height $s/6$, and P lies entirely within the grid, as shown in Fig. 3.1a. We can now bound the number of vertices with respect to s and d.

Lemma 3.1 *Let P be a polygon of n vertices. It holds that $n{=}O((\frac{d}{s})^2)$.*

Proof: Let P be a polygon with n vertices. By the definition of d we have that the area of P is less than $\frac{\pi \cdot d^2}{4}$. We know that the sum of all the interior angles of P is $180° \cdot (n{-}2)$. Construct for every vertex v_i a region $r(v_i)$, such that every point in P closer than $s/2$ to v_i belongs to $r(v_i)$. According to the definition of s, the shortest distance between two vertices of P is at least s, thus the regions are disjoint. The total area covered by these regions is $(n{-}2) \cdot \frac{\pi \cdot s^2}{8}$, which is less than the area of P. Thus we have

$$(n{-}2)\frac{\pi \cdot s^2}{8} < \frac{\pi \cdot d^2}{4}, \quad \text{hence} \quad n < 2 \cdot \left(\frac{d}{s}\right)^2 + 2.$$

□

We are now ready to construct the pool of rectangles. First a point set C_P is created which will be used to construct a subset of C_R. Let C_P be a set of points, such that:

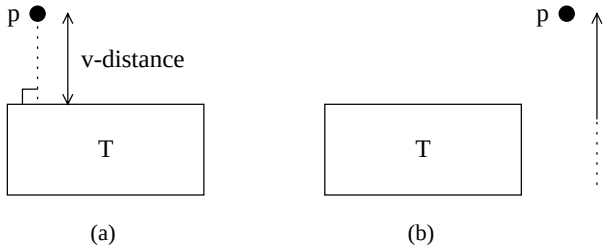

Figure 3.2: If p has a perpendicular projection on one of T's long sides (a) then the v-distance is the shortest distance between p and T, otherwise (b) it is infinite.

1. every vertex of P is a point in C_P,
2. every crossing between P's perimeter and the grid is a point in C_P, and
3. every crossing in the grid is a point in C_P.

The number of points in C_P is $O((d/s)^3)$, according to Lemma 3.1. We construct a set of rectangles, C_R, by first inserting into C_R each maximal rectangle which lies within P and which has a side with both endpoints in C_P, Fig. 3.1b. These rectangles are denoted Class I rectangles or C_{I}-rectangles. The number of rectangles in C_R is now $O((d/s)^6)$.

We will need to extend the number of rectangles in C_R, with two sets of rectangles. First we add the following set.

For every pair of vertices v_i, v_j of P, where v_i is visible from v_j, let s_{ij} be the maximal segment, within P, that passes through or ends at v_i and v_j, as shown in Fig. 3.1c. For every segment s_{ij} let S_{ij} be a set of points that contains the set of intersection points between s_{ij} and the grid segments, plus the two endpoints of s_{ij}, v_i and v_j. The number of points in S_{ij} is $O(d/s)$. For every possible pair of points $p_l, p_k \in S_{ij}$ insert into C_R all maximal rectangles within P which have a side with endpoints in p_l and p_k (there are 0,1 or 2 such rectangles). These rectangles are denoted C_{II}-rectangles and the total number of C_{II}-rectangles inserted in this step is at most $n^2 \cdot (d/s)^2 = O((d/s)^6)$.

The final set of rectangles added to C_R is needed to cover the rectangles in Lemma 3.6, *Case d*. The rest of this section is devoted to the construction of these special rectangles, denoted C_{III}-rectangles. We will need the following definitions:

Definition 3.2 The shortest distance between a rectangle T, within P, and a point p, with a perpendicular projection on one of T's long sides (or any side if T is a square), is denoted the *v-distance* between T and p. If p does not have a perpendicular projection on T's long sides then the v-distance is infinite.

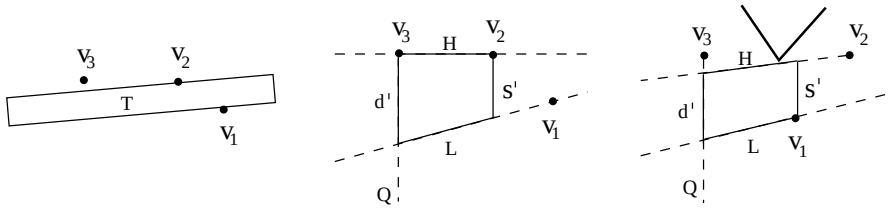

Figure 3.3: (a) A valid triplet. (b) and (c) Two $3v$-trapezoids.

The v-distance between a rectangle T, within a polygon P, and P's perimeter is the shortest v-distance between T and a point on P's perimeter.

Definition 3.3 Let (v_1, v_2, v_3) be a triplet of vertices. Rotate P such that v_2 and v_3 have the same y-coordinate, and v_3 lies to the left of v_2. For simplicity we assume that v_1 lies below v_2 and v_3. The triplet (v_1, v_2, v_3) is said to be *valid* if and only if the following conditions hold.

- There exists a rectangle T with long sides of length at least $4s$ and short sides of length at most $5s/6$ such that v_1 and v_2 touch opposite long sides of T, and the v-distance between v_3 and T is at most $s/6$.
- v_3 lies to the left of v_1, see Fig. 3.3a.

For simplicity we will assume that v_3 lies above T, i.e., v_2 and v_3 lie on the same side of T, as shown in Fig. 3.3. Assume that (v_1, v_2, v_3) is a valid triplet. Let Q be the vertical segment of length s with upper endpoint at v_3. Let H be a segment within P between v_2 and Q, such that the intersection between Q and H is as close as possible to v_3, and let L be a segment within P between v_1 and Q such that the intersection between Q and L has the smallest possible y-coordinate (as far from v_3 as possible), Fig. 3.3.

Let s' be the longest of the two vertical segments between v_2 and L (or its extension), respectively v_1 and H (or its extension), and let d' be the segment on Q between H and L. If the length of s' is shorter than the length of d' then the trapezoid described by the two vertical segments s' and d', and the segments of H and L between s' and d', is denoted a *3v-trapezoid*. Note that a $3v$-trapezoid is hole-free since the shortest diagonal of a hole in P is at least s and $|d'| \leq s$.

For every $3v$-trapezoid t in P, construct at most $\lceil |d'|/|s'| \rceil$ rectangles with short-sides of length $|s'|$ where every rectangle has one endpoint on s''s lower endpoint and one on d'. We place the rectangles in such a way that a rectangle's

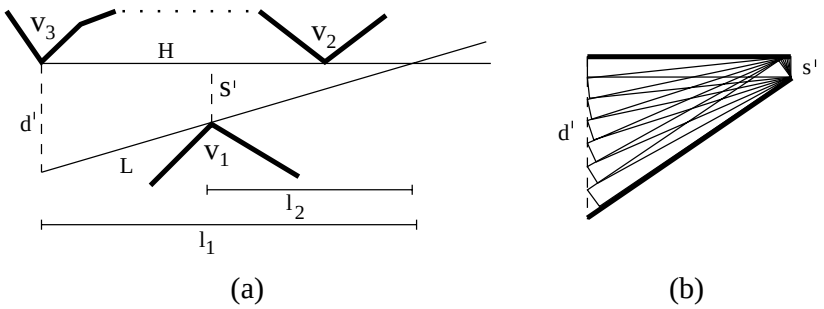

Figure 3.4: (a) Finding a $3v$-trapezoid. (b) A $3v$-trapezoid partially covered by C_{III}-rectangles.

short side, closest to d', touches both d' and the upper long-side of the rectangle below, Fig. 3.4b. The rectangles will at least cover the part of t at distance more than $s/2$ from s' and d', see Fig. 3.4b. Add these rectangles to C_R. These rectangles are denoted C_{III}-rectangles and can always be constructed since a $3v$-trapezoid is hole-free. Next we bound the number of C_{III}-rectangles in C_R.

Observation 3.4 *For any $3v$-trapezoid it holds that $\frac{|d'|}{|s'|} < 2\frac{d}{s}$.*

Proof: Rotate the $3v$-trapezoid in such a way that s' and d' are vertical. Let l_1 be the horizontal distance between d' and the intersection of H's and L's extensions, and let l_2 be the horizontal distance between s' and the intersection of H's and L's extensions, Fig. 3.4a. Note that $l_2 > s/2$ since the shortest distance between v_1 and v_2 is s and the vertical distance is less than $5s/6$. So for every $3v$-trapezoid in P we will construct at most $|d'|/|s'|$ rectangles, and by simple trigonometry we have: $|d'|/|s'| = (l_1/l_2) < (d/l_2) < 2 \cdot d/s$. □

Since P consists of n vertices there can be at most $O(n^3)$ $3v$-trapezoids in P, and since $n = O((d/s)^2)$, according to Lemma 3.1, we will add at most $n^3 \cdot 2 \cdot (d/s) = O((d/s)^7)$ C_{III}-rectangles to C_R. The total number of rectangles in C_R will hence be $O((d/s)^7)$. It is easily seen that C_R can be constructed in time linear with respect to the number of rectangles in C_R.

3.2 Covering rectangles

In Lemmas 3.5-3.7 we will show that every rectangle T within a polygon P can be covered by a constant number of rectangles in C_R. To be able to guarantee that we handle all possible cases, we divide the rectangles into three cases according to their shape, Fig 3.5. The three cases are:

1. Rectangles whose longest side is shorter than $s/2$. Here we will have two subcases depending on whether the rectangle lies close to the perimeter of P or not.

2. Rectangles whose shortest side is shorter than $s/2$. Here we will have a total of four subcases depending on the distance between the long sides of the rectangle and the perimeter of P. The most complicated case of all the cases in Lemmas 3.5-3.7 is when we have a thin rectangle whose both long sides lie close to the perimeter of P. The C_{III}-rectangles are specially designed in order to handle this case.

3. Rectangles whose shortest side is longer than $s/2$. By using the results from case 1 and 2 it is easy to prove that "large" rectangles can be covered by a constant number of rectangles from C_R.

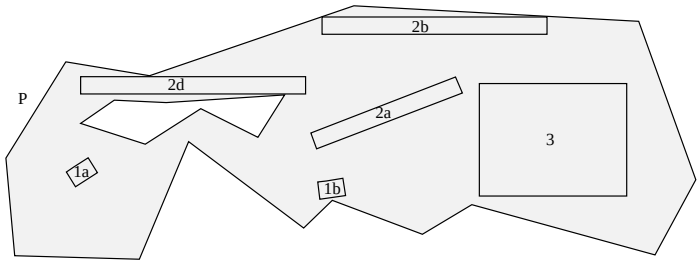

Figure 3.5: An example showing some of the cases that we have to handle.

In order to facilitate an easier description in the continuation of the text, we assume w.l.o.g. that the two longer sides of a rectangle T, within P, lie horizontally, that is, we rotate T, P and the grid until T's long sides lie horizontally. Let A, B, C and D be the corners of T in a counter-clockwise order, where A and B are the endpoints of T's lowest long-side, as shown in Fig. 3.8. Thus the segments $\overline{AB}$ and $\overline{CD}$ are at least as long as $\overline{AD}$ and $\overline{BC}$.

Lemma 3.5 *Every rectangle T, within P, whose longest side is shorter than $s/2$ can be covered by a constant number of rectangles in C_R.*

Proof: We have two cases:

Case a: The shortest distance between T and P's perimeter is at least $s/\sqrt{18}$.

The distance between T and P's perimeter is at least as long as the diagonal of a square in the grid, so T will never lie within the same square as P's perimeter. Since the diagonal of T is at most $1/\sqrt{2}s$, T can at most overlap six rows or columns of squares in the grid. And since every side of a square is the base of a

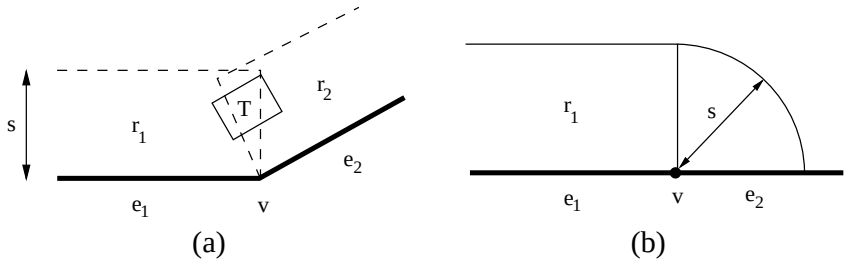

Figure 3.6: (a) T is covered by r_1 and r_2. (b) The quadrant bounds the area which T may overlap.

C_{I}-rectangle in C_R, the rectangle T can be covered by at most six C_{I}-rectangles in C_R.

Case b: The rectangle T lies closer than $s/\sqrt{18}$ from P's perimeter.

Let e_1 be an edge of P closest to T. Let r_1 be the C_{I}-rectangle in C_R with base e_1. Since T lies closer than $s/\sqrt{18}$ from e_1, the largest distance between a point of T and e_1 is $\frac{s}{\sqrt{18}} + \frac{s}{\sqrt{2}} = \frac{\sqrt{8}}{3}s < s$. The height of r_1 is at least s. That is, the part of T with a perpendicular projection on e_1 is covered by r_1. If r_1 covers T then we are finished, otherwise let e_2 be the edge incident to e_1 that lies on the same side of e_1 as the part of T that is not covered by r_1, Fig. 3.6a. The vertex connecting e_1 and e_2 is denoted v. It remains to show that there exists a rectangle r_2 in C_R that covers the rest of T. Two cases can occur:

(*i*) The interior angle (e_1, e_2) is less than or equal to 180 degrees.

Since every point in T lies within distance less than s from e_1, the upper right quadrant of a circle with center at v and radius s bounds the area outside r_1 where T partly can overlap, as shown in Fig. 3.6b. The C_{I}-rectangle r_2 with base e_2 covers the entire area described by the quadrant, so the remaining part of T that is not covered by r_1 is covered by r_2.

(*ii*) If the interior angle (e_1, e_2) is greater than 180 degrees then partition T into nine rectangles, $T_1, \ldots, T_9$ of equal size, such that the longest side of

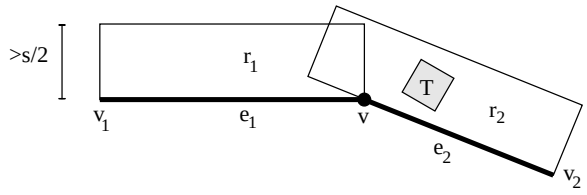

Figure 3.7: r_1 and r_2 cover every small rectangle that lies closer than $s/\sqrt{18}$ from P's perimeter.

each subrectangle T_i, $1 \leq i \leq 9$, is at most of length $s/6$. Now we can cover every subrectangle T_i separately. Every subrectangle that lies at least $s/\sqrt{18}$ from P's perimeter is covered according to *Case a*. Let T' be any of the subrectangles that lies closer than $s/\sqrt{18}$ from P's perimeter. Let v_1 (v_2) and v be the vertices of e_1 (e_2). Recall that on every edge and its extensions there exist $O((d/s)^2)$ C_{II}-rectangles in C_R. Let r_1 (r_2) be the maximal C_{II}-rectangle in C_R with base on e_1 (e_2) and its extension, such that one corner lies at v_1 (v_2) and one corner on the extension of e_1 (e_2) between $s/2$ and $(s/2+s/\sqrt{18})$ from v, Fig. 3.7. Since the shortest distance from v to a non-incident edge is s, it follows that the height of r_1 and r_2 is at least $s/2$. Also, since the length of T''s diagonal is at most $s/\sqrt{18}$, we have that a point in T' lies within distance $(s/\sqrt{18}+s/\sqrt{18}) < s/2$ from e_1. Thus r_1 and r_2 covers T', and T can be covered by a constant number of rectangles in C_R. □

When the rectangles are "long and thin" we need to cover the rectangles in different ways depending on the v-distance between the rectangle and P's perimeter.

Lemma 3.6 *Every rectangle T, within P, whose shortest side is at most of length $s/2$, can be covered by a constant number of rectangles in C_R.*

Proof: If $|AB| \leq 4s$ then partition T into eight equal rectangles, whose long sides are of length at most $s/2$, which are then covered according to Lemma 3.5. Otherwise insert four points A', B', C' and D' on T's perimeter, where A' and B' are points on $\overline{AB}$ at a distance s from A, respectively B, and C' and D' are points on $\overline{CD}$ at a distance s from C, respectively D. Partition T into three rectangles by inserting two segments $\overline{A'D'}$ and $\overline{B'C'}$ in T. We denote these partitions, from left to right, T_l, T_m and T_r, see Fig. 3.8. Partition T_l, respectively T_r, into two equal rectangles with long sides of length $s/2$ and cover these according to Lemma 3.5. It remains to cover T_m. We distinguish four cases:

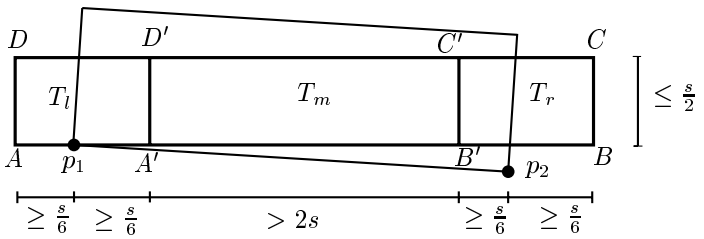

Figure 3.8: T partitioned into subrectangles and, the rectangle with base $\overline{p_1p_2}$ covers T_m.

Case a: The v-distance between T and P's perimeter is at least $s/6$.

We claim that there exists a C_{I}-rectangle r in C_R that covers T_m. To prove this claim, let p_1 be the point in C_P with largest y-coordinate below or on $\overline{AA'}$, such that the horizontal distance from p_1 to the extension of $\overline{DA}$ is between $s/6$ and $5s/6$, and symmetrically, let p_2 be the point in C_P with largest y-coordinate below or on $\overline{BB'}$, such that the horizontal distance from p_2 to the extension of $\overline{CB}$ is between $s/6$ and $5s/6$, Fig. 3.8.

Let r be the C_{I}-rectangle in C_R with base $\overline{p_1p_2}$. Since the vertical difference between p_1 and p_2 is less than $s/6$ and since $|p_1p_2| > s$, it follows that the slope of r's left and right side is greater than six and, hence r covers T_m.

Case b: The v-distance between one of T's long sides and P's perimeter is less than $s/6$, and the v-distance between the opposite long side and P's perimeter is at least $s/6$.

We can w.l.o.g. assume that $\overline{AB}$ is the long side of T with shortest v-distance to P's perimeter. Let p_1 be the point in C_P with largest y-coordinate below or on $\overline{AA'}$, such that the horizontal distance from p_1 to the extension of $\overline{DA}$ is between $s/6$ and $5s/6$. Symmetrically, let p_2 be the point in C_P with largest y-coordinate below or on $\overline{BB'}$, such that the horizontal distance from p_2 to the extension of $\overline{CB}$ is between $s/6$ and $5s/6$, see Fig. 3.8.

If p_1 and p_2 can be connected by a straight line segment within P, then let r be the C_{I}-rectangle in C_R with base $\overline{p_1p_2}$. Since the vertical difference between p_1 and p_2 is less than $s/6$ and since $|p_1p_2| > s$, the slope of r's left and right side is greater than six, hence r covers T_m, Fig. 3.9a.

Otherwise, if p_1 and p_2 cannot be connected by a straight line segment within P, there exists a vertex of P between $\overline{AB}$ and $\overline{p_1p_2}$. Let v be the vertex of P with shortest v-distance to $\overline{AB}$ lying between $\overline{AB}$ and $\overline{p_1p_2}$. If $|p_1v| < s$ or $|p_2v| < s$ then partition the region in T_m with x-coordinates between p_1 and v, or p_2 and v, into two rectangles which are then covered according to Lemma 3.5. Assume w.l.o.g. that $|p_1v|$ is greater than s. We now claim that the uncovered area of T_m can be covered by a C_{I}- or a C_{II}-rectangle in C_R. If $|p_2v|$ also is greater than s we do the corresponding procedure to the right of v. The region

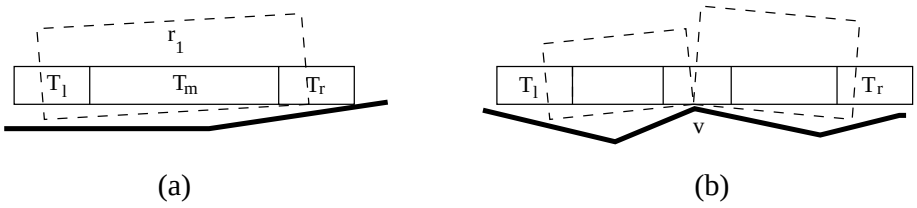

Figure 3.9: Covering T according to Lemma 3.6, Case b.

of T_m at most $s/4$ to the left and right of v is covered according to Lemma 3.5, see Fig. 3.9b. Recall that there exist rectangles in C_R that lie on the visibility lines and their extensions for every two vertices of P. So, even if p_1 and v cannot be connected by a straight-line segment within P there exists a rectangle r in C_R with lower right corner in v and with lower left corner, denoted p', between 0 and $s/6$ below $\overline{AA'}$ and with horizontal distance between $s/6$ and $5s/6$ from $\overline{AD}$. Since the vertical difference between p' and v is less than $s/6$, and since $|p'v| > s$, the slope of r's left and right side is greater than six, thus r covers the uncovered area of T_m to the left of v, as shown in Fig. 3.9b.

Before we show the case when the v-distance between both T's long sides and P's perimeter is less than $s/6$, we need to prove the following special case, which will be used in the proof of *Case d*.

Case c: Two diagonally opposite corners of T both touch vertices in P.

We can assume w.l.o.g. that the vertex v_1 coincides with A, and the vertex v_2 coincides with C, Fig. 3.10. Let p_1 be the point in C_P with largest y-coordinate below or on $\overline{BB'}$, such that the horizontal distance from p_1 to the extension of $\overline{BC}$ is between $s/4$ and s, Fig. 3.10a. Let r_1 be the C_{I}-rectangle in C_R with base $\overline{v_1p_1}$ and let α be the angle $(\overline{v_1p_1}, \overline{AB})$. If r_1 covers T_m then we are finished, otherwise there exists a vertex v_3, or an edge connected to v_3, above $\overline{CD}$ that prevented r_1 from covering T_m, Fig. 3.10b. Recall that there exist rectangles in C_R that lie on the visibility lines and their extensions for every two vertices of T. Thus there exists a C_{II}-rectangle r_2 in C_R with upper right corner at v_2 and upper left corner, denoted p_2, between 0 and $s/6$ above $\overline{DD'}$ and at horizontal distance between 0 and $s/4$ from $\overline{AD}$. Let β be the angle $(\overline{v_2p_2}, \overline{CD})$. We know that $\beta < \alpha$, since β is at most equal to the angle $(\overline{v_2v_3}, \overline{CD})$, (i.e., v_3, v_2, D). Thus r_2 cannot be stopped by any part of the perimeter above r_1's base, and it follows that r_2 covers T_m.

Case d: The v-distance between both T's long sides and P's perimeter is less than $s/6$.

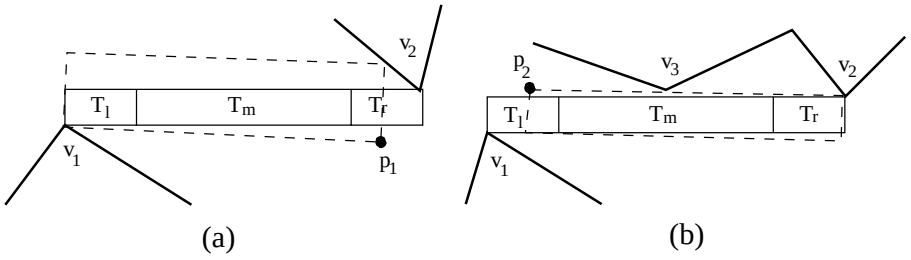

Figure 3.10: T_m is covered by a rectangle in C_R.

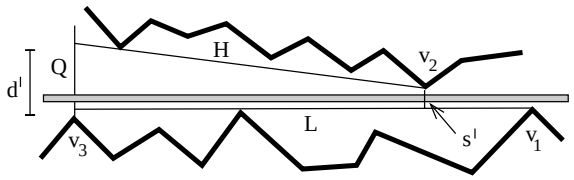

Figure 3.11: A $3v$-trapezoid within P.

Let v_2 be the vertex in P, above or on $\overline{C'D'}$, with smallest v-distance to $\overline{C'D'}$, and let v_1 be the vertex in P, below or on $\overline{A'B'}$, with smallest v-distance to $\overline{A'B'}$. We may assume w.l.o.g. that v_1 lies to the right of v_2. The region of T between v_1 and v_2 is covered according to *Case c*, since the part of T between v_1 and v_2 can be expanded in such a way that v_1 and v_2 touch T's opposite long sides. Let T_l' be the uncovered region of T to the left of v_2 and to the right of T_l and let T_r' be the uncovered region of T to the right of v_1 and to the left of T_r. Assume that T_l''s both long sides lie closer to P's perimeter than $s/6$, otherwise cover T_r' according to *Case a* or *b*. We will now prove that it is possible to cover T_l' with a constant number of rectangles in C_R. If T_r''s both long sides also lie closer to P's perimeter than $s/6$ then we do the corresponding procedure on T_r', otherwise we cover T_r' according to *Case a* or *b*.

Let v_3 be the leftmost vertex in P, such that the v-distance from v_3 to T_l' is less than $s/6$. Let Q be the vertical segment within P of length s with one endpoint at v_3, see Fig. 3.11. The part of T_l' to the left of Q can be covered according to *Case a* or *b*, since the shortest v-distance from P's perimeter to one of T_l''s long sides to the left of Q is greater than $s/6$. Let L be the lowest segment between v_1 and Q. Let H be the highest segment between v_2 and Q, such that H's and L's extensions intersect to the right of v_1. The segment on Q between H and L is denoted d'.

The shortest vertical segment from v_2 to L is denoted s'. The length of s' is the largest possible thickness of T. The two regions between s' and d' of T closer than $s/2$ from s' or d' are covered according to Lemma 3.5. Let t be the trapezoid bounded by the two parallel segments s' and d', and the parts of the segments H and L between s' and d'. This trapezoid is a $3v$-trapezoid, according to Definition 3.3, and according to the definition of C_R the uncovered region of t is entirely covered by C_{III}-rectangles in C_R. Therefore, since the thickness of these C_{III}-rectangles is s', the part of T in t can be covered by two of these rectangles in C_R. Thus the original rectangle T can be covered by a constant number of rectangles in C_R. □

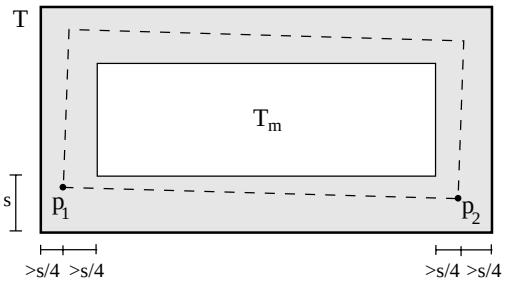

Figure 3.12: The shaded region is covered according to Lemma 3.6.

Lemma 3.7 *Every rectangle T, within a polygon P, whose shortest side is longer than $s/2$, can be covered by a constant number of rectangles in C_R.*

Proof: If $|AB| \leq 3s$ partition T into six equal rectangles which are then covered according to Lemma 3.6. Otherwise partition T in such a way that we can cover the border of T, that is the region in T that lies at most s from T's perimeter, according to Lemma 3.6. Let T_m be the uncovered region of T, Fig. 3.12. Let p_1 be the point in C_P with largest y-coordinate below or on the extension of T_m's base, such that p_1 lies between $s/4$ and $3s/4$ to the right of $\overline{AD}$, and let p_2 be the point in C_P with largest y-coordinate below or on the extension of T_m's base, such that p_2 lies between $s/4$ and $3s/4$ to the left of $\overline{BC}$. Let r be the C_{I}- or C_{II}-rectangle in C_R with base $\overline{p_1 p_2}$. Since the vertical distance between p_1 and p_2 is less than $s/6$, according to the definition of C_P, and since $|p_1 p_2|$ is greater than T_m's height, the rectangle r will cover T_m. □

3.3 Two algorithms and their complexity

In Lemmas 3.5-3.7, we have shown that C_R includes a covering which is within a constant factor of the optimal. Thus it is possible to translate the original geometrical covering problem into the **set-covering problem**, as follows.

An instance (X, F) of the **set-covering problem** consists of a finite set X and a family F of subsets of X, such that every element of X belongs to at least one subset in F. We want to cover the polygon P with a minimum number of rectangles in C_R. Define X to be the set of all cells in the partition induced by the perimeters of all the rectangles in C_R, (thus $|X| = O(|C_R|^2)$), and we define F, such that every element $f \in F$ corresponds to a rectangle r in C_R, where f is the subset of X that is included in r. In the following two sections we will show how one can use this translation to find two algorithms for the polygon-

covering problem by using two known set-covering algorithms. Before we show the algorithms we note the following observation:

Observation 3.8 *If the input polygon is given as integer coordinates in the universe* $[0..u]$ *then* $s \geq \frac{1}{\sqrt{2}\cdot u}$.

Proof: Let v_1 and v_2 be two vertices of P connected by an edge e, and let v' be an arbitrary vertex of P. We will use the notation $v.x$ and $v.y$ to denote the x- respectively the y-coordinate of a point v. We want to compute the shortest possible distance between v' and e. Assume that there exists a point p on e with the same y-coordinate as v', then the distance between p and v' is:

$$|v'.x - p.x| = |v'.x - (v_1.x + \frac{|v_1.x - v_2.x|}{|v_1.y - v_2.y|}|p.y - v_1.y|)|.$$

Let $\alpha = |v_1.x - v_2.x|$, $\beta = |v_1.y - v_2.y|$ and let $\gamma = |p.y - v_1.y|$. Thus we have $|v'.x - p.x| = |v'.x - (v_1.x + |\frac{\alpha}{\beta} \cdot \gamma|)|$. Since α, β and γ are in the universe $[0..u]$ and if $p \neq v'$, the horizontal distance between p and v' is:

$$|v'.x - p.x| \geq 0 + \frac{1}{u} \cdot 1 = \frac{1}{u}.$$

According to symmetry we will get the same result if p's and v''s x-coordinates are equal. Since e can cross the horizontal line $v'.x$ and the vertical line $v'.y$ at a distance $1/u$ from p, we have that the shortest distance, s, between v' and e is $\frac{1}{\sqrt{2}u}$, as shown in Fig. 3.13. □

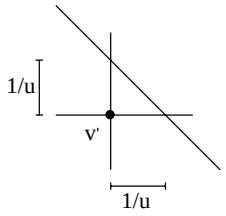

Figure 3.13: The shortest distance between a vertex v' and an edge of P is $\frac{1}{\sqrt{2}u}$.

3.3.1 An exponential algorithm

This is a simple straight-forward algorithm which checks if there is any subset of k rectangles in C_R that cover P. If such a subset is found then we are finished, otherwise increment k and continue as described above. The initial value of k is 1 and from Lemmas 3.5-3.7 we know that C_R includes a covering which is

within a constant factor of the optimal, thus k=1, ..., $O(\mu(P))$. Since there are $((d/s)^7)^{O(\mu(P))}$ subsets of C_R with $O(\mu(P))$ rectangles, we can find a covering with $O(\mu(P))$ rectangles in time:

$$\sum_{k=1}^{O(\mu(P))} ((d/s)^7)^k = (d/s)^{O(\mu(P))} = 2^{O(\mu(P)\cdot \log d/s)}.$$

According to Theorem 1 in [58] it holds that $\mu(P) = O(n \cdot \log d/s)$, thus we get the time complexity $2^{O(n\cdot(\log d/s)^2)}$. Hence by Observation 3.8 we obtain the following theorem:

Theorem 3.9 *When the vertices of the input polygon are given as integer coordinates in the range* $[0..u]$, *the algorithm described above will produce a covering within a constant factor of the optimal in* $2^{O(n\cdot(\log u)^2)}$ *time.*

3.3.2 A polynomial algorithm

A natural approach to find an approximation algorithm would be to use a known set-covering algorithm, for example a greedy algorithm. A Greedy-Set-Cover algorithm [27] can easily be implemented to run in time $O(|X||F|min(|X|,|F|))$, with a ratio bound of $(\ln|X|+1)$. Recall that $|X|$=$|C_R|^2$ and F=C_R.

If the input polygon P is given in integer coordinates, where the coordinates are in the universe $[0..u]$, the above greedy algorithm produces a covering which is within a logarithmic factor of the optimal, $(\ln|C_R|^2+1)$, in pseudo-polynomial time $O(((d/s)^7)^4)$=$O((d/s)^{28})$. Since $s\geq\frac{1}{\sqrt{2}\cdot u}$ and $d\leq\sqrt{2}\cdot u$ we have that the time-complexity for the greedy algorithm is $O(u^{56})$. So we obtain the following:

Theorem 3.10 *When the* n *vertices of the input polygon are given as polynomially bounded integer coordinates, the algorithm described above will produce a covering with a logarithmic approximation factor in polynomial time.*

3.4 Lower bounds

From Lemmas 3.5-3.7 we derived the following proposition:

Proposition 3.11 *There exists a constant* c, *such that for any polygon* P *there exists a finite set,* C, *of rectangles lying within* P, *where every rectangle in* P *can be covered by* c *rectangles from* C.

In Sections 3.1 and 3.2 we constructed a finite set C_R of size $O((\frac{d}{s})^7)$ and proved that C_R includes a covering which is within a constant factor, c, of the optimal. In Section 3.2 we focused on simple and short proofs for proving that c is a

constant, therefore by just calculating c from Lemmas 3.5-3.7 the value of c would be close to 1000. We conjecture that if we use the set C_R the constant factor c is below 20. We may note here that $O((\frac{d}{s})^7)$ is much greater than $\mu(P)$. Since $n{=}O((\frac{d}{s})^2)$, it follows from [60] (see introduction) that $\mu(P){=}O((\frac{d}{s})^2 \log(\frac{d}{s}))$.

Two questions arise naturally in connection with these proofs:

a) What is the minimum constant c, for which the above proposition holds? That is, if we do not have any restrictions on C except that it should be finite?

b) How large does C have to be in the worst case, in terms of n, d and s, in order for the proposition to hold?

Therefore, in this section we give lower bounds for (*a*) and (*b*).

Theorem 3.12 *For all integers $n{\geq}8$, there exists a polygon P_n with n vertices, such that for any finite set, C, of rectangles lying inside P_n, there exists a rectangle T within P that cannot be covered by fewer than six rectangles in C.*

Proof: To prove the theorem we consider a polygon P_8 with eight sides, two horizontal parallel long sides of length $2l$ and six sides of length l, such that the angle between every pair of incident edges is 135 degrees, Fig. 3.14a. We place a maximal rectangle T within P, such that (1) its corners touch each one of the non-horizontal and non-vertical edges of P, and (2) its corners do not coincide with any of the corners of any rectangle in C. It is easily seen that there always exists such a rectangle T, independently of which method we employ to choose the rectangles in C. Now we can see that the minimum number of rectangles in C that is needed to cover T is at least six. □

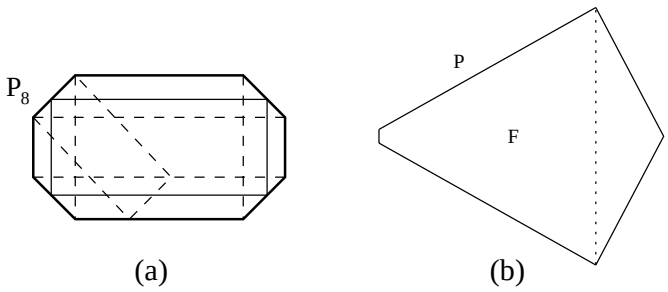

Figure 3.14: (a) T is described by the solid thin lines within the polygon. (b) A polygon P for which the lower bound in Theorem 3.13 would hold.

Theorem 3.13 *For every $n{\geq}5$, and $d{>}s{>}0$ there exists a polygon P, such that for any set of rectangles, C, for which it holds that every possible rectangle within P can be covered by at most c rectangles of C, it holds that $|C|{=}(d/s)^{\Omega(1/c)}$.*

Proof: Consider a funnel F whose arms (if extended) form an angle of 60 degrees, as shown in Fig. 3.15a, with base of length d and top of length s. Partition F into kc subfunnels (where k is some constant > 1), $F_1, \ldots, F_{kc}$, such that the ratio between the length of every subfunnel's base (d') and top (s') is $(d'/s')=(d/s)^{1/kc}$. Let F_i be any one of these subfunnels of F. A rectangle r is said to *pass through* a funnel if and only if r's two long sides intersect both F_i's top and base. If two rectangles r_1 and r_2, that pass through F_i, should overlap each other in F_i's top and base, the angle between r_1's and r_2's long sides would have to be less than $\frac{2\cdot 60^\circ}{\sqrt{3}} \cdot \frac{s'}{d'}$, Fig. 3.15b. This implies that, if a set of rectangles, that covers F_i, includes only rectangles that pass through F_i then the number of rectangles in this set would be at least $\frac{2}{\sqrt{3}} \cdot (d/s)^{1/kc}$.

For simplicity we rotate F such that F's lower arm lies horizontal and F's upper arm has an angle of 60° to the horizontal line. A rectangle r whose long sides have an angle of $0 \leq \alpha \leq 60^\circ$ to the horizontal line is said to be a C_i-rectangle, where $i=(\lceil \frac{\alpha (d/s)^{\frac{1}{2c}}}{60} \rceil)$. That is, we classify the rectangles with respect to the slope of their long sides into $(d/s)^{1/2c}$ classes. Let us assume that there exists a set C, in contradiction to the statement of the lemma, consisting of less than $(d/s)^{\Omega(1/c)}$ rectangles. This implies that at least one of the classes of rectangles, say C_i, is not represented in C. Let T be a C_i-rectangle that passes through F. From the above arguments we have that T has to be covered by at least one local rectangle within each one of the kc subfunnels. Hence, more than c rectangles in C are needed to cover T. We have a contradiction. □

Note that the theorem holds for any polygon containing the funnel F. For example, we can construct a polygon P by just adding three segments to F, as shown in Fig. 3.14b.

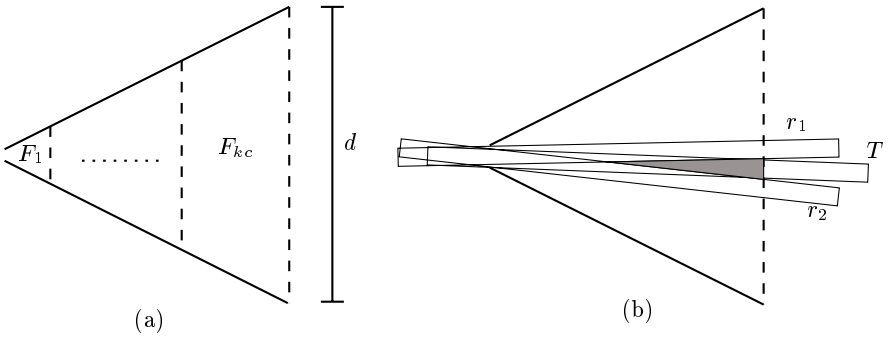

Figure 3.15: (a) A funnel F partitioned into kc subfunnels. (b) Two rectangles overlapping at F_i's top.

Chapter 4

Linear-time covering

We continue to examine the problem of covering a polygon with a minimum number of rectangles. Here we will establish that, for simple polygons, the algorithm by Levcopoulos produces $O(min[n+\mu(P), \alpha(n)\cdot\mu(P)])$ rectangles in time $O(n+\mu(P))$, where $\alpha(n)$ is the extremely slowly growing inverse of Ackermann's function. For proving this result we develop new techniques which we believe are interesting themselves, and can be used, e.g., for showing properties of other types of coverings.

4.1 Preliminaries

In the following sections we will perform a close analysis of the algorithm by Levcopoulos [58], in the case when the given polygon is simple. To fully understand this result we will present the original algorithm by Levcopoulos, which will be denoted H. We start with some simple definitions which will be used throughout this chapter.

The *generalized Voronoi diagram* [54, 96] of a finite set of objects, S, is a partition of the plane so that each region of the partition is the locus of points which are closer to one member of S than to any other member. In the standard Voronoi diagram the objects in S are just points, in our case we will have to extend the definition to include open line segments as well as points, see Fig. 4.1. The part of the generalized Voronoi diagram lying within P partitions P into $n + w$ faces, where each segment and each concave vertex induces a face of the Voronoi diagram (w is the number of concave vertices of P).

A *funnel cell* of a polygon P is a trapezoidal piece of a Voronoi face in P having the following properties. Let A, B, C, D be the vertices of the trapezoid in counter-clockwise order, such that $\overline{AD}$ is parallel to and shorter than $\overline{BC}$ and

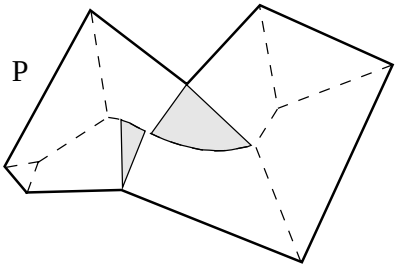

Figure 4.1: Generalized Voronoi diagram (dashed) within P. Faces induced by concave vertices are shadowed.

the interior angle $\angle ADC$ is greater than 90 and less than 135 degrees, as shown in Fig. 4.2a. The segment $\overline{AB}$ lies on an edge of P, say e, and $\overline{CD}$ is a Voronoi edge bounding the Voronoi face induced by e in P. By this, and by the definition of generalized Voronoi diagrams [54], it follows that the mirror image of a funnel cell with respect to the Voronoi edge bounding the cell is also a funnel cell. Such a pair of funnel cells with a Voronoi edge separating them is denoted a *funnel shell*, Fig. 4.2a.

Let A' and B' be the symmetric image of A respectively B. Let β be the straight-line segment with endpoints in B and B', and let τ be the shortest segment, within the shell, touching both AB and $A'B'$. The part of the shell bounded by β, τ and the segments AB and $A'B'$ is denoted a *funnel* of P.

Let t be any funnel of P. The *base* of t is defined as the segment β_t, and the *top* of t is defined as the segment τ_t, Fig. 4.2b. The segments $\overline{AB}$ and $\overline{A'B'}$ are denoted the *arms* of t. Finally, the *characteristic angle* of t, denoted α_t, is defined to be half the smallest angle built by the intersection of the straight extension of t's arms. When it is clear from the context which funnel we refer to, we shall simply write β, τ and α instead of β_t, τ_t and α_t.

4.1.1 Levcopoulos' algorithm

The algorithm works by iteratively partitioning the polygon P into smaller and smaller pieces which are processed independently until every resulting piece can be trivially covered by some rectangles lying within P. The first partition is achieved by drawing the skeleton of the polygon, also called "generalized Voronoi diagram" and "medial axis".

First step: Generate a list of Voronoi faces

The first step of the algorithm is to construct a list with all faces of the generalized Voronoi diagram of P. Each face is represented by a list of its edges in

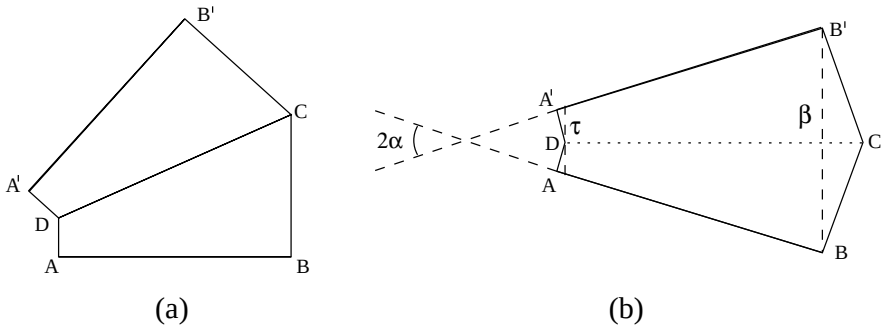

Figure 4.2: (a) Two funnelcells. (b) A funnel shell. (solid lines)

clockwise order. By using the results in [54] this step takes $O(n \log n)$ time.

Second step: Cover each face independently
The rest of the algorithm consists of independently processing each face and outputting a set of rectangles which covers it. Each face is processed in time linear with respect to the number of the edges comprising the face, plus the number of rectangles produced during processing that face. The total number of edges is $O(n)$ and therefore the overall time performance of this step is $O(n+H(P))$, where $H(P)$ is the number of rectangles produced by H.

There are two major cases: (1) when the face is induced by a concave vertex of the polygon, and (2) when it is induced by an edge of the polygon. We describe the action of the algorithm independently for each one of these cases. We will only give a very brief description of the algorithm. The complete algorithm can be found in [58].

Case 1. The face is induced by a concave vertex of the polygon
In this case the face is covered by two rectangles. Let W be the concave vertex inducing the face. Let s be a straight line splitting the concave angle at W into two equal angles. On each side of s one rectangle is placed, with one of its edges collinear with s. Moreover, the rectangles placed are the smallest ones which cover the whole face on the respective side of s, see Fig. 4.3a. It is easily seen that these two rectangles can be determined in linear time with respect to the number of Voronoi edges bounding the face.

Case 2. The face is induced by an edge of the polygon
Let g be the segment of P which induces the face. The face is partitioned into cells with only three or four vertices each, by drawing from every vertex of the face a segment connecting the vertex with its perpendicular projection on g.

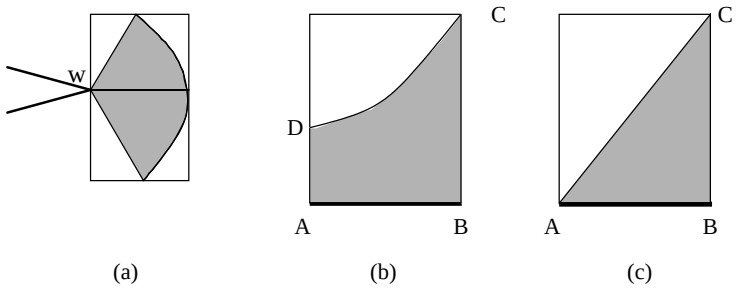

Figure 4.3: (a) Case 1: Rectangles produced to cover a face induced by the concave vertex W. (b) Subcase 2.1. (c) Subcase 2.2

Definition 4.1 The edge of each cell which is at the boundary of P is called the *base* of the cell. The segments which are perpendicular to its base are called the *sides* of the cell and, finally, the edge opposite to its base is called the *top* of the cell.

Definition 4.2 A cell is said to be *trivially coverable* iff the smallest rectangle whose one side is the base of the cell and which covers the cell lies entirely within the polygon.

In the continuation of the algorithm each cell is processed independently. The action of the algorithm is described independently for various types of cells.

Subcase 2.1. The top of the cell is a paraboloid
In this subcase the cell is covered with one or two rectangles. In order to facilitate an easier description, we assume w.l.o.g. that the base of the cell lies at the bottom, horizontally. Assuming this orientation, if the top is monotonically increasing (or decreasing) then the cell is trivially coverable, see Fig. 4.3b. Otherwise, let A be the lowest point of the top of the cell. Split the cell into two subcells, by drawing a segment from A to its perpendicular projection onto the base. The resulting subcells are trivially coverable. It is easily seen that both cases can be handled in constant time by the algorithm.

Subcase 2.2. The cell is a triangle
Let A, B and C be the vertices of the cell in counter-clockwise order, such that $\overline{AB}$ is the base of the cell and $\overline{BC}$ the side of the cell. The angle, $\angle ABC$, is of 90 degrees. The rectangle with base $\overline{AB}$ and top $\overline{DC}$ covers the cell and lies within P, see Fig. 4.3c.

Subcase 2.3. The cell is a trapezoid
If the cell is a rectangle then, of course, it is trivially coverable. Otherwise, let

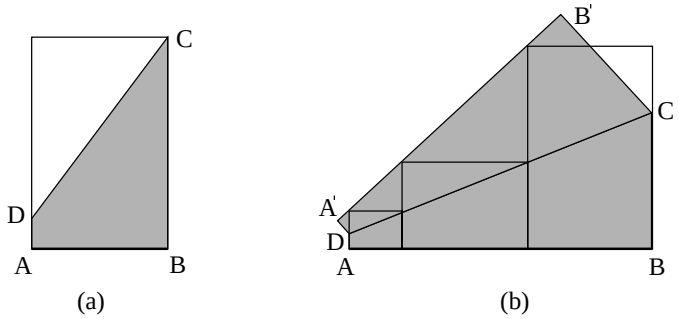

Figure 4.4: (a) If $\angle ADC$ is at least 135 degrees then the cell is trivially coverable. (b) If $\angle ADC$ is greater than 135 degrees then the cell is covered by $\lceil \log_c |BC|/|AD| \rceil$ rectangles.

A, B, C and D be the vertices of the cell in counter-clockwise order, where A and B are the endpoints of its base. Assume w.l.o.g. that $|AD| < |BC|$ and that the base is horizontal and at the bottom. If $\angle ADC$ is greater than or equal to 135 degrees, then the cell is trivially coverable, Fig. 4.4a.

There is one case left, when $\angle ADC$ is greater than 90 and less than 135 degrees. In order to compute the maximal height of any rectangle within the polygon with base on $\overline{AB}$, flip the cell around its top (with endpoints C and D) obtaining the symmetrical image of the cell with respect to its top, as shown in Fig. 4.4b. Let A' and B' be the symmetric image of A and B respectively. Let E be the point on $\overline{A'B'}$ or on $\overline{B'C}$, such that the segment $\overline{AE}$ goes through D. If $|AE| \geq |BC|$, then the cell is trivially coverable. It remains to consider the case $|AE| < |BC|$.

If $|AE| < |BC|$, then E lies on $\overline{A'B'}$. Let α be the angle $\angle ADC$ minus 90 degrees. We have $\angle A'DE = 2\alpha$. Moreover, we have $|DE| = |A'D|/\cos(\angle A'DE)$. Since $|A'D| = |AD|$ we get $|AE| = |AD| + |DE| = |AD| \times (1 + 1/\cos 2\alpha)$, Fig. 4.4b.

Definition 4.3 Assuming the above context, we define the *funnel ratio* of a funnel cell to be $\frac{|AE|}{|AD|} = (1 + \frac{1}{\cos 2\alpha})$.

Now the algorithm splits the cell into two subcells, $AFE'D$ and $FBCE'$, where $\overline{EE'}$ is parallel to $\overline{AB}$, and E' lies on $\overline{DC}$, as shown in Fig. 4.5. The subcell $AFE'D$ is covered trivially. If the subcell $FBCE'$ is not trivially coverable, it is processed in the same way as funnel cells. Thus, the original cell $ABCD$ is eventually partitioned into $\lceil (\log_c(|BC|/|AD|)) \rceil$ trivially coverable subcells, where c is the funnel ratio of the cell.

From the above argumentation we get the following lemma:

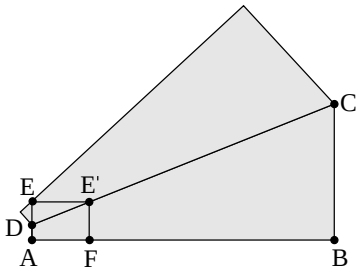

Figure 4.5: The algorithm splits the cell into two subcells, $AFE'D$ and $FBCE'$

Fact 4.4 *After constructing the Voronoi diagram, in $O(n \log n)$ time, any polygon P, except its funnels, can be covered with $O(n)$ rectangles in time $O(n)$. Within the same asymptotic time, all funnels of P, if there are any, can be detected.*

The above fact shows advantageous properties of the algorithm with respect to non-funnel parts of polygons. The following fact shows how good the algorithm is with respect to covering funnels.

Fact 4.5 *Let t be an arbitrary funnel of P. Algorithm H produces $O(\mu(t))$ rectangles to cover it, where $\mu(t)$ is the minimum number of rectangles needed to cover t.*

Hence, the following theorem is obtained:

Theorem 4.6 *Let P be an arbitrary polygon, and let C be the set of funnels in P. The algorithm described above covers P with $O(n+H(P))$ rectangles in time $O(n \log n+H(P))$, where $H(P) = O(n+\sum_{t \in C} \mu(t))$.*

4.2 Idea and approach

In the rest of this chapter we study the difference between a global (optimal) covering and a local covering of a simple polygon. We will show that algorithm H, which is a typical local covering algorithm, produces $H(P) = O(n+\mu(P))$ rectangles to cover a simple polygon P, where $\mu(P)$ is the minimum number of rectangles needed to cover P.

As described above, H works by partitioning P into $O(n)$ cells which are then covered separately. It has been shown that H covers a polygon P with $O(n)$ rectangles, Theorem 4.6, provided that P does not contain any so-called

funnels. In [60] it was shown that for non-simple polygons that contain funnels H may need $\Omega(n \log n + \mu(P))$ rectangles to cover P, even if $\mu(P) = O(n^{0.5+\epsilon})$.

The advantage of a global covering versus a local covering is that the global algorithm may cover several funnels at a time instead of just one funnel at a time, i.e., one may use long thin rectangles that overlap several funnels by using a global covering algorithm.

Our claim is that for any simple polygon, a global optimal covering of all the funnels is not much better than a covering produced by H, Lemma 4.8. The proof consists of two main results, Lemma 4.15 and Lemma 4.30. Assume that we partition all the funnels into a number of subsets, such that all the funnels in a subset belong to the same weak visibility polygon, have approximately the same orientation and the length of their tops are approximately the same. We say that these subsets are sorted normalized sets. If this holds then we show in Lemma 4.15 that our claim is correct, i.e., a local covering is almost as good as an optimal covering. In the proof we assume the opposite, that a global covering is much better than a local covering. But, if a global covering would be much better than a local covering, then many funnels in the set have to be "mostly" covered by rectangles that also overlap several other funnels. To prove Lemma 4.15 we show that all funnels in a subset that are not mostly covered by such rectangles may be covered locally without increasing the number of rectangles by more than a constant factor. We then extend this argument and show that it is even possible to locally cover most of the funnels, that are "mostly" covered by rectangles that also overlap several other funnels, without increasing the number of rectangles by more than a constant factor. The remaining set of funnels that cannot be covered locally without increasing the number of rectangles by more than a constant factor is denoted N. We conclude the proof of Lemma 4.15 by proving that the set N has to be empty. The main idea of the proof of Lemma 4.15, is to show that for every funnel in N there exists a funnel in the set, such that the two funnels cannot be connected by three straight line segments, thus violating the fact that they lie in the same weak visibility polygon.

In Lemma 4.30 we extend the above result to hold even for normalized subsets, that is without the restriction on the length of their funnel tops. As it turns out, this is enough to prove our main results, since it can be argued that any rectangle at most can overlap a constant number of such subsets.

Finally we prove a tight lower bound, $\Omega(n/\alpha(n))$, Theorems 4.31 and 4.32, for the minimum number of rectangles needed to cover a simple polygon P, by using some known results about Davenport-Schinzel sequences.

By combining the two main results, Theorem 4.7 and 4.31, we get that H will produce a rectangular covering of a simple polygon P within an $O(\alpha(n))$ approximation factor from the minimum in optimal time $O(n+\mu(P))$.

4.3 Improving the approximation factor for simple polygons

The region of the funnel shell to the right of β can be covered by two rectangles since the angle $\angle B'CB$ is greater than 90°, the same holds for the region to the left of τ within the funnel. Since the area of the funnel shell outside τ and β can be covered by a constant number of rectangles we will in the continuation of this text only consider the funnels in P. Since some funnels in P can be covered by a constant number of rectangles we will only consider the funnels of P for which it holds that the minimum number of rectangles needed to cover a funnel is greater than 20. The total number of rectangles added to a covering by only considering the funnels, instead of the funnel shells is linear. Let C be the set of funnels in P, and let $\mu(C)$ denote the minimum number of rectangles, within P, needed to cover the funnels in C.
We will prove the following theorem:

Theorem 4.7 *If P is an arbitrary simple polygon, then it holds that $H(P) = O(n+\mu(P))$.*

Remark: The constants hidden by the O-notation are very large, we believe that this is due to the construction of the proof, and not to the problem itself.

Since there are $O(n)$ funnels in P and since $H(P) = O(n+\sum_{t\in C}\mu(t))$, according to Theorem 4.6, it suffices to show the following lemma:

Lemma 4.8 *Let C be the set of funnels in an arbitrary simple polygon P. Then it holds that:*

$$\sum_{t\in C}\mu(t) = O(\#C + \mu(C))$$

We will start by partitioning, in three steps, the set of funnels C into smaller subsets. First we partition the polygon P into $O(n)$ weak visibility polygons, which are defined as follows:

Definition 4.9 A polygon P is said to be *weakly visible* from an edge pq of P if for every point x in P there exists a point y in pq that is visible from x.

The partitioning is done as follows. Select an arbitrary edge e of P. Construct a weak visibility polygon $P_1 \subseteq P$ with respect to e. That is, P_1 will include all points of P that are visible from e. The weak visibility polygon P_1 will include edges that are not edges in P. Select one of these edges, denoted e, and construct a weak visibility polygon P_2 such that $P_1 \cap P_2 = e$. Continue this procedure iteratively until P is completely partitioned into weak visibility polygons $P_1, \ldots, P_m$.

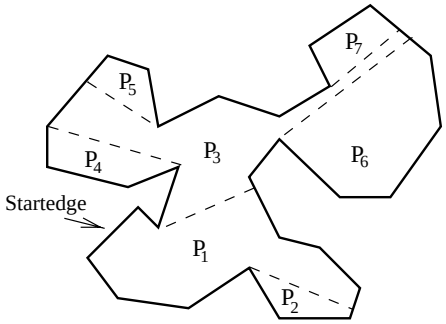

Figure 4.6: A simple polygon partitioned into weak visibility polygons.

A funnel is said to be in a weak visibility polygon P_i if its interior overlaps with P_i. The set of funnels in P_i is denoted C_i. The following observation is now straight-forward.

Observation 4.10 *Let P be a simple polygon partitioned into weak visibility polygons as described above. Every rectangle within P overlaps with at most three constructed weak visibility polygons.*

For every weak visibility polygon P_i, we sort the funnels in C_i with respect to their orientation into a constant number of funnel sets $T_1, \ldots, T_k$. The orientation of a funnel t is the direction of the corresponding edge of the generalized Voronoi diagram (the direction of a straight line from the middle of t's base to the middle of t's top). We say that the funnel sets $T_1, \ldots, T_k$ are *normalized*. Hence, the funnels in a normalized set all belong to a weak visibility polygon and have approximately the same orientation (within $\frac{360}{k}$ degrees).

Since every rectangle that overlaps a simple polygon P can overlap at most three of the constructed weak visibility polygons, according to Observation 4.10, and since we have a constant number of funnel sets for every weak visibility polygon, we can without loss of generality reduce the original problem to each of these funnel sets T_j, $1 \leq j \leq k$. Let T be any of the funnel sets $T_1, \ldots, T_k$. So, to prove Theorem 4.7, it suffices to show:

$$\sum_{t \in T} \mu(t) = O(\#T + \mu(T)) \tag{1}$$

We now sort the funnels in T with respect to the length of their tops, and get the subsets $F_1, \ldots, F_s$, where we may assume without loss of generality that the smallest top has length 1. We sort the funnels in such a way that for every funnel $t \in F_i$, $1 \leq i \leq s$, it holds that $2^{i-1} < |\tau_t| \leq 2^i$. Let l_i denote the

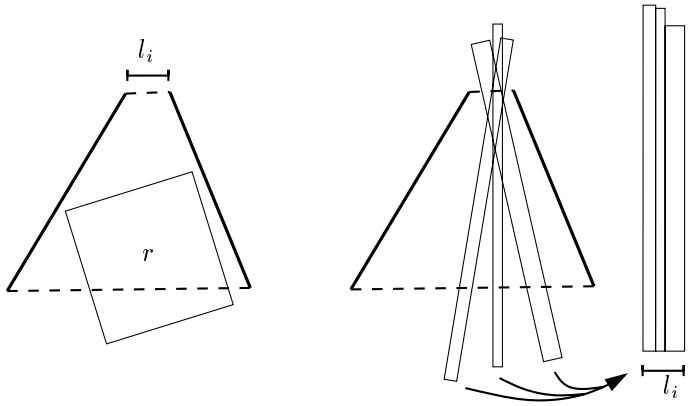

Figure 4.7: (left) An i-local rectangle r. (right) Three rectangles that can be grouped into one i-collection.

length of the largest allowed funnel top in F_i, that is $l_i = 2^i$. We say that F_i is a *sorted normalized* set of funnels. The reason why this partition is done is because the proof of (1) will be made in two steps. In the first step we will show that (1) holds for sorted normalized sets. Then we show that this result can be generalized to normalized sets of funnels.

Before we continue, we need to define the different types of rectangles and collections of rectangles that may overlap any sorted normalized set of funnels. The thickness of a rectangle is the length of its shortest side. The total thickness, or just thickness, of a set of rectangles is the sum of the rectangles thickness.

Definition 4.11 Every rectangle in a covering of F_i is either an i-local rectangle or a member in exactly one i-collection. An *i-collection* is a set of rectangles R overlapping F_i, whose total thickness is less than or equal to l_i. Every rectangle that overlaps F_i and whose thickness is greater than l_i is defined as an *i-local rectangle*. The minimum number of i-collections and i-local rectangles required to cover a funnel set F_i is denoted $\mu''(F_i)$.

The above definition distinguishes between rectangles that may overlap several funnels in a sorted normalized set, the rectangles in an i-collection, and the rectangles that only can overlap a constant number of funnels, the i-local rectangles, see Fig. 4.7. Note also that the number of i-collections and i-local rectangles needed to cover a funnel set F_i is always less than or equal to the number of rectangles needed to cover F_i. This follows since all rectangles of thickness less than l_i may be grouped into i-collections of total thickness l_i.

We need some additional definitions.

Definition 4.12 A set S of i-collections and i-local rectangles is said to be *approximately optimal* if S covers F_i and $\#S \leq c \cdot (\#F_i + \mu(F_i))$, where c stands for some constant independent of the input.

For any approximately optimal set S it holds that every rectangle in S is either an i-local rectangle or a member in exactly one i-collection.

Definition 4.13 A rectangle r is said to *pass through* a funnel t if and only if r's two long sides intersect both t's top and base. We say that a set of rectangles is a *t-collection*, where t is any funnel in any F_i, if (1) the rectangles pass through t, (2) they do not pass through any other funnels in F_i and (3) their total thickness is less than or equal to $|\tau_t|$. Every rectangle that overlaps t and does not pass through any funnels in F_i is defined as a *t-local rectangle*. The minimum number of t-collections and t-local rectangles required to cover any funnel t is denoted $\mu''(t)$. Hence, we have $\mu''(t) \leq \mu(t)$.

Any t-local rectangle or any rectangle in a t-collection in some approximately optimal set S is also an i-local rectangle or a member in an i-collection of S. Also note that the rectangles in a covering produced by H are either t-local rectangles or members in a t-collection, while the rectangles in an optimal covering are either i-local rectangles or members in i-collections.

Observation 4.14 *For an arbitrary funnel t, it holds that $\mu(t) = \Theta(\mu''(t))$.*

Proof: The proof of Lemma 1 in [58] can easily be modified to t-collections and t-local rectangles. See also the proof of Observation 4.19. □

This observation allows us to translate the left side of (1) to t-collections and t-local rectangles, i.e., to prove Theorem 4.7 it suffices to show that

$$\sum_{t \in T} \mu''(t) = O(\#T + \mu(T)). \tag{2}$$

4.3.1 Covering a sorted normalized set of funnels locally

We will now show, Lemma 4.15, that the minimum number of t-collections and t-local rectangles needed to cover every funnel in a sorted normalized set of funnels, F_i, is not "much worse" than the minimum number of i-collections and i-local rectangles needed to cover F_i, or more precisely:

Lemma 4.15 *Let F be any of the sets $F_1, \ldots, F_s$. Then it holds that:*

$$\sum_{t \in F} \mu''(t) = O(\#F + \mu''(F))$$

To prove the above lemma we will start by showing that in a number of steps we can cover some funnels in F with $O(\#F+\mu''(F))$ t-local rectangles and t-collections. Then in the remaining part of this section we will show that this set actually covers every funnel in F.

We may assume without loss of generality that the funnels in F have their tops pointing upwards. Now, if the lemma would not hold then many of the funnels in F would have to be "mostly" covered by rectangles that also overlap other funnels in F. We have to define (Definition 4.17) what we mean when we say that a funnel is "mostly" covered ("well-fanned") by rectangles that also overlap other funnels. First we need the following definition.

Definition 4.16 A funnel t_1 is said to be *above* a funnel t_2 if and only if t_2's and t_1's funnel tops can be connected by a straight line segment within P that intersects β_{t_1}. In this case t_2 is said to be *below* t_1. The funnel t_1 is said to be *directly above* t_2 if and only if t_1 is above t_2, and t_1 and t_2 can be connected by a straight line segment within P that does not pass through any other funnels in F.

Definition 4.17 Let M be a set of rectangles that pass through a funnel $t\in H$, such that every rectangle in M also overlaps at least one funnel (may be different funnels) in H below t. We define a function $\varphi(t, M, H)$ as the largest distance, on β_t, between two consecutive rectangles in M such that their intersection with t is not completely overlapped by t-local rectangles and t-collections. If $\varphi(t, M, H)<|\beta_t|/10$ we say that the funnel t in H is *well-fanned* by M, as shown in Fig. 4.8.

Covering non well-fanned funnels

Our objective in this section is to show that all funnels that are not well-fanned by an approximately optimal set can be covered by t-local rectangles and t-collections. Thus, we will not have to consider these funnels in the continuation.

Let S be an approximately optimal set. Then, any rectangle in S overlapping any funnel $t\in F$ that does not overlap any other funnels below t is either a t-local rectangle or a member in an i-collection where t is the lowest funnel the rectangle overlaps. Since we do not consider these rectangles in Definition 4.17, where t is the lowest funnel they overlap, we have to show that we can replace (i.e., cover) these, in t, with t-collections or t-local rectangles, without increasing the size of S more than allowed.

Proposition 4.18 *Let D be the set of i-collections in some approximately optimal set S and let D_R be the set of rectangles in D. For each rectangle $r\in D_R$*

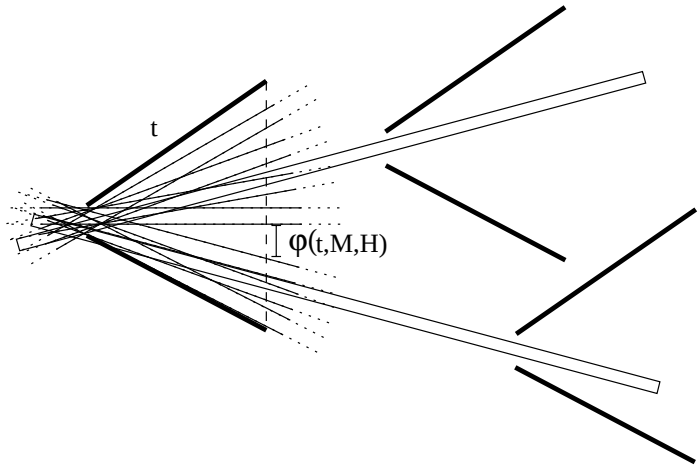

Figure 4.8: If $\varphi(t, M, H) < |\beta_t|/10$ then $t \in H$ is well-fanned by M

let t_r be the lowest funnel that r overlaps. Every rectangle in D_R that passes through more than one funnel in F can be overlapped by a local rectangle in t_r, such that r's entire intersection with t_r is covered by this local rectangle. If we overlap all rectangles in D_R as described and then group the local rectangles into a minimum number of t-collections, the number of new t-collections will be $O(\#F+\mu''(F))$.

Proof: Let m be the number of t-collections that is constructed according to the Proposition. The smallest possible total thickness of these t-collections will be greater than $(m-\#F)\cdot l_i/2$, since every t-collection, except for at most one for every funnel t, will have a total thickness greater than $l_i/2$ (otherwise we could have grouped them into a smaller number of collections). Also, the largest total thickness possible by the $\#D$ i-collections is $\#D \cdot l_i$, according to Definition 4.11, which gives us the equation:

$$(m - \#F) \cdot l_i/2 < \#D \cdot l_i \quad \Longleftrightarrow \quad m < 2 \cdot \#D + \#F.$$

Since $\#D = O(\#F+\mu''(F))$, it holds that:

$$m = O(\#F+\mu''(F)).$$

□

We will now show that a funnel $t \in F$ that is not well-fanned by some approximately optimal set S may be entirely covered by t-collections and t-local rectangles, without increasing the size of S by more than a constant factor.

Recall that $\mu''(t) = \Theta(\log_\rho(|\beta_t|/|\tau_t|))$, according to Observation 4.14. The following observation is a modified version of Lemma 1 in [58].

Observation 4.19 *Let M be the set of rectangles that passes through a non well-fanned funnel $t \in F$. If $\mu''(t) > 20$ then it holds that at least $\log_c(|\beta_t|/|\tau_t|)$ additional rectangles are needed to cover t, where $c = 1 + \frac{20}{1-\tan\alpha_t}$.*

Proof: We can without loss of generality assume that t's orientation is exactly vertical. In the rest of this proof all rectangles or sets of rectangles which we define or refer to are understood to be t-local rectangles which overlap with t.

To continue we need some definitions. Let β' be the part on t's base that is not covered by M, according to Definition 4.17, and let h_0 be the horizontal segment in t, such that h_0 touches both arms of t and $|h_0| = 40|\tau|$. Let t' be the largest trapezoid in t, not overlapped by the rectangles in M, with β' as its base and its top on h_0. Let τ' be the top of t'. Since $|\beta'| > \frac{|\beta|}{10}$ we have that $|\tau'| > \frac{h_0}{10} - 2\tau$, and hence, $\tau' > \frac{h_0}{10} - \frac{h_0}{20} = 2\tau$.

Let R be a set of rectangles. Let h_R be the lowest horizontal segment in t', such that h_R touches both arms of t', and the whole trapezoid between τ' and h_R is covered by the rectangles in R. We say that R is *top-coherent*, if and only if no rectangle in R lies entirely below h_R. If R covers t', then it is top-coherent, and $|h_R| = |\beta'|$.

For any rectangle r we define the *crossing* of r in t, denoted by C_r to be the length of the maximal intersection between r and a horizontal segment in t. Let R be the set of rectangles in t. We define the crossing of R, denoted by C_R to be the sum $\sum_{r\in R} C_r$. Note that if R covers t', then the inequality $C_R \geq |\beta'|$ holds.

For every integer $i \geq 1$, we define m_i to be the maximum real number, such that there exists a top-coherent set R consisting of i rectangles, and the equality $C_R = m_i$ holds. Also, we define m_0 to be equal to the length of h_0. Let f be the least integer such that $m_f \geq |\beta'|$. Since the crossing of any set of rectangles covering t' is at least $|\beta'|$, it holds that the minimum number of rectangles needed to cover t' is greater than or equal to f. Hence, to prove the lemma it is sufficient to show that for any set R, such that $\#R = i < f-1$, it holds that $C_R \leq m_i \leq |h_0| \cdot c^i$, where $c = 1+\frac{20}{1-\tan\alpha}$. Since $m_0 = |h_0|$, it suffices to show the following statement:

(*) For any integer i, $0 \leq i \leq f-2$, it holds that $m_{i+1} \leq m_i \cdot (1+\frac{20}{1-\tan\alpha})$.

This assumption is not necessarily true for m_f, since $(m_{f-1} \cdot (1+\frac{20}{1-\tan\alpha}))$ could be greater than β. So, if $|\beta| < (m_{f-1} \cdot (1+\frac{20}{1-\tan\alpha}))$ then we let m_f be the largest possible rectangle in t that does not intersect the shell of t.

Let R be any top-coherent set of $i+1$ rectangles such that $m_{i+1} = C_R$. Let r be a rectangle in R whose uppermost corner is farthest down. Let h' be the

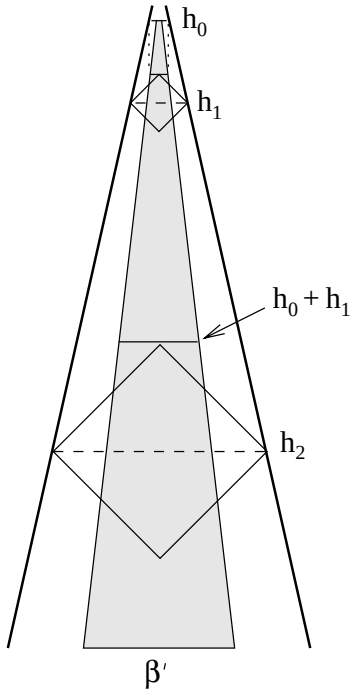

Figure 4.9:

horizontal segment passing through this corner and having its endpoints on the two arms of t', and let h be the horizontal segment overlapping h' and touching t's both arms. Hence, we have $20|h'| \geq h$. The crossing of r, C_r, is maximized if r is a maximal square touching h at its middle point, such that its sides form angles of 45 degrees with h. By straight-forward trigonometric calculations one can check that C_r is less than $\frac{h}{1-\tan\alpha}$. Since $20|h'| \geq |h|$, the following inequality holds:

(i) $C_r \leq \frac{h}{1-\tan\alpha} \leq \frac{20h'}{1-\tan\alpha}$.

To prove (*) we distinguish two cases.

Case 1: $i = 0$. In this case it holds that $m_1 = C_r$. Also, it holds that $|h| = m_0$. Combining this with inequality (i) we obtain $m_1 \leq \frac{m_0}{1-\tan\alpha}$, so statement (*) holds in this case.

Case 2: $1 \leq i \leq f-2$. Let R' be $R-\{r\}$. From the definition of crossing it follows that $C_R = C_{R'} + C_r$. Since R is top-coherent, every point on h' is included in some rectangle of R'. Hence it holds that $|h'| \leq C_{R'}$. Combining this with the latter equality and with (i) we obtain the inequality $C_R \leq C_{R'} + \frac{20C_{R'}}{1-\tan\alpha}$.

Since $m_{i+1} = C_R$, to show the statement (*) it remains only to show that $C_{R'} \leq m_i$.

Since R is top-coherent, it follows from the definition that every point in t' above, and including, h' is in some rectangle in R'. Hence, R' is also top-coherent, so the mentioned inequality $C_{R'} \leq C_i$ holds. This completes the proof of (*) and of the proposition. □

A rough idea

Later in this chapter, we will produce a set, L, of $k \cdot (\#F + \mu''(F))$ t-collections and t-local rectangles, where k is some constant independent of F, such that this set covers F. We will need the following crucial observation.

Observation 4.20 *In every weak visibility polygon it's possible to construct a path between any pair of points with three straight line segments.*

Proof: The observation follows directly from the definition of weak visibility polygon. □

Assume that there exists a subset of F, denoted N, such that every funnel in N is not completely covered by L. Our claim is that N has to be empty. One approach to show this would be to show, according to Observation 4.20, that for every funnel in N there exists a funnel in F, such that the two funnels cannot be connected by three line segments within P. Thus N has to be empty since the funnels in N cannot belong to the same weak visibility polygon as some funnels in F.

So, we want to find a "chain" of consecutive well-fanned funnels in F, such that every path between the first funnel, a funnel in $N \subset F$, and the last funnel, a funnel in $F \setminus N$, has to consist of more than three line segments within P. If for every funnel in N we can find a "chain" that fulfills the following two conditions, then we can prove the lemma.

(i) for every two consecutive funnels in this "chain", say t_1 and t_2, there exists a rectangle $r \in L$ that passes through t_1 and overlaps t_2 in such a way that r intersects β_{t_1} at a "moderate" distance from t_1's closest arm, to be defined below, and

(ii) the tops of every three consecutive funnels in this "chain" cannot be connected by less than two straight line segments within P.

Fulfilling the conditions

Let $t \in F$ be some funnel well-fanned by an approximately optimal set S and let r be a rectangle in S passing through t. The largest distance on β_t between t's

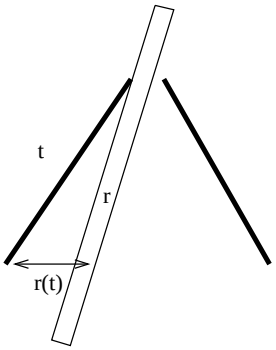

Figure 4.10: $r(t)$ denotes the largest distance on β_t between t's nearest arm and a point on the intersection between r and β_t.

nearest arm and a point on the intersection between r and β_t, is denoted $r(t)$, as shown in Fig. 4.10.

Definition 4.21 The part on β_t that lies more than $\frac{|\beta_t|}{10} + 5|\tau_t|$ from both t's arms is denoted t's *corebase*, see Fig. 4.11b.

Definition 4.22 A rectangle r is said to intersect β_t at a *moderate* distance from t's closest arm, if it holds that $3|\tau_t| < r(t) \leq \frac{|\beta_t|}{10} + 5|\tau_t|$.

Since t is well-fanned, there exist at least two rectangles in S that pass through t at a moderate distance from t's closest arm and overlap at least one funnel below t. So, for every funnel t well-fanned by an approximately optimal set S there exists a funnel below t, such that condition **(i)** holds. Condition **(ii)** is harder to fulfill, since the following problem can occur: if there exists a funnel $t_1 \in F$ well-fanned by S directly above any funnel $t_2 \in F$ whose top can (weakly) see t_2's entire corebase and t_1 is overlapped by a rectangle in S that also overlaps t_2, then it's possible that t_1's top can see all the funnels below t_2 that are overlapped by any rectangle that passes through t_2.

We have to guarantee that the tops of every triple of well-fanned funnels in the "chain", cannot be connected by less than two line segments within P. In Observation 4.23, which will be stated below, we show that there can be at most a linear number of funnel pairs, say t_1 and t_2, such that t_1 *dominates* t_2. A funnel t_1 is said to dominate a funnel t_2 if:

(1) t_1 is well-fanned by a set S,

(2) t_1 lies directly above t_2,

(3) t_1's top can see t_2's entire corebase, and

(4) there exists a rectangle r in S that passes through t_1 and overlaps t_2, where r intersects β_{t_1} at a moderate distance from t_1's closest arm, see Fig. 4.11a.

In Corollary 4.24 we note that we can replace (i.e., cover) the rectangles in t_1 that also overlap t_2 with a constant number of t-local rectangles. This will ensure that t_1 and t_2 won't be consecutive funnels in the "chain".

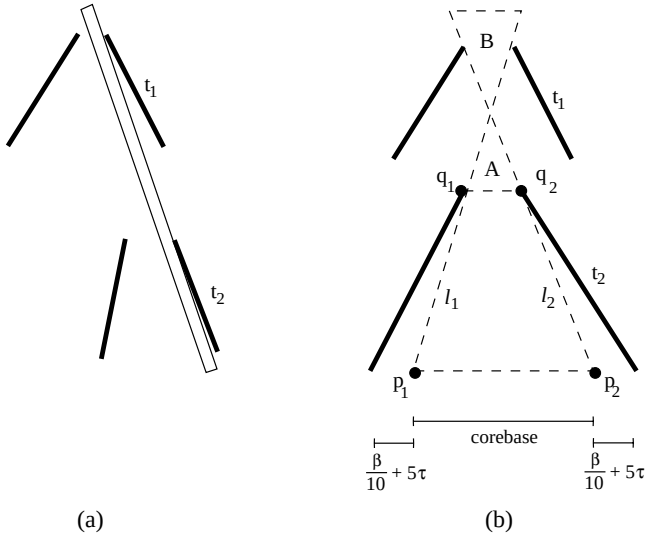

Figure 4.11: (a) t_1 may dominate t_2. (b) A funnel t_1 above t_2.

Observation 4.23 *For any funnel $t_2 \in F$ there exists at most one funnel $t_1 \in F$ which dominates t_2.*

Proof: Let p_1 and p_2 be the endpoints of t_2's corebase, and let p_1' and p_2' be the endpoints of τ_{t_2}, such that the lines l_1, the line that passes through the points p_1 and p_1', and l_2, the line that passes through p_2 and p_2', do not intersect within t_2. The lines l_1 and l_2 form two regions above t_2's top, the region above t_2's top and below the crossing is denoted A, and the region above the crossing is denoted B, as shown in Fig. 4.11b. Above the crossing (in region B) τ_{t_1} has to intersect the lines l_1 and l_2 to be able to see t_2's entire corebase. That is, at most one funnel t_1, that overlaps B, can lie directly above t_2 and see t_2's entire corebase.

Now, let r be any rectangle according to the observation. Every point in area A can see t_2's entire corebase, but since A's width is less than $\tau_{t_2} \leq 2\tau_{t_1}$ and r partly overlaps β_{t_1} at least $3\tau_{t_1}$ from t_1's arms, β_{t_1} will cut A in two, thus there

exists at most one funnel directly above t_2 that sees t_2's entire corebase. The observation follows. □

To fulfill condition **(ii)** we will later in the text need the following corollary, where we let E be any subset of F and R_E be a set of rectangles. Let t_1 and t_2 be any pair of funnels, denote by $Q(t_1, t_2)$ the quadrangle where τ_{t_1} and τ_{t_2} are opposite sides of $Q(t_1, t_2)$. Finally, let $Q'(t_1, t_2)$ be the intersection between t_1 and $Q(t_1, t_2)$

Corollary 4.24 *The number of t-local rectangles needed to cover $Q'(t_1, t_2)$, for every pair of funnels t_1 and t_2, where t_1 dominates t_2, is less than $3 \cdot \#E$.*

Proof: According to Observation 4.23, there are at most $\#E$ pairs of funnels where a funnel t_1 dominates a funnel t_2. Since the difference of the length of two funnel tops in E can be at most a factor two, we will only need three rectangles to cover $Q'(t_1, t_2)$, Fig. 4.12. The corollary follows. □

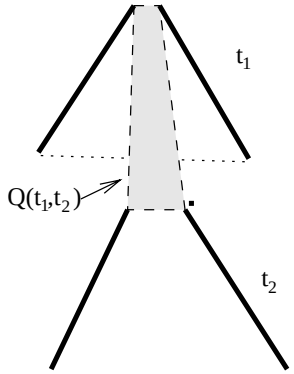

Figure 4.12: The part of the quadrangle $Q(t_1, t_2)$ in t_1 can easily be covered by a constant number of rectangles.

Thus it is possible to fulfill the two conditions needed to prove the lemma and at most add a linear number of t-local rectangles to any approximately optimal set S.

Finding the "impossible" zig-zag chains

We create an empty set L to which we will add t-local rectangles and t-collections. Our objective is that the set L eventually will cover all funnels in F, and that the number of t-local rectangles and t-collections in L should be at most

$k{\cdot}(\#F{+}\mu''(F))$, where k is some constant independent of F. Let L'' be a set of $\mu''(F)$ i-collections and i-local rectangles covering F, i.e., L'' is an optimal covering of F.

For every rectangle $r{\in}L''$ we insert a t-local rectangle in L'' such that the t-local rectangle overlaps the whole intersection between r and t_r, where t_r is the lowest funnel r overlaps. If we group these local rectangles into t-collections we will at most add $O(\#F{+}\mu''(F))$ t-collections to L'', according to Proposition 4.18.

For every pair of funnels t_1 and t_2 in F, where t_1 dominates t_2, insert three t-local rectangles in L'', such that the t-local rectangles cover $Q'(t_1, t_2)$. We will at most add $3{\cdot}\#F$ t-local rectangles to L'', according to Corollary 4.24.

Let H_1 be the subset of F such that every funnel $t{\in}H_1$ is well-fanned by L''. Recall that each funnel $t{\in}H_1$ has to be well-fanned by rectangles that pass through t, overlap at least one funnel in H_1 below t and the rectangles' intersection with t is not entirely covered by t-local rectangles or t-collections. Add the minimum number of t-collections and t-local rectangles needed to cover $\{F{-}H_1\}$ to L. According to Proposition 4.19 we have,

$$\#L = O(\#\{F - H_1\} + \mu''(\{F - H_1\})).$$

Let M_1 be the set of rectangles in L'' that pass through more than one funnel in H_1. For the sake of completeness we describe the procedure once again, this time with the sets M_1 and H_1 instead of L'' and F.

For every rectangle $r{\in}M_1$ we insert a t-local rectangle in M_1, such that the t-local rectangle overlaps the whole intersection between r and t_r, where t_r is the lowest funnel r overlaps. If we group these local rectangles into t-collections we will at most add $O(\#H_1{+}\mu''(H_1))$ t-collections to L''.

For every pair of funnels t_1 and t_2 in H_1, where t_1 dominates t_2, insert three t-local rectangles in L'', such that the t-local rectangles cover $Q'(t_1, t_2)$. We will at most add $3{\cdot}\#H_1$ t-local rectangles to M_1.

Let H_2 be the subset of H_1 such that every funnel $t{\in}H_2$ is well-fanned by M_1, and the intersection between t and the rectangles in M_1 is not entirely covered by the local rectangles produced above. Add the minimum number of t-collections and t-local rectangles needed to cover $\{H_1{-}H_2\}$ to L. We have,

$$\#L = O(\#\{F - H_2\} + \mu''(\{F - H_2\})).$$

Construct in the same way the sets $M_2, H_3, M_3, \ldots, H_7, M_7, H_8$. Note that H_8 is the set denoted N in the previous sections. We have that,

$$\#L = O(\#\{F - H_8\} + \mu''(\{F - H_8\})).$$

Proof (Lemma 4.15)

When all the sets $H_1, M_1, \ldots, M_7, H_8$ are constructed two cases can occur:
i) H_8 is empty.
We know that the set of rectangles, L, covers all the funnels in $\{F-H_8\}$ and $\#L = O(\#\{F-H_8\} + \mu''(\{F-H_8\}))$. Since H_8 is empty we have shown the lemma:

$$\sum_{t \in F} \mu''(t) = \#L = O(\#F + \mu''(F))$$

ii) H_8 is non-empty.
All the funnels in $\{F - H_8\}$ are covered by L, where $\#L = O(\#F+\mu''(F))$. We want a contradiction by showing that a funnel in H_8 cannot belong to the same weak-visibility polygon as some funnel in $F - \bigcup_{i=1}^{8} H_i$. For this we need the following straight-forward observation.

Observation 4.25 *Every possible path between two funnels in a simple polygon has to traverse all funnels traversed by the shortest path between them.*

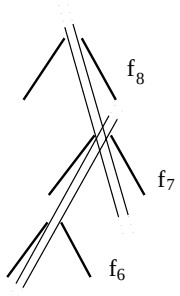

Figure 4.13: Constructing the zig-zag chain.

To show that the funnels in H_8 cannot belong to F, we will show that for every funnel in H_8 there exists at least one funnel in F, such that every path between these two funnels has to consist of at least four line segments.

Let $H_0 = F$ and let $M_0 = L''$. Let $M_{i-1}(f_i)$, $1 \leq i \leq 8$, be a subset of M_{i-1}, such that every rectangle $r \in M_{i-1}(f_i)$ passes through f_i and overlaps a funnel $f_{i-1} \in H_{i-1}$ below f_i in such a way that (1) r intersects β_{f_i} at a moderate distance from f_i's closest arm and (2) r's intersection with $\beta_{f_{i-1}}$ cannot be seen from f_i's top. (This is possible according to the definition of the rectangle sets M_{i-1}, $1 \leq i \leq 8$.) In the following steps if we use the rectangles from $M_{i-1}(f_i)$, $1 \leq i \leq 8$, to choose the funnels for the "chain", then we can guarantee that the tops of f_i

and any funnel below f_{i-1} in the "chain" cannot be connected by less than two line segments within P.

Consider a funnel $f_8 \in H_8$, see Fig. 4.13. We choose a rectangle in $M_7(f_8)$, as described in the previous paragraph, that overlaps a funnel $f_7 \in H_7$ below f_8. Such a rectangle exists according to the definition of M_7. In funnel f_7 we choose, in the same way, a rectangle in $M_6(f_7)$ that overlaps a funnel $f_6 \in H_6$ below f_7. (There exists such a rectangle according to the definition of M_6.) In f_6 we choose a rectangle in $M_5(f_6)$ that overlaps a funnel $f_5 \in H_5$ below f_6. We continue like this through the funnels $f_4 \in H_4, \ldots, f_1 \in H_1$ until we reach a funnel $f_0 \in H_0 = F$.

If we want to construct a path within P from f_8 to f_0, this path has to pass through the funnels $f_7, \ldots, f_1$, according to Observation 4.25. Any line segment that starts from funnel f_8 has to pass through f_7's top. We know that f_7's and f_5's tops cannot be connected by less than two line segments within P. The same holds for the funnels f_5 and f_3, so a path between f_8 and f_3 has to consist of at least three line segments within P. Since the same also holds for the funnels f_3 and f_1 we have that any path connecting f_8 and f_0 has to consist of at least four line segments within P. We have shown that H_8 has to be an empty set, since a funnel in H_8 cannot belong to F. This completes the proof of Lemma 4.15.

Note the following observation.

Observation 4.26 *It holds that*

$$\mu(F_i) = O(\#F_i + \mu''(F_i)).$$

Proof: Since $\mu(F_i) \leq \sum_{t \in F_i} \mu(t)$ and since $\sum_{t \in F_i} \mu(t) = O(\#F_i + \mu''(F_i))$, according to Lemma 4.15, the observation follows. □

From Observation 4.26 and since $\mu(t) = \Theta''(\mu(t))$ we may translate Lemma 4.15 to rectangles instead of collections and local rectangles, which gives us:

$$\sum_{t \in F} \mu(t) = O(\#F + \mu(F)). \tag{3}$$

4.3.2 Covering a normalized set of funnels

In the previous section we showed that an optimal covering of a sorted normalized set of funnels is not much better than a local covering produced by algorithm H. In this section our aim is to show that the number of rectangles produced to cover each of the sorted normalized sets $F_1, \ldots, F_s$ separately is not much more than the number of rectangles in an optimal covering of the normalized set $T = \{F_1, \ldots, F_s\}$, i.e., $\sum_{i=1}^{s} \mu(F_i) = O(\#T + \mu(T))$. The idea is to show that even if all rectangles in an optimal covering of T overlap many different sorted normalized sets, the total number of rectangles cannot be much less than

$\sum_{i=1}^{s} \mu(F_i)$, since the thickness of a rectangle is bounded by the length of the smallest funnel top it passes through.

Before the last part of the proof of the theorem, we need some definitions. These definitions describe what sorts of collections and rectangles that may overlap any normalized set of funnels.

Let R be a minimal set of rectangles that covers a normalized set of funnels T, and let R_i be a subset of R such that every rectangle in R_i overlaps with at least one funnel in F_i, where $F_1, \ldots, F_s$ is the sorted normalized set of funnels of T.

Definition 4.27 We define R_i' to be the set of all rectangles in R, such that $r \in R_i$ and $r \notin \bigcup_{j=1}^{i-1} R_j$, and r's thickness is less than or equal to l_i. Partition $R_i^{'}$ into a minimum number of collections of rectangles, which we will call D_i-collections. The total thickness of every D_i-collection is less than or equal to l_i. Note that in this case the total thickness of every D_i-collection, except for at most one, will be greater than $l_i/2$. Let D_i denote the set of D_i-collections and let d_i be the number of D_i-collections in D_i. Every rectangle in R_i whose thickness is greater than l_i is defined as an i-local rectangle in R_i. The number of i-local rectangles overlapping F_i is denoted b_i.

It may happen that some rectangles in various D_j-collections in D_j also overlap the funnel set F_i, where $j < i$. These are the rectangles in $\{R_i - R_i^{'}\}$ that are not i-local rectangles. Therefore we have to consider the case when the D_j-collections that overlap any of the funnel sets $F_1, \ldots, F_{i-1}$ also overlap (some of) $F_i, \ldots, F_s$. The thickness of a D_i-collection in D_i is at most l_i, consequently the total thickness of all the rectangles in the D_i-collections that may overlap all the subsequent funnel sets is $d_i \cdot l_i$.

Definition 4.28 We define L_i as follows: $L_i = L_{i-1} + d_{i-1} \cdot l_{i-1}$, where $L_1 = 0$. Thus, L_i is an upper bound on the total thickness of all rectangles in the j-collections, $1 \leq j < i$, that overlap any of the funnel sets $F_1, F_2, \ldots, F_{i-1}$.

Let $R_i^{''}$ be the set of all the rectangles in $\{R_i - R_i^{'}\}$ that are not i-local rectangles. We partition $R_i^{''}$ into a minimum number of sets of rectangles, which we will call i-free collections. The total thickness of every i-free collection is less than or equal to l_i. Note that the total thickness of every i-free collection, except for at most one, will be greater than $l_i/2$. Let g_i denote the number of i-free collections that overlap F_i, that is $g_i \leq \left\lceil \frac{L_{i-1}}{l_i/2} \right\rceil$. The total number of i-collections, i-free collections and i-local rectangles is denoted $\mu'(F_i)$. Since D_i-collections and i-free collections are just special cases of i-collections it holds that $\mu''(F_i) \leq \mu'(F_i)$. Recall that $\mu''(F_i)$ is the minimum number of i-collections and i-local rectangles needed to cover F_i.

Observation 4.29 *It holds that*

$$\mu(F_i) = O(\#F_i + \mu'(F_i)).$$

Proof: It holds that $\mu''(F_i) \leq \mu'(F_i)$ and, according to Observation 4.26, we have $\mu(F_i) = O(\#F_i + \mu''(F_i))$. The observation follows. □

Lemma 4.30 *Let T be a normalized set of funnels and let $F_1, \ldots, F_s$ be sorted normalized subsets of T as defined. Then it holds that:*

$$\sum_{i=1}^{s} \mu(F_i) = O(\#T + \mu(T))$$

Proof: According to Observation 4.29 it suffices to show that $\sum_{i=1}^{s} \mu'(F_i) = O(\#T + \mu(T))$. A funnel set F_i can be overlapped by at most g_i i-free collections, according to Definition 4.28. If a funnel set F_i is empty, g_i will naturally be 0. We will now calculate the maximum number of i-free collections overlapping F_i. Note that $l_{i+1} = 2 \cdot l_i$.

$$g_i \leq \left\lceil \frac{L_i}{l_i/2} \right\rceil = \left\lceil \frac{L_{i-1}}{l_{i-1}} \right\rceil + d_{i-1} =$$

$$= \left\lceil \frac{d_1}{2^{i-2}} + \frac{d_2}{2^{i-3}} + \ldots + d_{i-1} \right\rceil = \left\lceil \sum_{j=1}^{i-1} \left(\frac{d_j}{2^{i-j-1}} \right) \right\rceil.$$

It follows that the total number of free-collections will not exceed:

$$\sum_{i=1}^{s} g_i \leq \sum_{i=2}^{s} \left\lceil \sum_{j=1}^{i-1} \left(\frac{d_j}{2^{i-j-1}} \right) \right\rceil \leq \#T + 2 \sum_{i=1}^{s} d_i \tag{4}$$

According to Definition 4.28 we have:

$$\sum_{i=1}^{s} (d_i + b_i + g_i) = \sum_{i=1}^{s} \mu'(F_i) \tag{5}$$

From (4) we have that $\sum_{i=1}^{s} g_i \leq \#T + 2 \sum_{i=1}^{s} d_i$. Combining (4) and (5), we get:

$$\sum_{i=1}^{s} \mu'(F_i) \leq \#T + 3 \sum_{i=1}^{s} d_i + \sum_{i=1}^{s} b_i \tag{6}$$

From Definition 4.27 we have that every rectangle in the optimal covering of T will be counted as a local rectangle in at most four different funnel sets F_i (one for each corner), where $1 \leq i \leq s$, that is $\sum_{i=1}^{s} b_i \leq 4\mu(T)$. From the same

definition we have that $\sum_{i=1}^{s} d_i \leq \mu(T)$ which together with (6) gives us the result $\sum_{i=1}^{s} \mu'(F_i) \leq 7\mu(T) + \#T$, which at last gives us:

$$\sum_{i=1}^{s} \mu'(F_i) = O(\#T + \mu(T)).$$

□

4.3.3 Putting it all together

We conclude the proof of Theorem 4.7. If we use (3) for a normalized set of funnels T, we get:

$$\sum_{t \in T} \mu(t) = O(\#T + \sum_{i=1}^{s} \mu(F_i)). \tag{7}$$

If we look at the result from Lemma 4.30 we get that (1) follows directly from (7). Since $\#T = O(n)$, $\mu(T) \leq \mu(P)$ and $H(P) = O(n + \sum_{t \in P} \mu(t))$ (Fact 4.6) we obtain the final result

$$H(P) = O(n + \mu(P)).$$

4.4 A tight lower bound on optimal coverings

Let $\alpha(n)$ denote the extremely slowly growing inverse of Ackermann's function [94].

Theorem 4.31 *For all integers $n \geq 4$ and for every simple polygon P with n vertices it holds that $\mu(P) = \Omega(\frac{n}{\alpha(n)})$.*

Proof: Hart and Sharir [45] showed that the Davenport-Schinzel sequence of order 3 has length $O(m \cdot \alpha(m))$. Let P be any polygon with n vertices, and let M be a set of $\mu(P)$ rectangles that covers P. The sequence E of indices of segments in M in the order in which they appear along the outer face of the rectangles is a Davenport-Schinzel sequence of order 3 - i.e., no two adjacent elements of E are equal and E contains no subsequence of the form $a \ldots b \ldots a \ldots b \ldots a$. Hence, P consists of $O(\mu(P) \cdot \alpha(\mu(P)))$ edges, since the number of segments is $4\mu(P)$. Thus, $\mu(P) = \Omega(\frac{n}{\alpha(\mu(P))})$ holds. By considering the two cases (1) $\mu(P) \leq n$ and, (2) $\mu(P) > n$ it follows that $\mu(P) = \Omega(\frac{n}{\alpha(n)})$.

□

Theorem 4.32 *The lower bound shown in Theorem 4.31 cannot be generally improved, i.e., for each n there is a polygon P with n vertices such that $\mu(P) = O(\frac{n}{\alpha(n)})$.*

Proof: According to Wiernik [94] there exists a construction of a set M, of m straight-line segments, such that the lower envelope Y of M consists of $\Omega(m \cdot \alpha(m))$ subsegments.

Construct k non-vertical connected edges, $E = \{e_1, \ldots, e_k\}$, such that the edges in E are identical to a lower envelope produced by the m segments in [94]. Let e_i be the i:th left-most edge of E and let d denote the largest vertical distance between two points in E. Expand E along the x-axis, by multiplying all x-coordinates such that the minimum difference along the x-axis between two incident vertices in E is $10d$. Let e_0 and e_{k+1} be two horizontal edges of length $10d$, such that e_0 is connected to the left endpoints of e_1 and e_{k+1} is connected to the right endpoint of e_k. Let E' be the set of edges in E plus e_0 and e_{k+1}. Now, construct an upside-down histogram P, with one horizontal upper long-side, denoted s_t, and two vertical edges s_l and s_r of length at least $10d$, such that s_l connects s_t's left endpoint with e_0's left endpoint and s_r connects s_t's right endpoint with e_{k+1}'s right endpoint, Fig. 4.14.

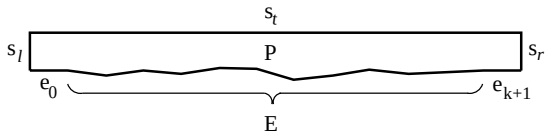

Figure 4.14: It is possible to cover P with $O(n/\alpha(n))$ rectangles.

We will now prove that it is possible to cover P with $O(\frac{n}{\alpha(n)})$ rectangles. Let r_{s_t} be a maximal rectangle whose upper side coincides with s_t. The region of P covered by r_{s_t} is denoted P_T and the uncovered area of P is denoted P_E. Let r_{s_l} and r_{s_r} be two maximal rectangles, within P, such that r_{s_l} includes s_l and e_0, and r_{s_r} includes s_r and e_{k+1}. Now, let R_E be a minimum set of maximal rectangles, within P, such that every edge of E coincides with at least one rectangle in R_E, and the length of the short sides of every rectangle in R_E is $2d$. Let $R_{E'}$ be the set of edges in R_E plus r_{s_l} and r_{s_r}. We now show that the rectangles in R_E cover the remaining uncovered part of P_E, i.e., $(P_E - (r_{s_l} \cap P_E) - (r_{s_r} \cap P_E))$. Note the following facts:

1. the largest slope of an edge in E' is $1/10$,
2. the shortest distance between two non-incident edges in E' is $10d$, and
3. the shortest distance between s_t and an edge in E is $9d$.

It is easily seen that every point in P_E with a perpendicular projection on any edge of E' is covered by the rectangles in $R_{E'}$, see Fig. 4.15a. Note that every

remaining uncovered region lies closer than d to a concave vertex of E'. Thus it remains to show that every point in P_E, closer than d to a concave vertex of E', is covered by the rectangles in $R_{E'}$.

Let v be a concave vertex connecting e_i and e_{i+1}, $0 \leq i \leq k$, in E', and let r_{e_i} and $r_{e_{i+1}}$ be the two rectangles in $R_{E'}$ whose bases include e_i, respectively e_{i+1}, Fig. 4.15b. According to the above facts, we have that r_{e_i} and $r_{e_{i+1}}$ extend at least $10d$ to the right, respectively to the left of v, and since the short sides of the rectangles in $R_{E'}$ have length $2d$, we have that r_{e_i} and $r_{e_{i+1}}$ cover the part of P_E closer than d to v. Hence $(R_E \cup r_t \cup r_{s_l} \cup r_{s_k})$ covers P_E and $\mu(P) = O(\#R_E)$. According to [94] we have that the number of segments in E is $\Omega(\#R_E \cdot \alpha(\#R_E)) = k(= n-5)$. Let $Y(R_E)$ be the lower envelope of R_E. Since $\#Y(R_E) = \#E$, according to the definition of E, we have that the number of rectangles in R_E is $O(\frac{\#E}{\alpha(\#E)})$, and hence, $\mu(P) = O(\frac{n}{\alpha(n)})$. □

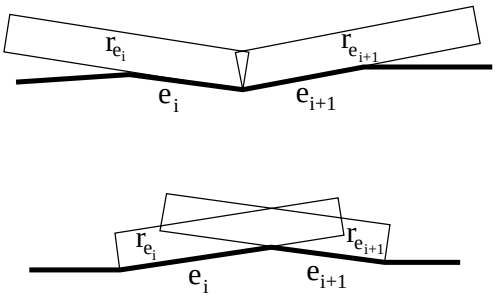

Figure 4.15:

Summarizing the main result of this paper, we have shown that

$$H(P) = O(min[\alpha(n) \cdot \mu(P), n + \mu(P)])$$

and combining with Lemma 4.6 we obtain the following theorem.

Theorem 4.33 *For any simple polygon P, with n vertices, Algorithm H produces a rectangular covering of P within an $O(\alpha(n))$ approximation factor from the minimum in optimal time $O(n+\mu(P))$.*

Chapter 5

Lower bounds for approximate polygon decomposition and minimum gap

In Chapter 4 we presented the approximation algorithm by Levcopoulos, which had a running time of $O(n \log n + \mu(P))$. In the same chapter we showed that the time-complexity of the algorithm could be improved to $O(n + \mu(P))$ in the case when P is a simple polygon. Similar behavior can be found for various covering problems. In the case when the given polygon is rectilinear Franzblau [40] gave an approximation algorithm with running time $O(n \log n)$ and approximation ratio $\log n$. In 1996, Bar-Yehuda and Ben-Hanoch [13] gave a linear time algorithm for the simpler hole-free case. This led us to study how the trivial linear lower bound can be improved in the case when the given input polygon contains holes. We prove the separation suggested by these results, i.e., we show that any approximation algorithm for covering an arbitrary polygon, with holes, with a finite number of polygons has a lower bound on the time-complexity of $\Omega(n \log n)$ in the algebraic decision tree model.

Next we give the general idea and show how the original problem of covering a polygon can be reduced to the simpler problem of finding a lower bound (greater than zero) on the minimum difference between two consecutive numbers in a set of n numbers. In Section 5.2 we show, by using the results of Ben-Or [14], that the time needed to solve the problem is $\Omega(n \log n)$.

The main result of this chapter is:

Theorem 5.1 *Every algorithm which for a polygon with holes on n vertices computes a decomposition into L (orthogonal) rectangles requires $\Omega(n \log n + L)$ time.*

Note that the theorem makes no assumptions on the quality of the solution, e.g., the approximation ratio of the algorithm. However, for algorithms that produce coverings of large cardinality (for example algorithms with approximation ratio worse than $O(\log n)$) the statement trivializes because the output must be reported.

The model underlying Theorem 5.1 is the algebraic computation tree model [81], which is the standard model for low-level lower bounds in computational geometry. Interestingly, because of the algorithm's non-exactness there seems to be no standard reduction that establishes the theorem. We believe that we in this chapter provide a canonical hard problem (computational prototype) that may be useful for lower bounds for other approximation algorithms.

By a reduction from sorting, Liou *et al.* [65] show an $\Omega(n \log n)$ lower bound for the *optimal* partition problem assuming that the polygon has holes. The best upper bound for this problem is $O(n^{3/2} \log n)$ [90]. The proof can be seen to work for the optimal covering problem as well, for which not even a polynomial-time algorithm is known (it is NP-complete [28]). Aggarwal *et al.* [2] present an $\Omega(n \log n)$ lower bound for another minimum gap problem on a simple polygon, but also that result is only relevant for the exact setting.

5.1 Improving the minimum gap

For $S = \{x_1, \ldots, x_n\}$, let

$$MinGap(S) = \min_{i \neq j} |x_i - x_j|$$

be the *minimum gap* of S.

Problem 1: (ImproveGap) Given a set S of real numbers and a real number ϵ with $0 \leq \epsilon < MinGap(S)$. Find a real number b such that $\epsilon < b \leq MinGap(S)$.

We prove in the next section that this problem is hard, that is:

Theorem 5.2 *In the algebraic computation tree model, every algorithm for* ImproveGap *needs time $\Omega(n \log n)$ for any values of ϵ, $0 \leq \epsilon < MinGap(S)$.*

The next reduction proves Theorem 5.1.

Lemma 5.3 *If there is an algorithm of time complexity $T(n)$ that decomposes a given polygon with n vertices into L orthogonal rectangles then* ImproveGap *can be solved in time $O(T(n) + n + L)$.*

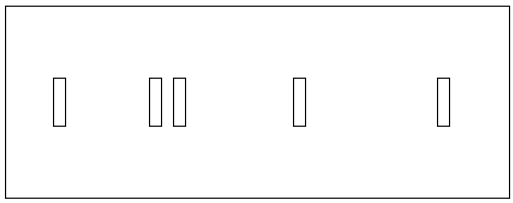

Figure 5.1: A polygon constructed from the instance $S = \{1, 3, 3.5, 6, 9\}$.

Proof: We assume without loss of generality that the values in a given instance S to ImproveGap are positive. We construct a polygon P as follows. Let $h = |x_1 - x_2|$, $x_{\min} = \min_i x_i$ and let $x_{\max} = \max_i x_i$. These values can be computed in linear time. The outer perimeter is a rectilinear rectangle with lower left corner at $(x_{\min} + \epsilon, 0)$ and upper right corner at $(x_{\max}, 2h + 1)$. For each x_i we place a rectangle of height 1 and width ϵ at (x_i, h), Fig. 5.1. This construction takes time $O(n)$. The algorithm for the decomposition problem returns a collection $\mathcal{C}$ of L rectangles, we can assume that none of these have zero width. Let $w > 0$ be the width of the narrowest rectangle in $\mathcal{C}$, this can be found in time $O(L)$. It is obvious that $w + \epsilon$ is a lower bound on $MinGap(S)$, which is greater than ϵ. □

One might be tempted to use a simpler problem than ImproveGap for the reduction, for example Min Gap or Uniqueness. A reduction from Min Gap is not appropriate, since we do not need an exact distance for approximation algorithms. Min Gap might be a suitable problem if we would consider an optimal algorithm, and not any approximation algorithm. The Uniqueness-problem cannot be considered since we do not allow the gap between two holes to be 0, and if the distance between two holes were zero then this "gap" would not be considered. Hence, neither the Uniqueness-problem nor the Min Gap-problem is suitable for the reduction.

5.1.1 Comments on input representation

Our results reinforce the literature's dichotomy between polygons with holes and without holes. However, we want to emphasize the fact that this is mainly a question about how the instance is represented. Indeed, the literature's linear-time algorithms exploit a feature in the representation of the input, not in the topology of the polygon.

First, it may be noted that our reduction relies crucially on a detail in the input representation, namely that the holes in P are not ordered from left to

right. Were they ordered, the gap problem presented in our reduction could be easily solved in linear time.

The presence of an ordering in hole-free polygons and its absence in the general case is a well-accepted assumption in the literature; the common input representation is a doubly-connected edge list of the edges constituting the perimeter of the polygon, sorted by their ordering on the perimeter. Thus there is a natural ordering of the edge list of a hole-free polygon. Indeed, the known linear-time algorithms for decomposing hole-free polygons rely on the input being presented as a sorted edge list, for example by using Chazelle's linear-time triangulation algorithm [23].

For polygons with holes on the other hand, there is no such ordering. Our reduction exploits this lack of information to solve a gap problem.

To illustrate this further we consider the covering problem with a different input representation.

Problem 2: PARTITIONFROMPOINTS For a set of n points that are the corners of a rectilinear, simple polygon P, compute a partition of P.

We remark that indeed such a polygon is uniquely described by its corners, so the problem is well-defined (assuming that every edge is maximal in the sense that no two edges can be replaced by a single edge). We show that even though this polygon is *hole-free*, it does not admit a linear time decomposition, no matter how bad. Contrast that with the result that the exact partition problem for hole-free polygons can be solved in linear time [65], if they are represented as an ordered list.

Proposition 5.4 *Every algorithm for* PARTITIONFROMPOINTS *runs in time* $\Omega(n \log n)$.

Proof: (Sketch) Given an input $x_1, \ldots, x_n, \epsilon$ to PARTITIONFROMPOINTS, let (as in the proof of Lemma 5.3) $h = |x_1 - x_2|$, $x_{\min} = \min_i x_i$ and let $x_{\max} = \max_i x_i$. The points of P are

$$\{(x_{\min} + \epsilon, 0), (x_{\min} + \epsilon, h + 1), (x_{\max}, 0), (x_{\max}, h + 1)\} \cup$$

$$\cup \bigcup_{i=1}^{n} \{(x_i, h + 1), (x_i, h), (x_i + \epsilon, h), (x_i + \epsilon, h + 1)\},$$

as shown in Fig. 5.1. □

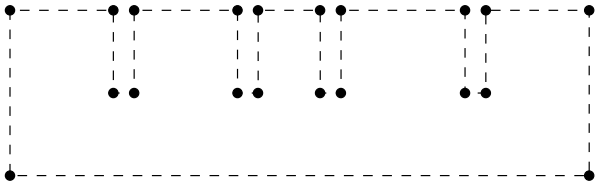

Figure 5.2: Reduction from PARTITIONFROMPOINTS.

5.2 Proof of Theorem 5.2

For the lower bound we introduce a decision version of the minimum gap problem, whose definition depends on an algorithm A for IMPROVEGAP. Let

$$N = \{\, (x_1, \ldots, x_n) \mid \{x_1, \ldots, x_n\} = \{1, \ldots, n\} \,\},$$

be the set of permutations of $(1, \ldots, n)$. Let $B_A = \min_{x \in N} A(x)$. Since N is finite and $A(x) > 0$ we have $B_A > 0$. Let

$$Y_A = \{\, x \in E^n \mid A(x) < B_A \,\},$$

where E^n denotes n-dimensional Euclidean space. Note that by construction, $Y_A \cap N = \emptyset$.

Problem 3: (DECIDEGAP$_A$) Given a set $S \in E^n$. If $S \in Y_A \cup N$ then decide if $MinGap(S) < B_A$, otherwise the output is undefined.

Lemma 5.5 *Let A be an algorithm for* IMPROVEGAP *with running time $T(n)$. Then* DECIDEGAP$_A$ *has an algorithm with running time $T(n) + O(1)$.*

Proof: By construction, all Y_A are 'yes'-instances to DECIDEGAP$_A$ and all N are 'no'-instances. Thus the algorithm for DECIDEGAP$_A$ runs A and returns true if and only if $A(x) < B_A$. □

It remains to prove that DECIDEGAP$_A$ is hard, which is a standard argument. We use the following result:

Theorem 5.6 *[14] Let W be a set in Cartesian space E^n and let T be an algebraic decision tree of fixed order $d \geq 2$ that solves the membership problem in W. If h is the depth of T and $\#W$ is the number of disjoint connected components of W, then $h = \Omega(\log \#W - n)$.*

Lemma 5.7 *Let A be any algorithm for the* IMPROVEGAP. *In the algebraic computation tree model, every algorithm for* DECIDEGAP$_A$ *requires $\Omega(n \log n)$.*

Proof: We will prove the result for the complementary problem. To show that each of the $n!$ points in N lies in its own component we merely observe that every curve connecting two different points $x = (x_1, \ldots, x_n)$ and $x' = (x'_1, \ldots, x'_n)$ in N must pass through a point whose minimum gap is less than B_A and thus belongs to Y_A.

Let x_i be the smallest element in x such that $x_i \neq x'_i$. Then there must exist a $x_j > x_i$ such that $x'_j = x_i$. We may now reduce this problem to two dimensions. Any curve connecting x and x' must include a point (z, z), where z lies in the interval $[x'_i, x_i] \cap [x_j, x'_j]$. The lemma follows. □

Thus we have proved Theorem 5.2.

5.2.1 Hardness of approximating the minimum gap

In this section we show that it is hard to approximate $MinGap(S)$.

The second smallest gap of S is defined as $MinGap_2(S) = \min_{i \neq j}(\{|x_i - x_j|\} \setminus MinGap(S))$. Obviously, $MinGap(S)$ is in the interval $[0, MinGap_2(S)]$. The next problem is to find a smaller interval containing $MinGap(G)$.

Problem 4: (IntervalMinGap) Given a set S of real numbers and the value $MinGap_2(S)$, find an interval I such that $MinGap(S) \in I \subset [0, MinGap_2(S)]$.

The argument from the previous section can be applied to prove the following hardness result.

Proposition 5.8 *In the algebraic computation tree model, every algorithm for the* IntervalMinGap *requires time* $\Omega(n \log n)$.

The above result is very strong and implies the hardness of the well-studied MinGap problem in almost every approximative sense. As an example of a natural approximation formulation of MinGap we propose the following.

Problem 5: ((c_1, c_2)-ApproximateMinGap) Given a set S of real numbers find a value δ such that

$$\delta / c_1 \leq MinGap(G) \leq \delta c_2$$

Proposition 5.9 *In the algebraic computation tree model, every algorithm for* (c_1, c_2)-ApproximateMinGap *requires time* $\Omega(n \log n)$, *for every* $c_1, c_2 > 1$.

The main theorem of this chapter concerning the problems we have considered in the previous three chapters is:

Theorem 5.10 *Let P be a polygon with holes. The problem of covering P with a set of L simpler polygons, for example, squares or rectangles, requires time* $\Omega(n \log n + L)$.

Chapter 6

TSP with neighborhoods

A salesman wants to meet some potential buyers. Each buyer specifies a region in the plane, his *neighborhood*, within which he is willing to meet. The salesman wants to find a tour of shortest length that visits all of the buyers' neighborhoods and finally returns to his initial departure point. The Traveling Salesman Problem with neighborhoods (TSPN) asks for the shortest tour that visits each of the neighborhoods. The problem generalizes the Euclidean Traveling Salesman Problem in which the areas specified by the buyers are single points, and consequently it is NP-hard [41, 80]. For the Euclidean TSP there is now a polynomial-time approximation scheme, allowing, one, for any fixed $\epsilon>0$, to get within a factor $(1+\epsilon)$ of the optimal in time $n^{O(1/\epsilon)}$, see [9] and [71].

One can think of the TSPN as an instance of the "One-of-a-Set TSP" also known as the "Multiple Choice TSP", and the "Covering Salesman Problem". This problem is widely studied for its importance in several applications, particularly in communication network design [47] and VLSI routing [84].

Arkin and Hassin [7] gave an $O(1)$-approximation algorithm for the special case in which the neighborhoods all have diameter segments that are parallel to a common direction, and the ratio between the longest and the shortest diameter is bounded by a constant. Recently, Mata and Mitchell [70] provided a general framework that gives an $O(\log k)$-approximation algorithm for the general case, where k is the number of neighborhoods, with polynomial time complexity $\Omega(n^5)$ in the worst case. In this chapter we give several results: First we show a simple algorithm that produces a TSPN tour with logarithmic approximation factor in the case when a start point is given. If no start point is given we show how a "good" start point can be computed in $O(n^2 \log n)$. Hence, by combining these two results we obtain a logarithmic approximation algorithm for the general case with running time $O(n^2 \log n)$. Our main result is an algorithm that, given an

arbitrary real constant ϵ as an optional parameter, performs at least one of the following two tasks (depending on the instance):

1. It outputs in time $O(n \log n)$ a TSPN tour guaranteeing that it is of length $O(\log k)$ times the optimum.

2. It outputs in time $O(n^3)$ a TSPN tour guaranteeing that it is of length at most $(1+\epsilon)$ times the optimum.

The first part of our method builds upon the idea in [70], in that our logarithmic approximation algorithm produces a guillotine subdivision. However, we produce a quite different guillotine partition (partly inspired from [43, 59]) and show that it has some nice "sparseness" properties, which guarantee the $O(\log k)$ approximation bound.

The method described in this chapter can also be applied to other problems as suggested in [70]. Here we will also consider the Red-Blue Separation Problem (RBSP). The RBSP asks for the minimum-length simple polygon that separates a set of red points from a set of blue points, a problem shown to be NP-hard [8, 35]. Mata and Mitchell showed that the general framework presented in [70] gives an $O(\log m)$-approximation algorithm for this problem in time $O(n^5)$, where $m<n$ is the minimum number of sides of a minimum-perimeter rectilinear polygonal separator for the points. By using the methods suggested in this chapter we obtain an $O(\log m)$ approximation algorithm that runs in time $O(n \log n)$. Finally we show that TSPN is APX-hard and cannot be approximated within a factor of 1.000374 unless P=NP, using an idea from Kann [50].

This chapter is organized as follows. In Section 6.2 the approximation algorithm is presented for the TSPN case where a start point is given. First we compute a bounding square in linear time that includes or touches all neighborhoods. Next, a guillotine subdivision algorithm operating within this bounding square is presented that runs in time $O(n+k \log k)$. Then we prove that the subdivision (together with the bounding square) is of length $O(\log k)$ times the length of an optimal tour. In Section 6.3 we show how to compute start points that are included in TSPN-tours of length within a constant factor times the optimal in time $O(n^2 \log n)$. In Section 6.4, we describe an algorithm that, depending on the given input, decides which method to apply to obtain a TSPN tour. We also present the ideas behind the approximation scheme that can be used for some instances of TSPN. We also briefly show, in Section 6.5, how the methods presented in Section 6.2-6.3 can be used to obtain approximation algorithms for some separation problems.

The algorithms presented in Section 6.2-6.3 and in Section 6.5 are easy to implement and fast in practice [76].

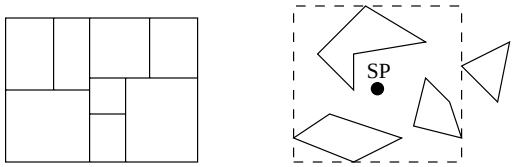

Figure 6.1: (a) A guillotine rectangular subdivision. (b) Constructing a bounding square.

6.1 Definitions and preliminaries

Throughout this chapter we will denote by X the given collection of k possibly overlapping simple polygons, called *neighborhoods*, with totally n vertices in the plane. We avoid dealing explicitly with degenerate neighborhoods, like points or segments, but our results are easy to extend to such neighborhoods. A way to incorporate degenerate parts in the algorithms implicitly would be to treat them as infinitesimally thin polygons. For example, single points could be treated as infinitesimally small squares.

A polygonal subdivision D of a polygon P is said to be "guillotine" if it is a binary planar partition of P, i.e., either there exist no edges of D interior to P, or there exists a straight edge of D such that this edge is a chord of P dividing P into two regions, P_1 and P_2, such that D restricted to P_i, $1 \leq i \leq 2$, is a guillotine subdivision of P_i. If all the faces of the subdivision are rectangles, we obtain a guillotine rectangular subdivision, see Fig. 6.1a. From now on we will denote by GRS the guillotine rectangular subdivision obtained from the procedure described in Section 6.2.1. We will denote by $|p|$ the total length of the segments of p, where p may be a polygon, a tour or a subdivision.

6.2 A fast approximation algorithm for TSPN

This section is devoted to the proof of the following theorem.

Theorem 6.1 *Let X be a collection of k possibly overlapping simple polygons, having a total of n vertices. Let* SP *be any start point. One can compute in time $O(n+k\log k)$ a* TSPN *tour starting at* SP*, whose length is $O(\log k)$ times the length of a minimum-length tour starting at* SP*.*

The general idea is very simple. Compute a bounding square that includes or touches all neighborhoods. Next a binary partition is performed within the bounding square such that every neighborhood is intersected either by the binary

partition or the bounding square. By traversing the bounding square and the binary partition we obtain a TSPN tour. Below we show how to do this in some more details.

First we detect each neighborhood whose closed region contains SP. This is done in linear time using standard techniques (see, e.g. [81]). These neighborhoods are visited without departing from SP, and we may exclude them from further consideration. Thus, we may assume w.l.o.g. in the continuation of this section that SP is disjoint from all neighborhoods in X. The next step is to compute the minimal isothetic square, called *bounding square*, $\mathcal{B}$, with center at SP such that each neighborhood is at least partially contained in or touched by $\mathcal{B}$. This bounding square can be constructed in linear time by computing, for each neighborhood $x_i \in X$, $1 \leq i \leq k$, the minimum isothetic square centered at SP and reaching x_i, and then taking the maximum over all these squares, Fig. 6.1b.

If all the polygons intersect the bounding square then we output the tour obtained by walking from SP to the bounding square, then following the perimeter of the bounding square and returning to SP. This tour is at most five times longer than an optimal TSPN-tour. (In an optimal tour, the salesman has to reach the boundary of $\mathcal{B}$ and then return.) Otherwise, one or more neighborhoods lie entirely inside $\mathcal{B}$. In this case we construct a rectangular binary planar partition, which we call GRS, of the bounding square $\mathcal{B}$, such that every polygon lying entirely within $\mathcal{B}$ is intersected or touched by the GRS. In [70], Mata and Mitchell, described how a guillotine subdivision and its bounding box can be traversed such that the obtained tour visits all neighborhoods in X and has length at most twice the length of the subdivision plus $|\mathcal{B}|$.

To prove Theorem 6.1 it remains to define the GRS, show how it can be computed in time $O(n+k \log k)$ (Section 6.2.1), and prove that its length is within a logarithmic factor longer than the length of a shortest tour (Section 6.2.2).

6.2.1 The guillotine rectangular subdivision

In this subsection we first describe how the subdivision algorithm works. Then, we show how this algorithm can be implemented to run in time $O(n+k \log k)$.

The partition is done by drawing the chords $e_1, e_2, \ldots$ within $\mathcal{B}$. Let T_j be the rectangle in the subdivision induced by all segments drawn by the procedure up to but not including the segment e_j, such that e_j is drawn in T_j during the next step. Let A, B, C and D be the corners of T_j in clockwise order, such that $\overline{AB}$ is the top side of T_j. The segment e_j is drawn parallel to a shortest side of T_j, and it is vertical if T_j is a square. Assume for simplicity that $|AD| \leq |AB|$, i.e., e_j will be a vertical segment, as shown in Fig. 6.2a.

Let e be the vertical chord splitting T_j into two rectangles of equal size. Let

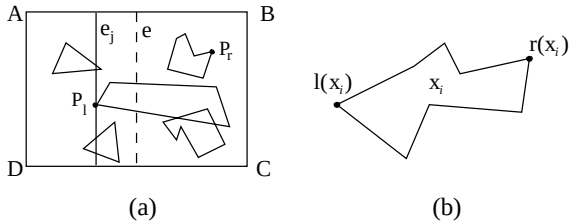

Figure 6.2: (a) A guillotine cut e_j. (b) $l(x_i)$ and $r(x_i)$ of a polygon x_i.

$N=\{x_1,\ldots,x_{k'}\}$ be the set of k' polygonal neighborhoods entirely within T_j, and let $l(x_i)$ respectively $r(x_i)$, $1\leq i\leq k'$, be the leftmost, respectively rightmost, point in the polygonal neighborhood x_i, see Fig. 6.2b. Two cases may occur:

1. If the region to the left of e is empty then the procedure draws e_j such that it intersects the leftmost point $r(x_i)$, $1\leq i\leq k'$. The corresponding procedure is performed symmetrically if the region in T_j to the right of e is empty.

2. If none of the regions are empty then let N_l, respectively N_r, be the polygons in N with non-empty intersection with the closed region of T_j to the left, respectively to the right, of e. (If, for example, e is tangent to a neighborhood, then this neighborhood is in both N_l and N_r.) Let P_l be the point in $\{\forall x\in N_l \quad l(x)\}$ with shortest distance to e (ties are broken arbitrarily). Symmetrically, P_r is the point in $\{\forall x\in N_r \quad r(x)\}$ with shortest distance to e. If the horizontal distance between P_l and e is shorter than the horizontal distance between P_r and e, then the procedure draws the vertical chord e_j such that it passes through P_l. Otherwise, it draws the vertical chord e_j such that it passes through P_r.

This is done recursively within the two resulting rectangles until all the polygonal neighborhoods are intersected by the subdivision. The final result is called the GRS.

Lemma 6.2 *The guillotine algorithm presented above can be implemented to run in time* $O(n+k\log k)$.

Proof: The algorithm only has to consider the extremal points of each neighborhood that lie entirely within the bounding square. Hence, we compute the minimal isothetic bounding rectangle for each polygon, i.e., the upper left corner and the lower right corner. We say that these points are *mates*. This step takes linear time to perform. Note that we now only have $2k$ points to consider.

The main algorithm is a refinement and generalization of the ideas in [59]. It maintains a set SEG of produced guillotine segments and a set REC of records, one record for each non-empty rectangle of the partition which remains to be processed. Every record in REC contains the coordinates of the rectangle which has to be processed, and pointers to the extreme elements of two doubly-linked lists, one *vertical* and one *horizontal*. The vertical resp. horizontal list pointed to from the record in REC contains the points within the rectangle stored in the record, sorted according to their x- respectively y-coordinates. In addition, there are double links between every element in the vertical list with the corresponding element (for the same point) in the horizontal list, so that when an element is removed from one of the lists it can be removed in constant time from the other list too. Finally there are double links between points that are mates in the vertical respectively horizontal list.

Initially there is one record in REC, corresponding to the bounding square, and SEG is the empty set. Let R'_0 be the rectangle with $k'=2k$ points. For $1\leq i\leq t$ (t will be defined in a moment), let R_i, R'_i, u_i and u'_i be defined as follows. Let R_i and R'_i be the two subrectangles of R'_{i-1} in the partition induced by the first segment within R'_{i-1} produced according to the definition of GRS. Among these two subrectangles, let R_i be the one with the least number of points, that is, not considering the points belonging to the neighborhoods which are intersected by segments produced earlier. Let u_i and u'_i be the number of active points in the open region bounded by R_i, respectively R'_i. Let t be the least integer such that $u_t=u'_t=0$.

We have that $k' \geq u_1+u'_1+1 = u_1+u_2+u'_2+2 = \ldots = t+\sum_{i=1}^{t} u_i$. Moreover, for every $1\leq i\leq t$, we have $u_i \leq \lceil k'/2^{i-1}\rceil -1$. The algorithm, computes R_i, R'_i and the horizontal and vertical lists for R'_i from R'_{i-1} in time $O(u_i)$. If $i<t$, then the algorithm continues to compute R_{i+1} and R'_{i+1} from R'_i. In this way the entire sequence $R_1,\ldots,R_t$ is computed in total time $O(k')$, and the corresponding partitioning segments are inserted into SEG. To build the vertical and horizontal lists for all R_i, before starting the computation of the rectangles, one copy of the vertical list and one of the horizontal are constructed. Let us call the original lists *master* lists and the copies *working* lists. While copying, additional links are added between the master and working lists, such that for every element in the working list, there is a pointer to the corresponding element in the master list. In the continuation, until R_t is computed, only the structure of the working lists changes. When R_i is computed the elements corresponding to the u_i points inside R_i are deleted from the working lists, and the corresponding elements in the master lists are labeled with the integer i. If a point and it's mate do not lie in the same open region, then the elements corresponding to the point and its mate are removed both from the working lists and master lists.

When R_t is computed, then all points inside R'_0 which do not lie on already produced segments have been labeled with some integer. Then the horizontal and vertical lists for all rectangles R_i, can be computed in total time $O(k')$ as follows. For easier access, an array with $k'-1$ entries is built, such that in the i-th entry, information is stored concerning the lists for the rectangle R_i. For example, to build the horizontal lists, they are first initialized to be empty. The horizontal master list is then scanned, say, from the leftmost to the rightmost point, and every labeled element is appended at the end of the horizontal list of the appropriate rectangle, by using the corresponding entry in the array. When the scanning is finished, all horizontal lists are ready. After this the algorithm unlabels the elements of these horizontal lists. The vertical lists are built in the same way.

The algorithm proceeds recursively in the same way with each rectangle R_i, if R_i is non-empty. Thus, in linear time the algorithm splits the problem into subproblems of linear total size, such that all the subproblems have size smaller than half the size of the problem. This step is performed until all subproblems have size 0, i.e., $\log k'$ times. From this it follows that the overall time performance of the algorithm is $O(k' \log k')$ plus the preprocessing time, hence, the algorithm runs in time $O(n+k \log k)$. □

6.2.2 Proving a logarithmic bound

Consider a minimum-length tour, L, with start point SP, and let B_L be its minimal isothetic bounding box. Let $\mathcal{B}$ be the bounding square of X computed by the algorithm, as described above, and let L' be a shortest tour in $\mathcal{B}$. Since the length of the segments of a guillotine rectangular subdivision can be $\Omega(\sqrt{n})$ times the length of the perimeter of the bounding box, we first have to prove that the length of L' is $O(|L|)$. To obtain the desired result, we show that the subdivision is within a logarithmic factor longer than a shortest tour in $\mathcal{B}$.

Proposition 6.3 *If $\mathcal{B}$ is a bounding square with perimeter of length within a constant factor longer than the perimeter of B_L, and $\mathcal{B}$ includes a point in an optimal tour, then there exists a TSPN-tour of length $O(|L|)$ within $\mathcal{B}$.*

Proof: Since we have a given start point, we know that a minimum-length tour L at least partly lies within $\mathcal{B}$. Denote by L_1, respectively L_2, the part of L within, respectively outside, $\mathcal{B}$. Note that every neighborhood touched by L_2 also touches the perimeter of $\mathcal{B}$. This means that a tour that follows the perimeter of $\mathcal{B}$ and L_1 visits all the neighborhoods of X. It is easy to see that this tour is of length $O(|L|)$. □

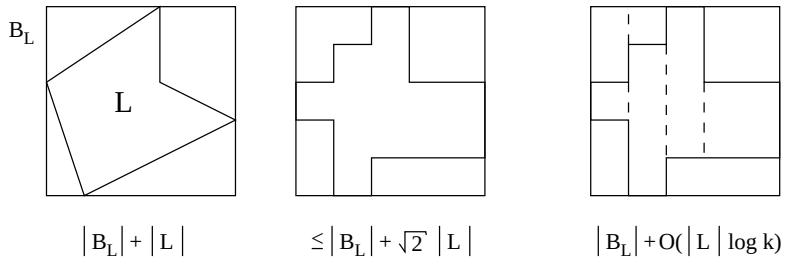

Figure 6.3: The length of the GRS is within a logarithmic factor longer than an optimal tour.

Recall that the perimeter of the bounding square produced by the algorithm is within a constant factor (4) longer than the perimeter of a minimal bounding box. Note the following:

Fact 6.4 *(Lemma 8 in [70]) A minimum-length tour is a simple polygon, having at most k vertices.*

This result also holds for L'. A shortest rectilinear TSPN-tour L_R, including the start point SP, is a simple rectilinear polygon and has length at most $\sqrt{2}|L|$. Furthermore, in [34, 59, 70] it was shown that there exists a guillotine rectangular subdivision of the rectilinear tour L_R and its minimal bounding box B_{L_R} such that the length of the subdivision is $O(|L_R| \log k)$, Fig. 6.3. Hence, we obtain the following fact:

Fact 6.5 *If $OPT(X, B_L)$ is a rectangular subdivision of minimum length of X and B_L, then it holds that $|OPT(X, B_L)|$=$O(|L| \log k)$.*

According to the above results we have that Fact 6.5 also holds if we replace B_L and L with $\mathcal{B}$ and L'. This means that an optimal rectangular subdivision of the polygonal neighborhoods X within the bounding square $\mathcal{B}$ is within a logarithmic factor longer than an optimal TSPN tour. Hence, it remains to prove that the GRS presented in the previous section produces a subdivision which is within a constant factor longer than an optimal subdivision.

The remainder of this section is devoted to the following lemma:

Lemma 6.6 *Let L' be a minimum-length tour within a bounding square $\mathcal{B}$, let L be an optimal tour and let B_L be its minimal isothetic bounding box. If the length of the perimeter of $\mathcal{B}$ is within a constant factor longer than the perimeter of B_L, then the GRS of X within $\mathcal{B}$ is of length $O(|L'| \log k)$.*

By using the result from Fact 6.5, it suffices to prove that the length of the GRS is within a constant factor longer than $OPT(X, \mathcal{B})$. This is done by proving

that for any possible rectangle T' within $\mathcal{B}$ and without polygonal neighborhoods entirely within its interior, the length of the GRS intersecting T' will be within a constant factor longer than the length of the perimeter of T'. Since a rectangle T' can be divided into squares such that the total length of the perimeters of the squares are not longer than twice the length of the perimeter of T', it suffices to prove the following lemma.

Lemma 6.7 *Let T be a rectilinear square within $\mathcal{B}$, without polygonal neighborhoods entirely inside its interior. The length of the GRS inside T is of length $O(p(T))$, where $p(T)$ is the length of the perimeter of T.*

We will prove the lemma in three steps, Proposition 6.8, 6.9 and 6.12.

Proposition 6.8 *Let T be a rectilinear square within $\mathcal{B}$ with sides of length t and without polygonal neighborhoods entirely within its interior. It holds that at most four parallel segments of the GRS touch two opposite sides of T.*

Proof: Let $e_1, \ldots, e_q$ be the parallel segments of the GRS inside T that touch two opposite sides of T in the order in which they are drawn by the procedure producing the subdivision. We may assume w.l.o.g. that these segments are vertical. Let S_i be the subset of the GRS just before e_i is produced. Let e_r and e_l denote the vertical segment to the right respectively to the left of e_1, such that e_1 partitions the rectangle in S_1 bounded from left respectively right by e_l and e_r. Note that e_l and e_r might be the left respectively right side of $\mathcal{B}$. We now examine what happens when e_2 is about to be produced. We have that e_2 will be placed between e_1 and the segment e_l or e_r. Assume w.l.o.g. that e_2 is placed between e_1 and e_r, as shown in Fig. 6.4a. The horizontal distance between e_1 and e_r is at least t, since e_2 is a vertical segment. Wherever e_2 is placed we know that the horizontal distance between e_1 and e_2 is less than t, since e_2 intersects T. Hence, the region bounded from left by e_1 and from right by e_2 will be partitioned by a horizontal segment e_h which means that there can be no vertical segments between e_1 and e_2 that touch both T's top and bottom side. But there can be a vertical segment e_3 that touches T's top and bottom side between e_2 and e_r, Fig. 6.4b. Since e_3 will lie to the right of the midpoint between e_1 and e_r we know that the distance between e_1 and e_3 is at least $t/2$. Can there be a segment e_4 between e_3 and e_r that touches T's top and bottom side? Assume that there exists such a segment e_4; then we know that the horizontal distance between e_3 and e_r is at least t; hence the horizontal distance between e_1 and e_r is at least $2t$, since e_3 lies closer to e_r than to e_1, and we have a contradiction since e_4 intersects T. Thus, there can be at most two vertical segments to the right of e_1 that touch two opposite sides of T. The same arguments as above hold for the region of T to the left of e_1. But note that

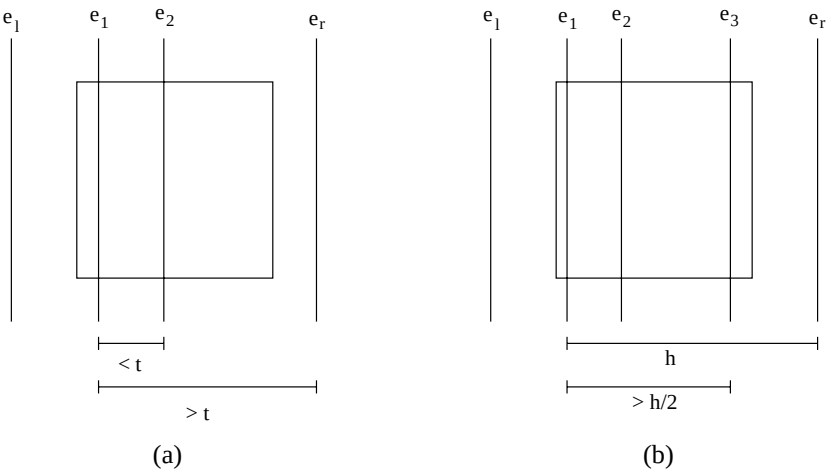

Figure 6.4: There can be at most four vertical segments that intersect opposite sides of T.

the horizontal distance between e_1 and e_3 is at least $t/2$ which means that there can only be one vertical segment to the left of e_1 that touches both T's top and bottom side. Hence there can be at most two vertical segments that touch two opposite sides of T's perimeter on one side of e_1 and at most one on the other side of e_1. □

Our next step is to calculate the length of the GRS inside T between two adjacent segments that touch two opposite sides of T.

Proposition 6.9 *Let T be a rectilinear square with sides of length t within $\mathcal{B}$, without polygonal neighborhoods entirely inside its interior, and let e_l and e_r be two adjacent parallel segments that touch two opposite sides of T. It holds that the length of GRS inside T between e_l and e_r is at most $7 \cdot s$, where s is the shortest distance between e_l and e_r.*

Proof: We may assume w.l.o.g. that e_l and e_r are vertical segments and that e_l lies to the left of e_r. Let T_j be the rectangle in the GRS with e_r as its right side and e_l as its left side. Since T is a square the part of T_j within T, denoted T'_j, will be partitioned into two regions, $T_j^{1'}$ and $T_j^{2'}$, by a horizontal segment, e_h, between e_l and e_r. Assume that $T_j^{1'}$ is above $T_j^{2'}$, Fig. 6.5a. It suffices to prove that the length of the GRS inside $T_j^{1'}$ is at most $3s$. Two cases may occur: $T_j^{1'}$ will be partitioned into two regions either by a horizontal segment or by a vertical segment with one endpoint on e_h and one endpoint outside T.

In the first case, where $T_j^{1'}$ is partitioned into two regions by a horizontal segment, we know that there cannot be any polygonal neighborhoods entirely

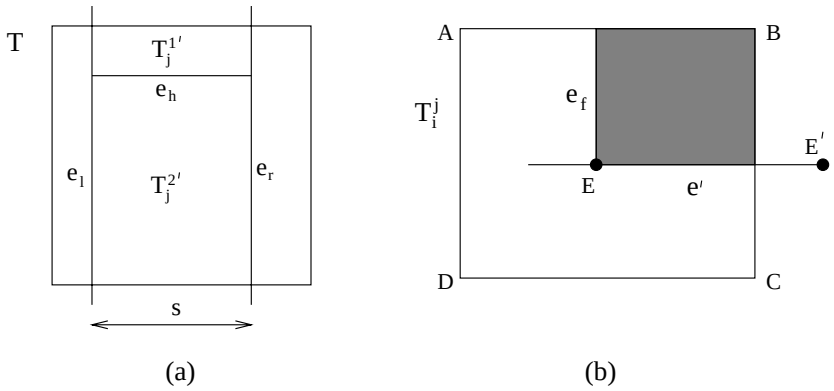

Figure 6.5: (a) The length of the edges within $T_j^{1'}$ is at most $7s$. (b) The vertical edges within the shaded region are not longer than e_f.

within $T_j^{1'}$. Hence, the horizontal segment lies on the perimeter of T or outside T. No other segments of GRS will intersect $T_j^{1'}$.

Otherwise, $T_j^{1'}$ is partitioned by a vertical segment with one endpoint on e_h and one endpoint outside T. Let e_1 denote the vertical segment. As we can see, this process may continue recursively until a horizontal segment is produced. It remains to prove that the total length of these vertical segments is at most $2s$, since the total length of the horizontal segments is s. For simplicity we assume that the vertical segments split the rectangular regions into equally sized rectangular subregions (worst case scenario). Then $T_j^{1'}$ is partitioned into, say, 2^p subrectangles and the length of a horizontal segment placed on or outside T's perimeter is $\frac{s}{2^p}$. By this we have that the length of the vertical segments is shorter than $\frac{s}{2^{p-1}}$ hence, the total length of the vertical segments is shorter than $2^p \times \frac{s}{2^{p-1}} = 2s$. The length of GRS in T_j' is less than $s+2s+2\times 2s = 7s$. □

We may assume w.l.o.g. that the segments that touch two opposite sides of T are vertical and ordered from left to right, $e_1, \ldots, e_q$ where $1 \leq q \leq 4$. We know that the length of the GRS inside T between and including e_1 and e_q is at most $4t+7|e_1e_q|$ where $|e_1e_q|$ is the horizontal distance between e_1 and e_q. It now remains to calculate the length of GRS within T, to the left of e_1 and to the right of e_q.

Fact 6.10 *Let T be a rectilinear square with sides of length t within $\mathcal{B}$, without any polygonal neighborhoods entirely within its interior. If any edge of GRS overlap T then there is at least one segment (and at most four) of GRS that touches two opposite sides of T.*

Let T_1 denote the rectangular region bounded from the left by e_q and from above, below and right by the perimeter of T. It is easy to see that the the next segment produced by the procedure will produce a horizontal segment partitioning T_1 into two rectangular regions, T_1^1 and T_1^2. Symmetrically the same holds for the rectangular region T_2 bounded from the right by e_1 and from above, below and left by T's perimeter. Hence, T_2 is partitioned into T_2^1 and T_2^2.

Let S_i^j , $1 \leq i, j \leq 2$, be the set of vertical segments inside T_i^j. By direct analogy, Observation 6.11 and Proposition 6.12 also hold for horizontal segments of T_i^j. Note that all the segments in S_i^j have one endpoint on the perimeter of T.

Observation 6.11 *(Modified version of Observation 1 in [59])*
For every segment e_f in S_i^j, it holds that either to the right or to the left of e_f there is no segment in S_i^j which is longer than e_f.

Proof: Assume that T_i^j is the upper right rectangular region inside T. Let A, B, C, D be the corners of T_i^j in clockwise order, such that $\overline{AB}$ is its top side. If e_f touches both $\overline{AB}$ and $\overline{CD}$ then the observation holds, because no segment in S_i^j is longer than e_f. Otherwise, the lower endpoint of e_f lies within the open region bounded by T_i^j, see Fig. 6.5b. Let E be that endpoint. We know that E lies on a horizontal segment of GRS, say e'. But at least one endpoint of e', say E', does not lie in the open region bounded by T_i^j, otherwise e' would lie on the perimeter of T. Moreover, no segments in GRS cross each other, and therefore on the side of e_f on which E' lies no endpoints of segments of S_i^j are below $\overline{EE'}$. □

Let S be any of the sets S_i^j, $1 \leq i, j \leq 2$, and let $e_1, e_2, \ldots$ be the segments in S ordered from left to right. By using this observation we know that there are three consecutive segments of S, which are longer than any other segments in S. Let us call three such segments e_{m-1}, e_m, e_{m+1}, for some integer m, $2 \leq m \leq k-1$, the *dominating* segments of S. The other segments of S are called *non-dominating*. Partition S into three disjoint subsets, S_1, S_2 and S_3, such that S_i, $1 \leq i \leq 3$, consists of the segments $e_j \in S$, for which it holds that $(j \mod 3) = i$.

Proposition 6.12 *(Modified version of Proposition 2 in [59])*
It holds that the total length of all the segments in S_i, $1 \leq i \leq 3$, is at most half the length of the perimeter of T_i^j.

The proof of Proposition 2 in [59] can be used as it is to prove our modified version. Also, similar arguments as in the proof of Proposition 6.8 can be used to show the proposition. Putting together the results from Proposition 6.8, 6.9 and 6.12, completes the proof of Lemma 6.7, Lemma 6.6 and Theorem 6.1.

6.3 Finding a good start point

In the previous section we showed a simple and fast algorithm that produces a TSPN tour when a start point is given. To be able to use this algorithm in the general case we will in this section show how to compute a nearly optimal start point, i.e., a start point that is included in a TSPN-tour of length $O(|L|)$, where L is a minimum-length tour of X. We describe a method which, given an arbitrary straight-line segment, computes in time $O(n \log n)$ a starting point on this segment which is a nearly optimal point under the condition that the optimal tour has to pass through this segment. This is of some interest in its own right. It yields an $O(\log k)$-approximation algorithm running in time $O(n \log n)$ whenever we know that some near-optimal tour crosses some fixed line. Next, by observing that there is always an optimal tour with non-empty intersection with the boundary of at least one neighborhood, we obtain as a corollary an $O(n^2 \log n)$-time method for finding a globally nearly-optimal start point. Hence, combining this result with the algorithm described in the previous section we obtain a logarithmic approximation algorithm that runs in time $O(n^2 \log n)$.

6.3.1 The length function

The idea is that we will for each of the n edges of the neighborhoods find the minimum L_∞-distance to all polygonal neighborhoods. This is done by constructing a length function $L(g, f)$ for each pair of edges $g, f \in E$ (not belonging to the same neighborhood), where E is the set of edges in X. This function describes the shortest distance in L_∞-metric from each point on g to f.

Let $g, f \in E$ be two edges in E. Rotate g and f such that g is horizontal. The endpoints of g respectively f are denoted G_1 and G_2 respectively F_1 and F_2. The length function is a piecewise linear function containing at most three different pieces, Fig. 6.6. Hence we will calculate the value of $L(g, f)$ in at most four points. Let $d(p, l)$ denote the shortest distance between an edge l and a point $p = (p.x, p.y)$. Compute the following four points: $p_1 = (G_1.x, d(G_1, f))$, $p_2 = (G_2.x, d(G_2, f))$, $p_3 = (F_1.x, d(F_1, g))$ and, finally, $p_4 = (F_2.x, d(F_2, g))$. The length function $L(g, f)$ is the function that we obtain by drawing straight line segments between these four points, from left to right. Note that some of these points may coincide. This function may now be used to compute the shortest distance between an edge $f \in E$ and a neighborhood $x \in X$, where $f \notin x$. First compute the length function for each edge in x and the edge f. We obtain $|x|$ length functions. Since we only are interested in the shortest distance between every point on f and the neighborhood x, we calculate the lower envelope of these $|x|$ functions, denoted $LEnv(f, x)$. Calculating the lower envelope of the $|x|$ functions can be done in time $O(|x| \log |x|)$ according to Sharir [89].

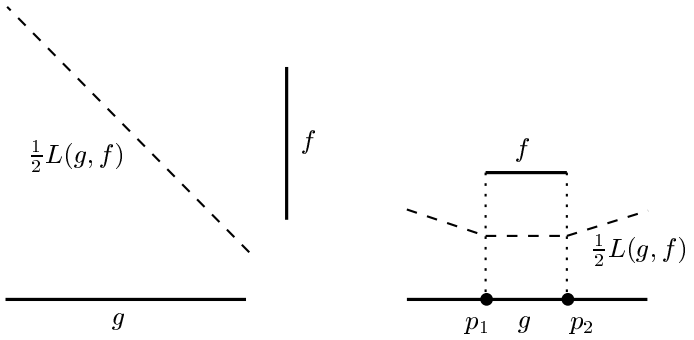

Figure 6.6: Two examples of the length function.

If f intersects the closed region described by x then we have to adjust the lower envelope. This can easily be done in linear time.

This new piecewise linear function describes the shortest distance between every point on f and the polygon x, and can be computed in time $O(|x| \log |x|)$.

6.3.2 Using the length function to obtain a good start point

The next step is to extend the above computation such that we obtain a function that for each point on f describes the shortest distance to all the neighborhoods in X. The algorithm is just an extension of the above described algorithm. Let f be an edge as above and let $\{x_1, \ldots, x_k\}$ be the neighborhoods in X. For each neighborhood $x_i \in X$ and $f \notin x_i$ compute the lower envelope, $LEnv(f, x_i)$, as described above. Combine the $k-1$ lower envelopes for all the neighborhoods in X to an upper envelope, $UEnv(f, X)$, for f. This new function describes the shortest distance between each point in f and all the neighborhoods in X. In the case when we know that an optimal TSPN-tour intersects f we can select the point on f with minimum $UEnv(f, X)$-value as start point (SP). Note that the function also gives us the size of an optimal bounding square with center at SP. For clarity we include a description of the algorithm (using a Pascal-like language).

The first **for**-loop is executed k times, the second loop will be executed n times (k times the number of edges in x_i). Calculating the upper/lower envelope can be done in time $O(n \log n)$. The idea is to partition the collection of functions into two subcollections. Then calculate recursively the envelopes of each subcollection separately. Finally merge the two partitions into a single refined partition. Hence the total time complexity for this algorithm is $O(n \log n)$.

for each neighborhood $x_i \in X$ **do**	k times
for each edge $e_j \in x_i$ **do**	$\lvert x_i \rvert$ times
calculate $L(f, e_j)$	$O(1)$
compute $LEnv(f, x_i)$	$O(\lvert x_i \rvert \log \lvert x_i \rvert)$
adjust $LEnv(f, x_i)$	$O(\lvert x_i \rvert)$
compute $UEnv(f, X)$	$O(n \log n)$
return the point on f with smallest $UEnv(f, X)$-value	

If the *UEnv* -function already is computed for every edge in E we can in linear time find a good start point for the general case, by just selecting the best of the n suggested start points. Hence, we obtain the following results.

Proposition 6.13 *Let L be an optimal TSPN tour. The computed bounding square $\mathcal{B}$ centered at SP includes a tour of length $O(|L|)$ and the length of the perimeter of the square is at most $4 \cdot |L|$.*

Proof: It is straight-forward to see that an optimal tour has at least length equal to one side of $\mathcal{B}$, since $\mathcal{B}$ is a smallest bounding square.

If an optimal tour at least partly lies within $\mathcal{B}$ then the proposition follows from Proposition 6.3. Otherwise, if $\mathcal{B}$ and an optimal tour are completely disjoint, it holds that every neighborhood intersects the perimeter of $\mathcal{B}$, hence the proposition follows. □

From the above proposition we have that the computed bounding square fulfills the requirement needed for Lemma 6.6 to hold and we obtain the following corollary.

Corollary 6.14 *Let X be a collection of k possibly overlapping simple polygons, having a total of n vertices. One can compute in time $O(n^2 \log n)$ a start point that is included in a tour of length within a constant factor times the optimal.*

6.4 TSPN when no start point is given

We have already described an approximation algorithm for the general case. By finding a start point as described in Section 6.3 and then applying the GRS in Section 6.2 we obtain an approximation algorithm that runs in time $O(n^2 \log n)$ and produces a tour that is within a logarithmic factor longer than an optimal tour. In this section we describe an algorithm that, depending on the given input, decides which method to apply to obtain a TSPN tour. The decision depends on the value of ϵ, an optional parameter given as input, and the ratio between the size of the smallest neighborhood in X and the length of a minimum-length tour. Recall that the minimal bounding box, B_L, is the smallest axis-aligned rectangle

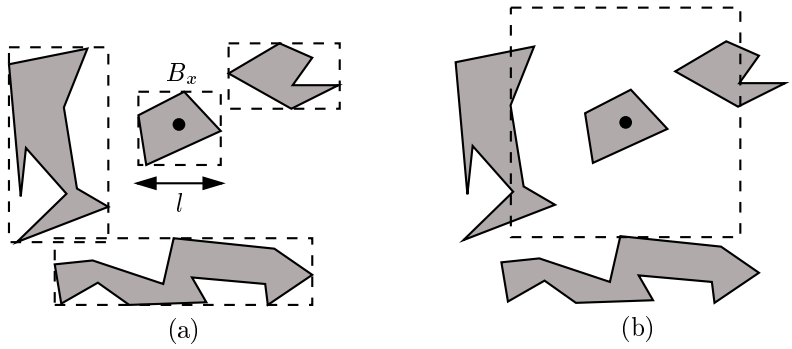

Figure 6.7: (a) x is the neighborhood in X such that its bounding box has shortest long side. (b) A minimal isothetic bounding square with center at the center of B_x.

that contains L, where L is a minimum-length tour of X. The algorithm works as follows.

First a minimal axis-aligned bounding box for every neighborhood in X is constructed. Let x be the neighborhood in X, such that its bounding box, B_x, has shortest long side. Let c denote the center of B_x, and let l be the length of B_x's longest side, see Fig. 6.7a. Next, a minimal isothetic bounding square $\mathcal{B}$ with center at c is constructed such that each neighborhood is at least partially contained in or touched by $\mathcal{B}$, Fig. 6.7b. This is done in linear time as described in Section 6.2. Two main cases may occur.

1. If $|\mathcal{B}|>4l$ then there exists a neighborhood $x'\in X$ such that x' lies outside B_x. Hence, an optimal tour must intersect B_x, that means that we can find a good start point, SP, that lies on one of the four sides of B_x. This can be done in time $O(n \log n)$ as explained in Section 6.3.2. By applying the subdivision algorithm defined in Section 6.2.1 with SP as start point we obtain a tour of length $O(\log k)$ times the optimal in time $O(n \log n)$.

2. If $|\mathcal{B}|<4l$ we first make a refined search in time $O(n)$ for much smaller bounding boxes in the vicinity of x. For this reason, we imagine a square-grid centered at c, with side length $2l$, and where each square has side length equal to $l/10$. For each one of the $O(1)$ corners of the small squares, we find the minimal isothetic bounding square, as if this corner was a given start point. Let $\mathcal{B}'$ be the smallest of these bounding squares and, let l' be the side length of $\mathcal{B}'$. Again, two cases may occur:

(a) If $l'>l/3$, then we output this bounding square as the TSPN tour, since no neighborhood can lie entirely within $\mathcal{B}'$ without touching the perimeter of $\mathcal{B}'$. We can show that it is within a constant factor from the minimum,

and we have found it in linear total time.

(b) If $l' \leq l/3$ then we run the approximation scheme described in Section 6.4.1. We obtain a TSPN tour of length $1+\epsilon$ in cubic time.

6.4.1 An approximation scheme

In this section we show how to obtain a polynomial-time approximation scheme, in the case when $l' \leq l/3$ (defined in the previous paragraph).

Observation 6.15 *For every TSPN tour of length less than $2l$, it holds that the convex hull of that tour also is a TSPN tour.*

The observation follows from the fact that no neighborhood of X can lie entirely within the closed region of a minimal bounding box of the TSPN tour, since the largest distance between two points on the tour is l.

Now, we search for $O(n^2)$ approximately minimal bounding squares (i.e. with perimeter at most a constant factor larger than the minimal tour), possibly overlapping, such that an optimal tour has to lie entirely inside at least one of them. This is done as follows. Find all *essential points* in X. These points are the vertices of the polygons of X and the intersections between the polygons. Since there are n edges there are $O(n^2)$ such intersections. Hence the number of essential points is $O(n^2)$. Then the algorithm selects every minimal bounding square with center at an essential point and with perimeter at most a constant factor longer than a minimal tour (a constant factor longer than the smallest found bounding square). Finally, for reasons explained in the proof of Observation 6.16, expand each selected minimal bounding square such that the side of a square is a constant, say 6, times the length of the original bounding square. There are $O(n^2)$ such bounding squares and each square takes $O(n)$ time to construct. Thus, the time complexity of this step is $O(n^3)$.

Observation 6.16 *An optimal tour is included in at least one of the $O(n^2)$ bounding squares produced above.*

Proof: First we show that there exists an essential point p such that the distance between p and every point in an optimal tour L is shorter than $\frac{\sqrt{3}+1}{2}|L|$. There are three cases.

1. If an optimal tour is a single point, then at least one of the essential points is also an optimal tour. Consider the closed region defined by the intersection of all neighborhoods in X. It follows that at least one vertex or intersection is included in this region.

2. If an optimal tour coincides with a straight line segment traversed back and forth, then there exists an optimal tour with one endpoint in an essential

point. Consider an optimal tour L described by a straight line segment. The endpoints of L coincides with either an edge of X, a vertex of X or an intersection between two edges of X. In the two latter cases it's obvious that L includes an essential point. Otherwise, if the endpoints of L lie on edges of X then it is possible to slide L until one of its endpoints coincides with a vertex of X or an intersection between two edges in X, such that the length of L does not increase.

3. Otherwise, if an optimal tour is a polygon with at least three vertices, each vertex of an optimal tour must lie on an edge of the neighborhoods or at an essential point. If none of the vertices of an optimal tour L touch an essential point then each vertex of L touches an edge of X. Since L is a convex shortest tour we know that it has perfect reflection at each vertex of L and that there exist two consecutive vertices v_1 and v_2 of L such that the sum of their interior angles is at least 120 degrees, see Fig. 6.8. Let e_1 respectively e_2 be the edges of X that touch v_1 respectively v_2. By straight-forward geometry we obtain that there exists an essential point p such that the shortest distance between p and the segment (v_1, v_2) is at most of length $\frac{\sqrt{3}}{2}|(v_1, v_2)|$. Note that p is either the intersection between e_1 and e_2 (worst case), or one of e_1's or e_2's endpoints. The distance between p and each point in L is at most $\frac{\sqrt{3}}{2}|(v_1, v_2)|+\frac{|L|}{2}$. By maximizing this formula we obtain that the largest distance between p and a point in L is less than $\frac{\sqrt{3}+2}{4}|L|$, since $|(v_1, v_2)| < \frac{|L|}{2}$.

From the above discussion we have that there exists an essential point p, such that the distance between p and every point in L is at most $\frac{\sqrt{3}+2}{4}|L|$. It remains to calculate the size of the smallest possible bounding square with center at p that does not include an optimal tour. Let B be a minimal bounding square with center at p, including a tour L' and with perimeter of length at most a constant factor longer than a minimal tour. The length of the sides of B is greater than $\frac{|L|}{2\pi}$, since the worst case occurs when L' has the shape of a circle with center at p and length $|L|+\epsilon'$. The minimal bounding square is expanded by the algorithm with a factor 6 ($>\frac{2\pi(\sqrt{3}+2)}{4}$), hence, an optimal tour is included in at least one of the produced bounding squares. □

For each selected bounding square b, let $l(b)$ be the length of the minimum tour overlapping with b and denote by $h(b)$ the length of the sides of b. For each b, we find a tour of length less than $(1+\epsilon)\cdot l(b)$ in linear time, and in this way in total time $O(n^3)$ we find a tour of length $(1+\epsilon)$ times the optimal.

A simple, although impractical, linear-time method to find a tour of length $(1+\epsilon)\cdot l(b)$, for a bounding box b, is as follows: We partition b into $f(\epsilon)\times f(\epsilon)$ equal-sized squares, where $f(\epsilon)$ is a sufficiently large constant only depending on ϵ. Let S be the set of all points which are corners of these small squares. Now, for each subset S' of S, we construct the convex hull of S'. Next, among those of the constructed convex hulls which are TSPN tours, we select the shortest one.

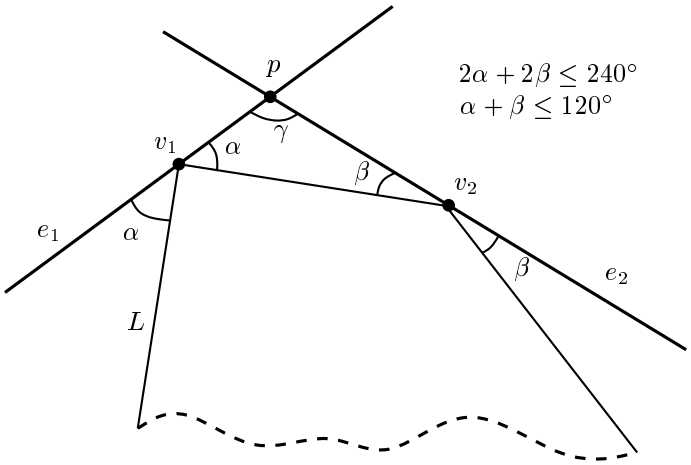

Figure 6.8: There exist two consecutive vertices of the tour such that the sum of their interior angles is at least 120 degrees.

(Of course, in practice we do not really have to consider all subsets of S.) It is easily seen that by choosing $f(\epsilon)$ sufficiently large, say, $f(\epsilon)=\frac{4h}{\epsilon}$, we will find a tour of length $(1+\epsilon)\cdot l(b)$.

6.5 Other applications

The methods described in this chapter apply also to other problems. In this section we show how to use our method for the Red-Blue Separation Problem (RBSP).

The objective is to find a minimum-length simple polygon that separates a set of red points, R, from a set of blue points B. This problem was shown to be NP-hard in [8] and [35]. In this section we give the idea of an $O(\log m)$ approximation algorithm for RBSP, where $m<n$ is the minimum number of sides of a minimum-perimeter rectilinear polygonal separator for the points. The total number of points is $n=|R|+|B|$. We will solve two problems and pick the one with shortest length: find a minimum-length polygon that encloses the red points, while excluding the blue points; and find a minimum-length polygon that encloses the blue points, while excluding the red points.

First construct a minimal bounding box for the red points respectively the blue points. This step is done in linear time since we just have to go through all the points and find the extreme points of each set.

Let P be a minimum-length separating simple polygon, of length $|P|$. With-

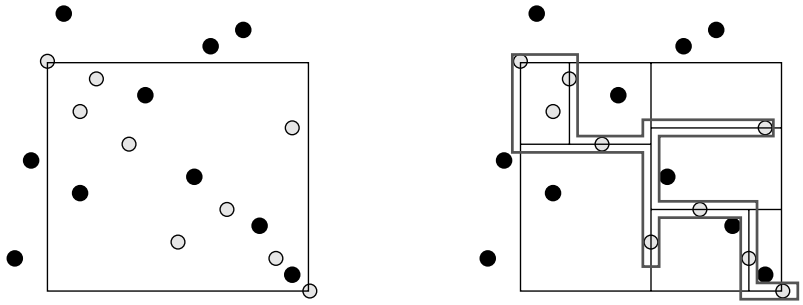

Figure 6.9: Computing a separating polygon for the red-blue separation problem.

out loss of generality, assume that P surrounds the blue points and excludes the red points. Let $\mathcal{B}$ be the bounding box for B. Note that P lies within $\mathcal{B}$ and that $\mathcal{B}$ is the bounding box of P. If no red points lie within $\mathcal{B}$ then we are finished. The length of the perimeter of this polygon is at most a factor $\sqrt{2}$ longer than $|P|$. Otherwise, one or more red points lie within $\mathcal{B}$. In this case construct a guillotine rectangular subdivision of $\mathcal{B}$, such that every rectangle within the subdivision only contains points of one color.

We need to adjust the GRS-algorithm presented in Section 6.2.1.

The partition is done as follows. Let T_j be the rectangle in GRS induced by all segments drawn by the procedure up to but not including e_j, such that e_j is drawn in T_j during the next step. Let A, B, C and D be the corners of T_j in clockwise order, such that $\overline{AB}$ is the top side of T_j. The segment e_j is drawn parallel to a shortest side of T_j, and it is vertical if T_j is a square. Assume for simplicity that $|AD| \leq |AB|$, i.e., e_j will be a vertical segment, as shown in Fig. 6.2. (The other case is identical if we imagine looking at the figure as if it was rotated 90°.) Let e be the vertical chord splitting T_j into two rectangles of equal size. Let N be the set of points entirely inside T_j. Two cases may occur:

1. If the region to the left of e is empty or if the region only contains points of one color (say, for example red) then the procedure draws e_j such that it intersects the leftmost point of the opposite color (in the example, blue). The corresponding procedure is performed symmetrically if the region in T_j to the right of e is empty or if it only contains points of one color.

2. If the above does not hold then the procedure draws e_j such that it passes through the point in N with shortest distance to e.

This is done recursively within the two resulting rectangles until all the polygonal neighborhoods are intersected by the subdivision.

By following the proof of Lemma 2 it follows that this algorithm runs in time $O(n \log n)$. And by doing some minor adjustments of the results in Section 3.2 one can show that the length of the GRS is $O(\log m)$ times the length of P. Mata and Mitchell [70] described how a guillotine subdivision and its bounding box can be traversed such that the resulting tour encloses all blue points, excludes all red points, and has length at most twice the length of the subdivision plus $|\mathcal{B}|$, as shown in Fig. 6.9. Hence, we obtain the following corollary.

Corollary 6.17 *Given a set of n points in the plane, where each point is either red or blue, a red-blue separating simple polygon of length $O(\log m)$ times the minimum can be computed in time $O(n \log n)$, where $m<n$ is the minimum number of sides of a minimum-perimeter rectilinear polygonal separator for the points.*

6.6 TSPN is APX-hard

In this section we show that TSPN is APX-hard, using an idea from Kann [50], and it cannot be approximated within a factor of 1.000374 unless P=NP. The reduction is done using the well-known Min Vertex Cover-problem (VC), which is known to be APX-hard and not approximable within a factor of $\frac{341}{340}$ in the case when the degree is bounded by 5 [16].

Definition 6.18 A *vertex cover* of an undirected graph $G=(V,E)$ is a subset $V' \subseteq V$ such that if $(u,v) \in E$ then u and/or v belongs to V'. The *min vertex cover problem* is to find a vertex cover of minimum size in a given graph.

Given a graph $G=(V,E)$ with $|V|=n$, construct an instance for the TSPN-problem that consists of $4|V|+|E|$ regions. We will define two kinds of regions: vertex regions and edge regions. The vertex regions are vertices in the plane and form a regular $|V|$-gon with circumference $|V|$. Hence, the distance between two "incident" vertices is 1. With incident we mean two vertices that are incident in the $|V|$-gon described by the convex hull of these vertices. Next, on the straight line between two incident vertex regions we place, evenly, another seven vertices. Hence the distance between two adjacent vertices in the $|V|$-gon is 1/8.

Now, let c_1 be the minimal circle, such that the vertex regions all lie on or within c_1, and let q be the center of c_1. The radius of c_1 is denoted $r(c_1)$. Next, we consider two circles c_2 and c_3 with center at q and radius $r(c_1)+1/2$ respectively $r(c_2)+K$, where K is some huge constant. For each edge (u,v) in E, we produce two "endpoints" of an edge region on the circle c_2, such that the distance between an endpoint and the incident vertex u or v is minimized, i.e., the distance between a vertex region and an edge region is 1/2, as shown in

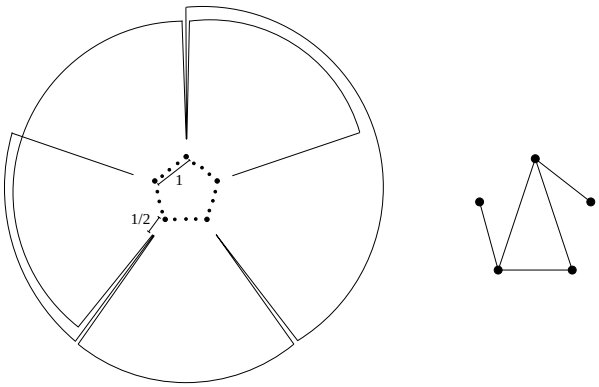

Figure 6.10: Transformation of an instance for vertex cover into an instance of TSPN.

Fig. 6.10. These two endpoints, p_1 and p_2, are then connected by three segments. One minimal segment from p_1 to the circle c_3, one minimal segment from p_2 to c_3 and finally the part of the circle c_3 between p_1 and p_2, Fig. 6.10.

If all the edges in the original graph can be covered with $k{\cdot}|V|$ vertices then there exists an optimal TSPN-tour of length $|V| + t{\cdot}k{\cdot}|V|$, where t is the length by which the tour increases when an edge region is visited. We have that

$$t > \frac{7}{8} \quad \text{for} \quad |\mathrm{V}| > 2.$$

That is, an optimal tour will follow the $|V|$-gon around until it reaches a vertex in the VC. Then the tour will leave the $|V|$-gon and visit the corresponding edge region, and then return to the $|V|$-gon. The length of this tour will be $|V| + t{\cdot}k{\cdot}|V|$, as stated above. It is not hard to show that this tour is optimal for any k.

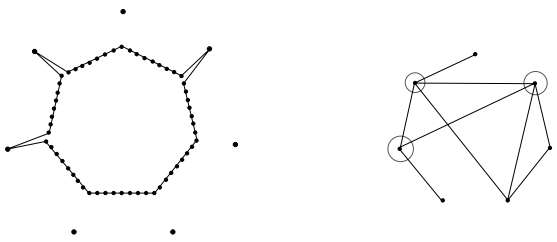

Figure 6.11: (a) The calculated tour. (b) The input graph and the suggested solution.

Assume that there is an approximative TSPN-tour of length $\leq |V|+t{\cdot}k{\cdot}C{\cdot}|V|$, then we get that the tour at most visits $C{\cdot}k{\cdot}|V|$ edge regions, hence VC is then approximated within a factor of C. This gives us a lower bound on the approximation ratio for TSPN. Note that $t > \frac{7}{8}$ and $k \geq \frac{1}{6}$ since the degree in the original graph is bounded by 5. The result is now obtained by calculating the maximal approximation ratio of an algorithm for TSPN such that the number of vertices in the resulting vertex cover is within a factor of $C{=}\frac{341}{340}$.

$$\frac{1+t{\cdot}k{\cdot}C}{1+t{\cdot}k} > \frac{1+\frac{7}{8}{\cdot}\frac{1}{6}{\cdot}C}{1+\frac{7}{8}{\cdot}\frac{1}{6}} > 1.000374.$$

We obtain the main theorem.

Theorem 6.19 *It is NP-hard to approximate TSP with neighborhoods with an approximation ratio smaller than 1.000374.*

6.7 Open problems

There are several open questions concerning these problems. It is natural to ask whether a constant approximation ratio is achievable for these problems. Further, is there a logarithmic approximation algorithm for TSPN in the case when the neighborhoods are not simple polygons, for example, sets of polygons or points in the plane?

Chapter 7

Approximating minimum Manhattan networks

A *rectilinear path* connecting two points in the plane is a path consisting of only horizontal and vertical line segments. A rectilinear path of minimum possible length connecting two points will be referred to as a *Manhattan path*, where the length of a rectilinear path is equal to the sum of the lengths of its horizontal and vertical line segments. Manhattan paths are monotonic. The *Manhattan distance* (or L_1-distance) between two points in the plane is the length of the Manhattan path connecting them. In this paper we introduce the concept of geometric networks that guarantee Manhattan paths between every pair of points from a given set of points.

Consider a set S of points in the plane. A *geometric network* on S can be modeled as an undirected graph $G = (V, E)$. The vertex set V corresponds to the points in $S \cup S'$, where S' is a set of newly added Steiner points; the edge set E corresponds to line segments joining points in $S \cup S'$. If all the line segments are either horizontal or vertical, then the network is called a *rectilinear geometric network*. Each edge $e = (a, b) \in E$ has length $|e|$ that is defined as the Euclidean distance $|ab|$ between its two endpoints a and b. The total length of a set of edges is simply the sum of the lengths of the edges in that set. The *total length* of a network $G(V, E)$ is denoted by $|E|$. For $p, q \in S$, a pq-path in G is a path in G between p and q.

For a given set S of n points in the plane, we define a *Manhattan Network* on S as a rectilinear geometric network G with the property that for every pair of points $p, q \in S$, the network G contains a Manhattan pq-path connecting them. A *Minimum Manhattan Network* on S is a Manhattan network of minimum length.

The complete grid on the point set S is clearly a Manhattan network. In

other words, the network obtained by drawing a horizontal line and a vertical line through every point in S and by considering only the portion of the grid inside the bounding box of S is a network that includes the Manhattan path between every pair of points in S. It is easy to show that the minimum Manhattan network on S need not be unique and that the complete grid on the point set S can have total weight $O(n)$ times that of a minimum Manhattan network on the same point set. Figures 7.1b and c below show examples of a Manhattan

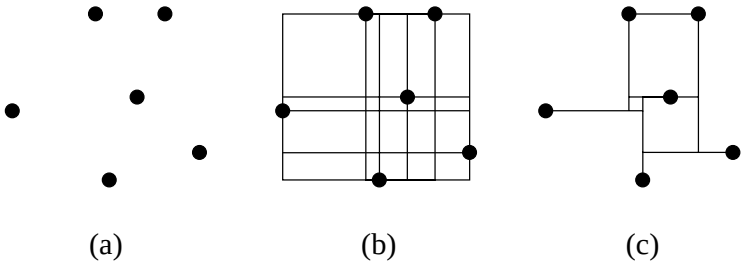

Figure 7.1: (a) A set of input points, (b) A Manhattan network, and (c) A minimum Manhattan network.

network and a minimum Manhattan network on the same set of points. In fact, the network in Figure 7.1b is also a complete grid on the input points.

Many VLSI circuit design applications require that a given set of terminals in the plane must be connected by networks of small total length. Rectilinear Steiner minimum trees were studied in this context. Manhattan networks impose additional constraints on the distance between the terminals in the network. The concept of Manhattan networks seems to be a very natural concept; it is surprising that this concept has not been previously studied. Manhattan networks are likely to have many applications in geometric network design and in the design of VLSI circuits.

Manhattan networks are also closely connected to the concept of spanners. Given a set S of n points in the plane, and a real number $t>1$, we say that a geometric network G is a *t-spanner* for S for the L_p-norm, if for each pair of points $p, q \in S$, there exists a pq-path in G of length at most t times the L_p-distance between p and q. In this connection, a minimum Manhattan network can be thought of as a sparsest 1-spanner for S for the L_1-norm, assuming that Steiner points are allowed to be added. 1-spanners are also interesting since they represent the network with the most stringent distance constraints. However, note that the sparsest 1-spanner for S in the L_p-norm (for $p \geq 2$) is the trivial complete graph on S. It is also interesting to note that a Manhattan network can

be thought of as a $\sqrt{2}$-spanner (with Steiner points) for the L_2 norm. Although complete graphs represent ideal communication networks, they are expensive to build; sparse spanners represent low cost alternatives. The weight of the spanner network is a measure of its sparseness; other sparseness measures include the number of edges, maximum degree and the number of Steiner points. Spanners have applications in network design, robotics, distributed algorithms, and many other areas, and have been a subject of considerable research [6, 11, 22, 31, 62]. More recently, spanners have found applications in the design of approximation algorithms for problems such as the traveling salesperson problem [10, 82].

In this paper we present an algorithm that produces a Manhattan network for a given set S of n points in the plane. The total weight of the network output by the algorithm is within a constant factor of the minimum Manhattan network. It is interesting to note that in this paper we reduce the problem of computing an approximate minimum Manhattan network to the problem of finding a minimum-weight rectangulation of a set of staircase polygons. If the rectangulation algorithm runs in time $O(R(n))$ and produces a rectangulation that is within a factor r of the optimal, our algorithm will produce a Manhattan network of total weight $4r$ times the weight of a minimum Manhattan network in time $O(n \log n + R(n))$. Using two known approximation algorithms for the minimum-weight rectangulation problem, we obtain two algorithms for the approximate minimum Manhattan network problem. The first algorithm runs in $O(n^3)$ time and produces a Manhattan network of total weight at most four times that of a minimum Manhattan network. The second algorithm runs in $O(n \log n)$ time and has an approximation factor of eight. It is unknown whether the problem of computing the minimum Manhattan network is a NP-hard problem. It is also unknown whether a polynomial-time approximation scheme exists for this problem.

A noteworthy feature of our result is that unlike most of the results on t-spanners, we compare the output of our algorithm to that of minimum Manhattan networks and our results involve small constants (4 or 8). Most results on sparse t-spanners prove weight bounds that compare it to the length of a minimum spanning tree; the constants involved in those results are usually very large.

In Section 7.2, we present the approximation algorithm. In Section 7.2.2 we prove that the algorithm produces a Manhattan network; in Section 7.2.3 we prove that the network produced is of weight at most $4r \times |E_{opt}|$, where E_{opt} is the set of edges in a minimum Manhattan network on S.

7.1 Definitions

Let u and v be two points in S, where u lies to the left of and above v. A $\llcorner$-path between u and v is a rectilinear path consisting of one vertical line segment with upper endpoint at u, and one horizontal line segment with right endpoint at v. Note that such a path is a minimum-weight, minimum-link path connecting u and v. The $\urcorner$-path is defined in a similar fashion, as shown in Fig. 7.2. If u lies to the left of and below v, we define a $\lrcorner$-path and a $\ulcorner$-path in a similar fashion. Note that each of the four paths described above introduces one Steiner point at the bend. We will denote by $[v, u]$ the closed region described by a rectilinear rectangle with corners in v and u. The coordinates of a vertex $v \in V$ are denoted $v.x$ respectively $v.y$.

Figure 7.2: (left) An $\llcorner$-path, and (right) an $\urcorner$-path.

If two different edges of the graph intersect, we will assume that their intersection defines a Steiner point.

7.2 The approximation algorithm

In this section we present an approximation algorithm to construct a Manhattan network $G'=(V', E')$ of small length. The algorithm will construct the edge set in four independent steps. In each step, for each vertex, the algorithm constructs a (possibly empty) local network connecting that vertex to a set of chosen "neighboring" vertices. Since the four steps are symmetrical the first step is explained in more detail than the later steps.

1. Sweep the points from left to right. As shown in Fig. 7.3a, for each point $v' \in V$ let v be the leftmost point below and not to the left of v' (if several, take uppermost). We say that v' *1-belongs to* v. For each vertex v let $B_1(v)$ denote the set of vertices in V that 1-belong to v. Let $v_1, \ldots, v_m$ be the vertices in $B_1(v)$ ordered from left to right, as shown in Fig. 7.3b. First, construct an $\urcorner$-path, denoted e_1, connecting v_1 and v. If $m>1$, draw a vertical edge e_2, with top endpoint at v_m and bottom endpoint on e_1. Next, a "local" Manhattan network is constructed such that there is a Manhattan path from each vertex v_i, $2 \leq i \leq v_{m-1}$, to e_1 or e_2; this step is

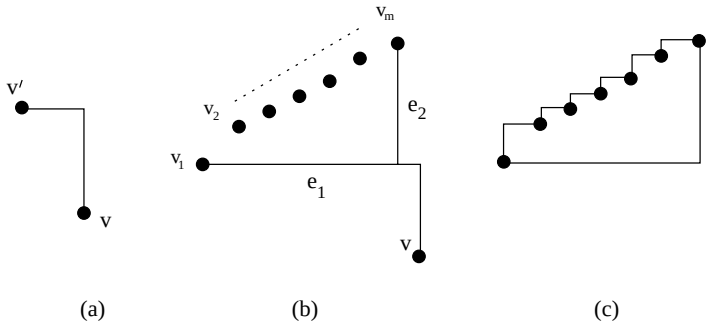

Figure 7.3: (a) Vertex v' 1-belongs to v. (b) The set $B_1(v) = \{v_1, \ldots, v_m\}$ 1-belongs to v. (c) The C-hull of $B_1(v)$.

explained in more detail in Section 7.2.1. Add the edges constructed in this step to the set of edges E'_1.

2. Sweep the points from left to right. For each point $v' \in V$ let v be the leftmost point above and not to the left of v' (if several, take bottommost). We say that v' *2-belongs to* v. For each vertex v let $B_2(v)$ denote the set of vertices in V that 2-belong to v. Perform the symmetrical procedure as performed in step 1 on the set $B_2(v)$ for every $v \in V$, to obtain the set of edges E'_2.

3. Sweep the points from bottom to top. For each point $v' \in V$ let v be the bottommost point to the left and not below v' (if several, take leftmost). We say that v' *3-belongs to* v. For each vertex v let $B_3(v)$ denote the set of vertices in V that 3-belong to v. Perform the symmetrical procedure as performed in step 1 on the set $B_3(v)$ for every $v \in V$, to obtain the set of edges E'_3.

4. Sweep the points from top to bottom. For each point $v' \in V$ let v be the topmost point to the left and not above v' (if several, take leftmost). We say that v' *4-belongs to* v. For each vertex v let $B_4(v)$ denote the set of vertices in V that 4-belong to v. Perform the symmetrical procedure as performed in step 1 on the set $B_4(v)$ for every $v \in V$, to obtain the set of edges E'_4.

After building the appropriate "local" networks, we say that every vertex v is *directly connected* to the vertices in $B_1(v) \cup \ldots \cup B_4(v)$. From these four sweeps we get four edge sets, $E'_1, \ldots, E'_4$. The Manhattan network is now defined as

$G = (V', E')$, where $E' = E'_1 \cup \ldots \cup E'_4$, and V' includes the points in S and all the Steiner points that are generated when adding the edges in E'.

Also note that there is some asymmetry in the above construction. This is deliberate; the asymmetric cases are required in the proof of correctness of the algorithm (see Lemma 7.3).

7.2.1 Constructing the local networks

We will explain the construction of the local networks involved in step 1 of the algorithm in detail. The constructions involved in the other steps are symmetrical. Let v be an arbitrary vertex of V and let $v_1, \ldots, v_m$ be the vertices in $B_1(v)$. By step 1 of the algorithm, we note that v_i lies below and to the left of v_{i+1}, $1 \leq i < m$; thus, $v.x \geq v_m.x$ and $v.y \leq v_1.y$.

In this section we describe how to construct a local network connecting v with the vertices $v_1, \ldots, v_m$; the local network is a Manhattan network on $v, v_1, \ldots, v_m$. We assume that $m > 2$, otherwise we are done. Recall that v_1 and v are connected by an $\urcorner$-path, e_1, and that v_m is connected to this edge by a vertical segment corresponding to edge e_2, see Fig. 7.3b. Let e'_1 be the horizontal part of e_1 between v_1 and e_2. Our aim is to produce a set of edges of minimum total weight such that there is a Manhattan path between v_i and v, $1 \leq i \leq m$. This already holds for v_1 and v_m. Consider the following staircase polygon obtained by adding $\ulcorner$-paths between v_i and v_{i+1}, $1 \leq i < m$, to the base e'_1 and the right side e_2, as shown in Fig. 7.3c. This polygon will be referred to as the C-hull of the set of vertices $\{v_1, \ldots, v_m\}$. We claim that a rectangulation of this polygon would give us a rectilinear network that guarantees Manhattan paths from v to every vertex in the set $\{v_1, \ldots, v_m\}$. This claim is easily proved by observing that every vertex $v_i, i = 1, \ldots, m$, must lie on the perimeter of a distinct rectangle (hence you should be able to proceed either down or to the right from v_i), and there always exists a monotonic rectilinear path from v_i to v that follows the borders of the rectangles encountered along the way. Denote by $E'_1(v)$ the set of edges constructed in the rectangulation of the C-hull plus the two edges e_1 and e_2. That is, $E'_1(v)$ is the set of edges produced in step 1 to connect the vertices in $B_1(v)$ to v. It should be noted that $E'_1(v)$ only consists of the edges in the interior of the C-hull of the set of vertices $\{v_1, \ldots, v_m\}$ and that the $\ulcorner$-paths from v_i to v_{i+1} are not included in it. While a rectangulation of these C-hulls guarantees Manhattan paths, we show later that a minimum-weight rectangulation of the C-hulls gives us an approximation algorithm for the problem. The following results on minimum-weight rectangulations were previously known:

Theorem 7.1 *([64]) An optimal rectangulation of a staircase polygon can be computed in time $O(n^3)$.*

Theorem 7.2 *A thickest-first rectangulation of a staircase polygon can be computed in linear time such that the weight of the added edges in the rectangulation is at most twice the weight of an optimal rectangulation.*

Proof: A thickest-first rectangulation "cuts" in every step off a maximal rectangle whose shortest side is as long as possible. Levcopoulos and Östlin [63] showed that a thickest-first partition of a histogram can be computed in linear time. Hence, it remains to prove that for a staircase polygon a thickest-first partition produces segments of length at most twice the length of the optimal.

Let P be a staircase polygon. We assume for simplicity that the base, b, of P is horizontal and at the bottom, and the longest vertical edge of P, denoted h, is the right side of P, Fig. 7.4a. Let R be a thickest rectangle within P and let l_R be the set of segments of R's perimeter which are disjoint from the boundary of P. The proof is by induction. If P is a rectangle then l_R is empty and we are finished. Otherwise l_R contains one or two segments. Assume that the observation holds for the staircase polygon of P above R's top side and the staircase polygon of P to the left of R's left side, if they exist. To be able to use induction let S be the open region defined by the interior of R plus the segment or segments in l_R. Let OPT be the segments in an optimal rectangulation of P. By the induction hypothesis it holds that the total length of the segments produced by the thickest-first rectangulation minus the segment or segments in l_R is at most of length $2\cdot(|OPT|-|OPT(S)|)$, where $OPT(S)$ is the set of segments in OPT intersecting S. Hence, it is enough to prove that $2|l_R| \leq |OPT(S)|$.

We will have two cases, either there are segments of $OPT(S)$ within the open region of R, or there are not.

1. No segments intersect the open region of R. In this case there must be segments of $OPT(S)$ that are equal to the segments in l_R, otherwise OPT would not be a rectangulation of P. Hence, the total length of the segments in l_R is equal to the length of $OPT(S)$.

2. Otherwise, if there are segments of $OPT(S)$ within the open region of R, we know that there must be segments in $OPT(S)$ of length at least equal to the shortest side of R. It is easy to see that every segment in l_r is shorter than the shortest side of R, otherwise R would not have been a thickest rectangle. It follows that the total length of the segments in l_r is less than two times the total length of the segments in $OPT(S)$. □

Sweeping the vertices of S takes $O(n \log n)$ time. If the optimal rectangulation procedure is used the time-complexity of our approximation algorithm

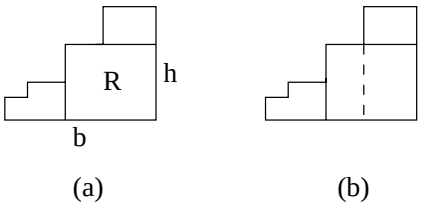

Figure 7.4: (a) A thickest rectangle R within the polygon. (b) A thickest-first rectangulation is within a factor two of the optimal.

is $O(n^3)$. And, if the thickest-first rectangulation algorithm is used then our algorithm runs in time $O(n \log n)$.

It remains to prove that the graph $G'=(V', E')$ is a Manhattan network for the points in S and that $|E'| \leq 4r \times |E_{opt}|$, where r is the approximation factor of the rectangulation algorithm.

7.2.2 The algorithm outputs a Manhattan network

To show that the algorithm outputs a Manhattan network, it suffices to prove the following lemma:

Lemma 7.3 *For each pair of points $s, t \in V$ there is a Manhattan path in G' connecting s and t.*

Proof: Without loss of generality, either t lies to the right and above s, or t lies to the right and below s (if not, switch s and t). We first assume that t lies to the right and above s. Without loss of generality, we may assume that no two points have the same x- or y-coordinate, since the algorithm always connects two such points by a line segment.

Consider $[s, t]$, the rectilinear rectangle with corners at s and t. If $s \in B_2(t)$ or $t \in B_4(s)$, then s is directly connected to t and we are done. Otherwise, we know from the algorithm that s is directly connected to a point $s_1 \in V$ above and to the right of s and to the left of t (Case 4 of construction), and that t is directly connected to a point $t_1 \in V$ below and to the left of t and above s (Case 2 of construction). We consider the following two cases:

Case 1 [s_1 or t_1 lies within $[s, t]$]: Without loss of generality, assume that s_1 lies within $[s, t]$, as shown in Fig. 7.5a. In this case, let s_1 be the new s and continue recursively.

Case 2 [s_1 and t_1 lie outside $[s, t]$]: Then we know that s_1 lies above and to the left of t and that t_1 lies above and to the left of s, see Fig. 7.5b. The Manhattan

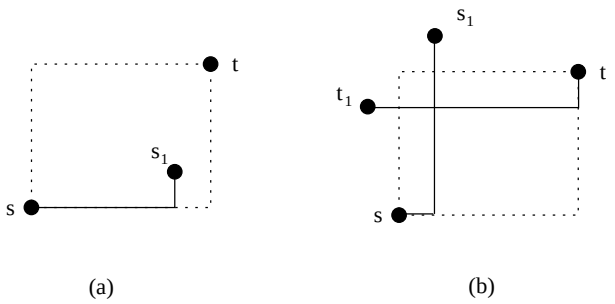

Figure 7.5: There exists a Manhattan path between s and t.

path connecting s and s_1 must intersect the Manhattan path connecting t and t_1 within $[s,t]$. Hence there is a Manhattan path connecting s and t.

The case when t lies to the right and below s uses Cases 1 and 3 of the construction, and is otherwise similar to the proof above. Hence the lemma.

□

7.2.3 Bounding the length of the network

For the length analysis, once again, it is sufficient to consider only one sweep, i.e., step 1 of the algorithm. The approximation factor for this sweep is then multiplied by four to obtain the approximation factor of the algorithm. Let v be an arbitrary vertex of V. Let $B_1(v) = \{v_1, \ldots, v_m\}$ and, let $V' = B_1(v) \cup \{v\}$. We define a *charging area* of v (with respect to this sweep) as the region $\cup_{i=1}^{m}[v, v_i]$, as shown by the shaded region in Fig. 7.6a. The charging area is denoted by $C_1(v)$. Note that the interior of the charging area for any vertex must be empty of input points. We start with the following observation:

Lemma 7.4 *For every pair of vertices $v_i, v_j \in V$, the charging areas $C_1(v_i)$ and $C_1(v_j)$ are disjoint, except possibly for the point v_i or v_j.*

Proof: Since no vertex can 1-belong to more than one vertex, the staircase parts of the two charging areas cannot share any vertices. Thus either v_i is on the staircase part of $C_1(v_j)$ or vice versa. But then, the rest of the charging areas cannot overlap because of its shape and orientation.

□

It is important to point out that the charging areas may share a point, but cannot share an edge of the boundary. Also note that all edges of $E_1'(v)$ that

were added in step 1 must lie entirely within $C_1(v)$. Hence, the edges produced in step 1 connecting vertices in V' cannot be used to connect vertices in any other charging areas. We will prove that $|E'_1(v)|$ is at most equal to the length of the edges in E_{opt} lying within the charging area of v. Since the charging areas are disjoint this implies that the total length of the edges produced in step 1 of the algorithm, $|E'_1| = \sum_{v \in V} |E'_1(v)|$, is at most $|E_{opt}|$.

First, partition the charging area $C_1(v)$ into three regions, where $R_1 = [v, v_1]$, $R_2 = [v, v_m] \backslash R_1$ and R_3 is the remaining region of the charging area. The three regions are shown in Fig. 7.6b. Before we continue, recall that $E'_1(v)$ consists of a $\urcorner$-path, e_1, connecting v_1 and v, a vertical edge e_2 connecting v_m with e_1, and a rectangulation of the C-hull of $B_1(v)$ (which is meant to connect v_i, $2 \leq i < m$, with e_1 or e_2).

Consider a minimum Manhattan network connecting the vertices in V. What do we know about E_{opt} of such a network?

1. E_{opt} must include a Manhattan path between v_1 and v within R_1. Note that this path has the same length as e_1.

2. E_{opt} must include a Manhattan path between v_m and v within $[v, v_m]$. Within R_2 this path has at least the same length as e_2.

3. If $m > 2$ then there must be a network N_{opt} connecting $v_2, \ldots, v_{m-1}$ with e_1 or e_2 within R_3. Hence, it remains to prove that a minimum weight network connecting $v_2, \ldots, v_{m-1}$ to the right or bottom side of R_3 has weight equal to a minimum weight rectangulation of R_3.

Lemma 7.5 *A minimum-weight network connecting v_i, $2 \leq i < m$, with e_1 or e_2 has length at least equal to the length of a minimum-weight rectangulation of the C-hull.*

Lemma 7.5 is a direct consequence of the following two lemmas. Consider a grid induced by the vertices of V'. Let N_{opt} be an optimal network within R_3 connecting $v_2, \ldots, v_{m-1}$ to the right or bottom side of R_3. Let N_{rect} be a minimum-weight network, connecting $v_2, \ldots, v_{m-1}$ to the right or bottom side of R_3, whose segments lie (only) on the grid induced by the vertices in V'.

Lemma 7.6 $|N_{rect}| = |N_{opt}|$.

Proof: Let P be a Manhattan path between vertices v_i and v. Assume that the path is moving right on the grid and that it changes direction downwards without reaching an intersection of the grid. Denote by t the last intersection of the grid that the path passed. First, it is obvious that any monotone path from v_i to v lying on the grid is of equal weight. Hence, the only reason to change

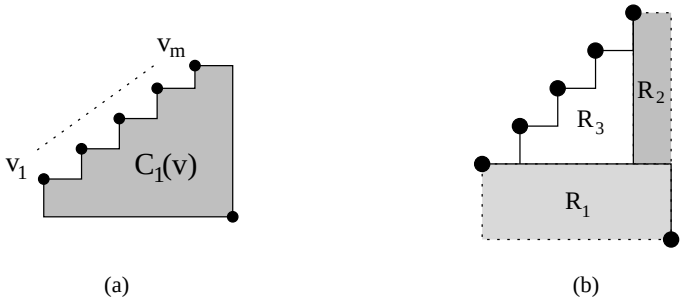

Figure 7.6: (a) The charging area. (b) Partitioning the charging area into three regions.

direction "outside" the grid is that other paths may use this segment. Since all paths that may use this segment start at vertices to the left of t, they must start from vertices v_j, where $j < i$, which cannot intersect till the next horizontal grid line below it. Thus the path can be "straightened" within that grid cell to follow the grid lines without decreasing its length. Going step by step, all paths can be modified to follow grid lines. Hence, the observation follows.

□

Lemma 7.7 *There exists a minimum-weight rectangulation of the interior of the C-hull of length $|N_{rect}|$.*

Proof: Every path between an interior point v_i and v moves (seen from v_i) only in two directions, right and down. The only case when a path would induce a non-rectangular network of the C-hull is when it turns right or down without meeting another path. Assume we follow a path from v_i going down and then turning right, without meeting a horizontal segment. In this case the path could have been shortened by not changing direction, see Fig. 7.7b, or by starting going right from the beginning. This is easy to see since the horizontal distance between a vertical segment and the turning point of the path is equal to the horizontal distance to the vertical segment and v_i. Hence, we do not gain anything by going downwards if the path is not meeting a horizontal segment. This means that there exists a rectangulation of the interior of the C-hull of weight $|N_{rect}|$.

□

Putting these results together we obtain the following lemma.

Lemma 7.8 $\sum_{v \in V} |E_1'(v)| \leq r \times |E_{opt}|$.

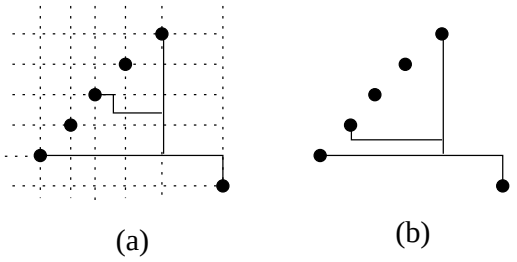

Figure 7.7: Following the grid induced by the set of points does not increase the length of the network.

To obtain the approximation factor for the algorithm just multiply the approximation factor for each step by four, since there are four sweeps. We summarize this paper by giving the main theorem.

Theorem 7.9 *Given a set of n points in the plane, and given an r-approximate, $R(n)$-time algorithm to compute a minimum-weight rectangulation of a staircase polygon, there exists an $O(n \log n + R(n))$-time algorithm that outputs a Manhattan network of length at most $4r$ times that of a minimum Manhattan network.*

7.3 Open Problems

The following problems remain open:

1. Determine the complexity of the problem of computing minimum Manhattan networks.
2. Design a PTAS for the problem of computing minimum Manhattan networks.
3. Design a 2-approximate algorithm for the problem of computing minimum Manhattan networks.

Chapter 8

Fast algorithms for constructing sparse geometric spanners

Complete graphs represent ideal communication networks but they are expensive to build; sparse spanners represent low cost alternatives. The weight of the spanner network is a measure of its sparseness; other sparseness measures include the number of edges, maximum degree and the number of Steiner points. Spanners for complete Euclidean graphs as well as for arbitrary weighted graphs find applications in robotics, network topology design, distributed systems, design of parallel machines, and many other areas, and have been a subject of considerable research [6, 11, 22, 31, 62].

Consider a set V of n points in $\mathbb{R}^d$, where the dimension d is a constant. A network on V can be modeled as an undirected graph G with vertex set V and with edges $e = (u, v)$ of weight $wt(e)$. A Euclidean network is a geometric network where the weight of the edge $e = (u, v)$ is equal to the Euclidean distance $d(u, v)$ between its two endpoints u and v. Let $t > 1$ be a real number. We say that G' is a *t-spanner* for V, if for each pair of points $u, v \in V$, there exists a path in G' of weight at most t times the Euclidean distance between u and v. A *sparse t-spanner* is defined to be a t-spanner of size $O(n)$ and weight $O(wt(MST))$, where $wt(MST)$ is the total weight of a minimal spanning tree. Given a geometric network $G = (V, E)$, a (generic) weight function w defined on its edges, and two vertices $u, v \in V$, we let $D_{\{G,wt\}}(u, v)$ denote the weight of the shortest path from u to v in G for the weight function wt.

The problem of constructing spanners has been investigated by many re-

searchers. Levcopoulos and Lingas [61] presented an $O(n \log n)$-time algorithm for the 2-dimensional case that produced a sparse t-spanner. But the problem gets much more difficult in higher dimensions. There are several algorithms that run in time $O(n \log n)$ [20, 52, 85, 93]. However, they only guarantee a linear number of spanner edges and they do not guarantee low weight. Das and Narasimhan [31] gave an $O(n \log^2 n)$-time algorithm that constructs for any set V of n points in $\mathbb{R}^d$, and any constant $t > 1$, a sparse t-spanner for V in which the degree of every point is bounded by a constant. Chen et al. [25] showed that the lower bound for computing any t-spanner for a given set of points V in R^d is $\Omega(n \log n)$ in the algebraic computation tree model.

Mount [74] has shown that a significant result claimed in Arya et al. [11] of an $O(n \log n)$-time algorithm to compute a sparse Euclidean spanner is incorrect. Thus the problem of devising an $O(n \log n)$-time algorithm to produce sparse spanners remained unsolved.

Sparse spanners are also useful in designing efficient approximation schemes for geometric problems. In a startling development, Rao and Smith [83] showed an optimal $O(n \log n)$-time approximation scheme for the well-known Euclidean *traveling salesperson problem*, assuming that it is possible to compute sparse spanners in time $O(n \log n)$. Also, Czumaj and Lingas [29] showed approximation schemes for minimum-cost multi-connectivity problems in geometrical graphs that also depend on the possibility of computing sparse spanners in time $O(n \log n)$. Since the claim by Arya et al. [11] was incorrect, the existence of an $O(n \log n)$-time algorithm to construct sparse spanners has become a critical open problem. Note that the most efficient algorithm to construct sparse spanners is due to Das and Narasimhan [31] and runs in $O(n \log^2 n)$ time. In this paper we design an algorithm that produces a t-spanner in time $O(n \log n)$, in the standard *real RAM model* defined in [81].

Theorem 8.1 *Given a set V of n points in d-dimensional space, and any real constant t>1, a t-spanner of the complete Euclidean graph can be constructed in $O(n \log n)$ time such that the spanner has $O(n)$ edges and weight $O(wt(MST))$. The constants implicit in the O-notation depend on t and d.*

It was shown in [31] that the greedy algorithm produces spanners with $O(n)$ edges and weight $O(wt(MST))$. However, a naive implementation of the greedy algorithm, shown in Fig. 8.1, takes $O(n^3 \log n)$ time, mainly due to the fact that a quadratic number of shortest path queries are needed to be answered in a "dynamic" graph with $O(n)$ edges. Each of the queries takes $O(n \log n)$ time.

Our algorithm is inspired by the algorithm due to Das and Narasimhan [31]. They showed how to use clustering in order to speed up shortest path queries, i.e., they showed that approximate shortest path queries sufficed to

produce sparse spanners. However, their algorithm was not efficient enough because they were unable to maintain the clusters efficiently and the algorithm had to frequently rebuild the clusters. For convenience, we will refer to the $O(n \log^2 n)$-time algorithm from [31] as the **DN-Clustering** spanner algorithm. We retain the general framework of that algorithm. Our main contribution is in developing techniques to efficiently perform clustering. We believe that the techniques that we have developed are likely to be useful in designing other greedy-style "dynamic algorithms", i.e., in situations where only insertions take place and particularly in increasing order of length. What we prove in this paper is that after some preprocessing, given a linear-sized edge-weighted graph with integral edge weights in the range $[0, N]$, and given a set of cluster centers, then one can perform clustering very efficiently in only $O(n + N)$ time.

The terms length and weight are used interchangeably throughout the paper.

Algorithm STANDARD-GREEDY(G, t)

```
1.    sort the edges in E by increasing weight
2.    E' := ∅
3.    G' := (V, E')
4.    for each edge (u, v) ∈ E do
5.        if SHORTESTPATH(G', u, v) > t · d(u, v) then
6.            E' := E' ∪ {(u, v)}
7.            G' := (V, E')
8.    output G'
```

Figure 8.1: The naive $O(n^3 \log n)$-time greedy spanner algorithm

8.1 The DN-clustering spanner algorithm

We first describe the previous cluster based spanner algorithm due to Das and Narasimhan [31]. It can be roughly described as follows.

The algorithm starts with an empty spanner G'. A preprocessing step helps to eliminate all but a linear number of edges from further consideration. Among the edges not eliminated, very short edges (i.e., those of length at most D/n, where D is the distance between the farthest pair of points) are simply added to G' since their contribution to the overall weight of the spanner cannot be more than the weight of a minimum spanning tree, $wt(MST)$. For the remaining edges, the greedy algorithm is simulated by sorting the edges (by increasing weight) and then processing them in $\log n$ phases. Greedy processing of an edge

$e = (u, v)$ entails a shortest path query, i.e., checking whether $D_{\{G', wt\}}(u, v) \leq t \cdot wt(e)$. If the answer to the query is no, then edge e is added to the spanner G', else it is discarded. Whenever shortest path queries are required to be answered, these are not solved on the spanner G' being constructed. Instead, they are solved on a cluster graph H, which is simultaneously maintained. The cluster graph H from [31] has the following properties:

1. distances in H "closely" approximate distances in the current spanner graph G',
2. every vertex in H has bounded degree, and
3. "specialized" shortest path queries in H can be answered in $O(1)$ time.

The shortest path query when processing edge $e = (u, v)$ is "specialized" in the sense that, at the instant that this query is processed, the cluster graph H only has edges between clusters, so-called inter-cluster edges, whose lengths are within a constant factor of $wt(e)$.

Since the edges considered have weights in the range $(D/n, D]$ and they are processed in $\log n$ phases, the edges can be sorted into $\log n$ bins, where the i-th bin has edges of weight in the range $(2^{i-1} \cdot D/n, 2^i \cdot D/n]$. In order for shortest path queries to be answered quickly, the cluster graph has to be carefully maintained. At the end of each phase, the cluster graph is recomputed from scratch using the graph G'. This was deemed necessary since, in order to answer specialized shortest path queries about edge $e=(u, v)$ in constant time, all inter-cluster edges in H need to be of length within a constant factor of $d(u, v)$.

The time complexity analysis is straightforward. Preprocessing steps ran in $O(n \log n)$ time. The $O(n)$ shortest path queries were processed in $O(n)$ time, since each query took only $O(1)$ time. The cluster graph computation at the start of each phase took $O(n \log n)$ time (since it involved running Dijkstra's shortest path algorithm on linear-sized graphs). Since there were $\log n$ phases, the cluster graph computations took a total of $O(n \log^2 n)$ time. The crucial observation made in [31] was that shortest path queries need not be answered precisely. Instead, approximate shortest path queries suffice to produce low-weight spanners. The second observation was that shortest path queries are expensive if the shortest path involves a number of short edges, and that clustering can help to eliminate all short edges. This, of course, meant that the greedy algorithm, too, was only approximately simulated by the algorithm.

8.1.1 A faster spanner algorithm

In this section, we present a simple modification to the DN-clustering algorithm to construct sparse t-spanners. This algorithm improves on the time complexity

of the DN-clustering algorithm and runs in time $O(\frac{n \log^2 n}{\log \log n})$ in the algebraic decision tree model of computation.

First we observe that there is wide disparity in the overall time spent by the DN-clustering algorithm on shortest path queries ($O(n)$) and the time spent on the cluster graph computations ($O(n \log^2 n)$). In order to balance out the two costs, it is necessary to do fewer than $O(\log n)$ cluster graph computations, which in turn would make the shortest path queries more expensive. Instead of processing the edges in $\log n$ phases, we process them in $\frac{4 \cdot d \cdot \log n}{\log \log n}$ batches. We use the term batches to distinguish from the word phases used by the earlier DN-clustering algorithm.

If the clustering is recomputed after processing every batch of edges the total time for cluster graph computations will be $O(\frac{n \log^2 n}{\log \log n})$, since each call to the clustering algorithm takes $O(n \log n)$ time. We carefully analyze the cost of the $O(n)$ shortest path queries and show that it can now be answered in a total of $O(n \log n)$ time. In phase i of the DN-Clustering algorithm, edges from the i-th bin were processed. These edges had weights in the range $(W, 2W]$, where $W = 2^{i-1}(D/n)$. During phase i, the cluster graph H could have inter-cluster edges whose weights were in the range $(\delta W, 2W(1 + 2\delta)]$, where $\delta < \frac{1}{2}$ is a positive constant. This meant that for edge (u, v) of weight $l \in (W, 2W]$, checking whether there is path from u to v of length at most $t \cdot l$ could be done in $O(1)$ time. More precisely, it was observed in [31] that if there exists a path from u to v of length at most $t \cdot l$, then the number of edges on this path can be at most $\frac{2t}{\delta}$. It was further observed that since the vertices of H had a constant degree bound (say c), and since there are at most $O(c^{\frac{2t}{\delta}})$ vertices that lie $\frac{2t}{\delta}$ edges away from vertex u, this shortest path query could be done in $O(c^{\frac{2t}{\delta}} \log c^{\frac{2t}{\delta}})$ time. A tighter analysis was unnecessary in the DN-Clustering algorithm of [31] since c, t, and δ were all constants; below we show an improved analysis of this cost.

Recall that our algorithm works in $\frac{4 \cdot d \cdot \log n}{\log \log n}$ batches. Batch i of our algorithm can be described as follows. For $W = 2^{\frac{(i-1) \cdot \log \log n}{4 \cdot d}}(D/n)$, the edges processed in batch i have weights in the range $(W, W 2^{\frac{\log \log n}{4 \cdot d}}]$, i.e., they are in the range $(W, W(\log n)^{\frac{1}{4 \cdot d}}]$. This implies that, for edge (u, v) of weight $l \in (W, W(\log n)^{\frac{1}{4 \cdot d}}]$, we need to check whether there is a path from u to v of length at most $t \cdot l$. During batch i, the cluster graph H can have inter-cluster edges whose weights are in the range $(\delta W, (1 + 2\delta) W(\log n)^{\frac{1}{4 \cdot d}}]$. Thus, if there does exist such a path from u to v, then the number of edges on this path can be at most $\frac{t(\log n)^{\frac{1}{4 \cdot d}}}{\delta}$. The crucial observation we make is that the vertices of the cluster graph correspond to clusters of radius δW. These clusters may overlap, but their centers can lie in only one cluster. In other words, if these clusters are shrunk in half, they do not intersect. Thus the vertices correspond to disjoint

clusters of radius $\delta \cdot W/2$. Now, it is possible to bound the number of vertices within distance at most $t \cdot l = tW(\log n)^{\frac{1}{4 \cdot d}}$. A simple packing argument shows that the number of balls of radius r that can be packed in a ball of radius R is bounded by $O((R/r)^d)$, where d is the dimension of the space. In our case, the number of balls of radius $r = \frac{\delta W}{2}$ that can be packed in a ball of radius $R = tW(\log n)^{\frac{1}{4 \cdot d}}$ is at most $O((\frac{t \cdot (\log n)^{\frac{1}{4 \cdot d}}}{\delta})^d)$. Thus, due to the constant degree, the maximum number of vertices and edges that can be reached when performing Dijkstra's algorithm starting from vertex u is $O((\frac{t \cdot (\log n)^{\frac{1}{4 \cdot d}}}{\delta})^d)$. Since t, d and δ are constants, we have that $O((\frac{t \cdot (\log n)^{\frac{1}{4 \cdot d}}}{\delta})^d) = O((\log n)^{\frac{1}{4}})$. We conclude that Dijkstra's algorithm for a shortest path query has a time complexity of $O((\log n)^{\frac{1}{4}} \cdot (\log((\log n)^{1/4}))) = O(\log n)$.

The obvious consequence is that all $O(n)$ shortest path queries can be answered in $O(n \log n)$ time, and hence, we have proved the following theorem:

Theorem 8.2 *In the algebraic decision tree model of computation, given a set V of n points in d-dimensional space, and any real constant $t>1$, a sparse t-spanner of the complete Euclidean graph can be constructed in $O(\frac{n \log^2 n}{\log \log n})$ time. The constants implicit in the O-notation depend on t and d.*

8.2 A fast spanner algorithm

In the rest of the paper, we describe an efficient algorithm to construct sparse spanners with a running time of $O(n \log n)$. This algorithm is also inspired by the DN-clustering algorithm in [31]. As explained in Section 8.1 the reason why their algorithm runs in time $O(n \log^2 n)$ is that the clustering step takes $O(n \log n)$ time per phase. The running time for our algorithm is achieved by designing a linear time algorithm for an "approximate" version of the clustering step, thus executing all the clustering steps in $O(n \log n)$ total time.

One crucial idea that we employ to speed up the clustering is to replace the real-valued edge weights by integral values. As observed in [31], the shortest path queries required by the algorithm need not be answered precisely; approximately correct answers suffice. A convenient way to achieve the *integralization* is to use the *floor/ceiling* function. However, this assumes a more powerful model of computation. In order to get around this problem, we reduce the dependence of the algorithm on the floor/ceiling function and compute the floor/ceiling function by using operations allowed under the real RAM model. The second crucial component of our algorithm is an implementation of the clustering algorithm in $O(n)$ time assuming small integral edge weights for the edges. We also prove that the integralization introduces only a bounded amount of error, and that

this error retains the correctness of the other required operations.

The improved spanner algorithm can be roughly described as follows; see Fig. 8.2. It is important to note that the skeleton of the algorithm is similar to the DN-clustering algorithm from [31]. In particular, this improved algorithm also runs in $O(\log n)$ phases. If a fewer number of phases are used, then the error due to integralization could be too large. Even if a fewer number of phases can be used, the running time of the overall algorithm will remain as $O(n \log n)$, since it is dominated by other steps in the algorithm. In particular, the integralization itself has an initial cost of $O(n \log n)$.

The algorithm starts with an empty spanner G' and employs the same preprocessing step to eliminate all but a linear number of edges. This step is done by a call to the t-spanner algorithm presented by Arya et. al. in [11], with $\sqrt{t/t'}$ as input parameter, for some t' such that $t \geq t' > 1$. Note that this algorithm in [11] is correct and runs in time $O(n \log n)$. It also guarantees that the graph has constant degree. As before, short edges of length at most D/n are simply added to G'; their contribution to the overall weight of the spanner is bounded by $wt(MST)$. The greedy algorithm is now simulated on the remaining edges and the edges are added to the graph G'.

The edges of the graph have real-valued weights that are equal to the Euclidean distance between their endpoints. The edges are sorted by increasing weight and then processed in $\log n$ phases. Each of the edges in the spanner graph also have corresponding integer-valued weights that are sufficiently close approximations of the real-valued weights; these integer-valued weights change through the course of the algorithm. In order to distinguish between the real- and integer-valued weights, we assume that there are two different weight functions defined on the edges of G'. For edge $e = (u, v)$, the real-valued weight function $wt(e)$, as mentioned before, is defined as the Euclidean distance $d(u, v)$ between u and v. The integer-valued weight function denoted by $Iwt_i(e)$ is a function of $wt(e)$ and the phase number i. It is maintained during the execution of the algorithm as will be described later. Whenever the phase number is clear by the context, we use the simpler notation $Iwt(e)$ instead of $Iwt_i(e)$. Also, unless specified otherwise, we assume that when we refer to the weight of an edge, we are referring to the real-valued weight of the edge.

At the start of each phase, the integer-valued weight function $Iwt(e)$ is recomputed for this phase. Then a set of vertices of G' are selected as cluster centers and a cluster graph H is constructed from the current spanner graph G', using the weight function Iwt. This cluster graph H is a simpler graph than the graph G' and distances between vertices in H are reasonably close to distances between the same pair of vertices in G'. The difference of this from the one in [31] lies in the fact that the cluster centers have to be selected before the

clustering is done and the clustering is done with the weight function Iwt. As mentioned before, we improve on the time complexity of this clustering step and show how it can be implemented to run in $O(n)$ time. Once the cluster graph H is constructed, the algorithm processes the set of edges for that phase. Greedy processing of an edge $e = (u, v)$ entails, as before, a shortest path query, i.e., checking whether $D_{\{G', wt\}}(u, v) \leq t \cdot wt(e)$. This query is answered in constant time per query by performing an approximate shortest path query on the simpler graph H, and not on the partial spanner graph G'. If the answer to the query is 'no', edge e is added to the graph G', else it is discarded. Each of the steps is described in more detail in the rest of the paper.

In Section 8.2.1 we describe the integralization process and analyze the error of the integralization. The clustering algorithm is described in Section 8.2.2, and finally, in Section 8.2.3 we describe how to compute the shortest paths in the cluster graph and prove that the total running time of the algorithm is $O(n \log n)$.

The detailed algorithm is given below in figure 8.2. The inputs are, V, which is a set of n points in d-dimensional space, and two constants t and t' such that $1 < t' \leq t$. As one can see, it is similar to the DN-clustering algorithm, except for the integralization steps (steps 9, 12 and 13) and the computation of the cluster centers (steps 10 and 20). In Sections 8.3 and 8.4 we will show that the output G' indeed is a t-spanner, and that a suitable selection of the input parameter t' will guarantee that G' has small weight. Recall that the really time-critical step of this algorithm is the clustering step (step 14) which will be closely described in Section 8.2.2. Step 19 of the algorithm is needed to maintain the cluster graph H, i.e., edges between clusters that lie close to each other are added. A more detailed description of this step is also given in Section 8.2.2. Note that the two values bounding δ are decided in Lemma 8.14 and Lemma 8.17.

8.2.1 Integralization

As mentioned before, in order to speed up the cluster graph computation, we replace the real-valued edge weights by integral values. The integralization changes in every phase. It is done in such a way that the edge weights and distances encountered in that phase are always in the range $[0, N]$, where $N = c \cdot n$ for some constant integer c. The choice of c will dictate the errors introduced in the distance computations; this will be discussed later.

A closer inspection of a phase leads to the following simple observations. At the start of phase i, the spanner graph constructed so far has edges of weight at most W_i. During phase i, the edges considered for inclusion by the greedy algorithm are in the range $(W_i, 2W_i]$. The shortest path query for an edge of

Algorithm IMPROVED-GREEDY(V, t, t')

1. Compute a $(\sqrt{t/t'})$-spanner $G = (V, E)$ using the algorithm from [11]
2. $\delta := \min\left(\frac{\sqrt{tt'}-(1+\epsilon)t'}{2(1+\epsilon)(\sqrt{tt'}+3t')}, \frac{\sqrt{tt'}-(1+\epsilon)}{2(\sqrt{tt'}(1+\epsilon)+5+7\epsilon+2\epsilon^2)}\right)$
3. $D :=$ length of longest edge in E
4. $E' = \{e \in E \mid wt(e) < D/n\}$
5. $G' := (V, E')$
6. $W_i := 2^{(i-1)}D/n$ for $i = 1, 2, \ldots, \log n$
7. **for** $i := 1$ to $\log n - 1$ **do**
8. $\quad E_i :=$ set of (sorted) edges of E with weights in $(W_i, W_{i+1}]$
9. INTEGRALIZE$(E', 0)$
10. $C_1 :=$ NAIVE-CENTERS$(G', r = \delta W_1)$;
11. **for** $i := 1$ to $\log n$ **do**
12. $\quad$ INTEGRALIZE(E_i, i)
13. $\quad$ REINTEGRALIZE(E')
14. $\quad H :=$ CLUSTER-GRAPH$(G', Iwt, C_i, r = \delta W_i, R = W_i)$
15. $\quad$ **for** each edge $e = (u, v) \in E_i$ in increasing order **do**
16. $\quad\quad$ **if** not SHORT-PATH$(H, u, v, \sqrt{tt'} \cdot d(u, v))$ **then**
17. $\quad\quad\quad E' := E' \cup \{e\}$
18. $\quad\quad\quad G' := (V, E')$
19. $\quad\quad\quad$ ADDINTRAEDGESTYPE2(u, v)
20. $\quad C_{i+1} :=$ UPDATE-CENTERS$(H, i, C_i, r = \delta W_i)$
21. output G'

Figure 8.2: The $O(n \log n)$-time spanner algorithm

length l involves checking whether the distance between a given pair of vertices is at most $t \cdot l$. Hence the longest paths that need to be dealt with during phase i are of weight $t \cdot 2W_i$. The idea is to make the largest distance we consider in phase i to correspond to the integer $c \cdot n$. To be on the safe side, since there are small errors in the distance computations, we set $2(t \cdot 2W_i)$ to correspond to $c \cdot n$. Thus, in phase i, unit integer length will correspond to real length of $U_i = \frac{4 \cdot t \cdot W_i}{c \cdot n}$.

Although a constant-time floor/ceiling function is not used in the algorithm, a convenient way to describe the integralization is as follows:

$$Iwt_i(e) := \left\lceil \frac{wt(e)}{U_i} \right\rceil.$$

We will describe below how the integralization step is performed.

Error bounds:

Assuming the integralization defined above, we observe that the function Iwt always involves a rounding up. Hence, $Iwt_i(e) \cdot U_i \geq wt(e)$. It follows that in phase i, the error in the length of any single edge of the spanner graph is at most U_i. In other words, $Iwt_i(e) \cdot U_i - wt(e) \leq U_i$. Note that this error is an additive or an absolute error. Since any simple path can use at most $n-1$ edges, it is also easy to see that the error in the length of any simple path of the spanner graph is less than nU_i. Another consequence is that given two simple paths P_1 and P_2, if $Iwt(P_1) = Iwt(P_2)$, then $|wt(P_1) - wt(P_2)| < nU_i$. It follows that nU_i is also a bound on the error that can be introduced when running Dijkstra's single-source-shortest-path algorithm using the integral weights instead of the real weights. The following lemma formalizes this statement:

Lemma 8.3 *In phase i, given any $\epsilon > 0$ and two vertices u and v in G' such that $D_{\{G',wt\}}(u,v) > W_i$, it holds that*

$$D_{\{G',wt\}}(u,v) \leq D_{\{G',Iwt\}}(u,v) \cdot U_i < (1+\epsilon) \cdot D_{\{G',wt\}}(u,v).$$

Proof: We give a sketch of the proof. Recall that the cluster radius is $r = \delta W_i$, where $\delta < 1/2$ is some positive constant. In phase i, for a path P such that $wt(P) \geq W_i$, the error in computing its weight is at most nU_i. Thus the relative error is at most $nU_i/W_i = \frac{4t}{c}$. The proof follows by setting $\epsilon = \frac{4t}{c\delta} > \frac{4t}{c}$ and using the well-known property of Dijkstra's algorithm that the minimum value in the priority queue is monotonically non-decreasing. Note that ϵ can be made as small as desired by choosing an appropriate value of c. □

As a direct consequence we obtain the following important corollary.

Corollary 8.4 *For a path P in G' with $wt(P) \geq \delta W_i$, the absolute error in computing its weight is at most nU_i, and the relative error is at most $\frac{nU_i}{\delta W_i} = \epsilon$, for any $\epsilon > 0$.*

Computing the integralization

Here we show how to compute the integer values of the weights of the edges over all phases in $O(n \log n)$ total time without using the floor/ceiling function.

We first observe that the spanner graph has at most $O(n)$ edges at the start of any phase. Consider a specific phase i. In this phase, for a specific edge e, since its integer value is in the range $[0, N]$ (where $N = c \cdot n$), $Iwt(e)$ can be computed in $O(\log n)$ time without the use of the floor/ceiling function by performing a binary search on the set of real values $j \cdot U_i$, for $j = 0, \ldots, N$. We assume that the function INTEGRALIZE(E_i, i) performs this operation for each edge in the set E_i in $O(\log n)$ time per edge.

If the above observations are used in a naive fashion for all edges, then the cost of integralization is $O(n \log n)$ just for one phase. Since the number of phases is not constant, the integralization would turn out to be too expensive. We have to show that the algorithm spends $O(\log n)$ time for computing the integralization of an edge weight over all the phases. The idea is to compute the integral value in $O(\log n)$ time when the edge is encountered for the first time. Integralizations of an edge for subsequent phases is done by calling ReIntegralize, and are computed in constant time from the integer weights of the edge computed in the previous phase. If the integral weight of an edge is I in phase i, then the integral weight of the edge in phase $i+1$ will be $I/2$ if I is even, and $(I+1)/2$ if it is odd. This is correct since $U_{i+1} = 2U_i$, i.e., the integralization in phase $i+1$ is twice as coarse as that in phase i. Checking if an integer is odd or even cannot be done in constant time in the real RAM model, but can be easily accomplished by using $O(n \log n)$ preprocessing. One way to accomplish this would be to build a balanced binary tree including $c \cdot n$ elements with the values $1, \ldots, cn$. Every element in the tree, with value val, also contains a pointer to the element in the tree containing the value $\lceil \frac{val}{2} \rceil$. This value can be computed in time $O(\log n)$ and searching the tree for the next value can be done in time $O(1)$. Hence, by using $O(n \log n)$ time preprocessing, the integral weight of an edge for the next phase can be computed in constant time. Another way to handle this problem would be to extend the model of computation with trigonometric functions, i.e., the sine function.

Note also that the relative error for an edge with newly computed weight is less than U_{i+1}, hence Lemma 8.3 still holds. It is clear that ReIntegralize(E') performs its operation for each edge in the edge set E' in $O(1)$ time per edge.

The above explanation proves that the integralization is computed in time $O(n \log n)$ for all edges over all phases. The integer weights are then used directly in the clustering algorithms described below.

8.2.2 Clustering the graph

Now we turn our attention to the main contribution of this paper, namely how to construct a cluster graph in linear time. First some definitions. Here we assume that $G = (V, E)$ is a metric graph with a weight function w defined on its edges E. The following definition of a cluster is modified from the one in [31] to allow for arbitrary weight functions. The definition of a cluster cover is also modified and is defined for a given set of cluster centers.

Definition 8.5 Cluster, cluster center, and radius
Given a vertex $v \in V$ and a real value r, Cluster(G, v, r, w) is defined as the

set of all vertices $U \subseteq V$ such that $D_{\{G,w\}}(v,u) \leq r$ for all $u \in U$. The vertex v is called the *cluster center* of this cluster and r is called the *radius* of the cluster.

Definition 8.6 Cluster-cover
Given a set of cluster centers $C = \{v_1, v_2, \ldots, v_m\} \subseteq V$ and a radius r, the CLUSTER-COVER(G, C, r, w) (if it exists) is a set of clusters $K = \{K_1, \ldots, K_m\}$ such that K_i is a cluster with radius r and cluster center v_i, $1 \leq i \leq m$, and such that $K_1 \cup K_2 \cup \cdots \cup K_m = V$.

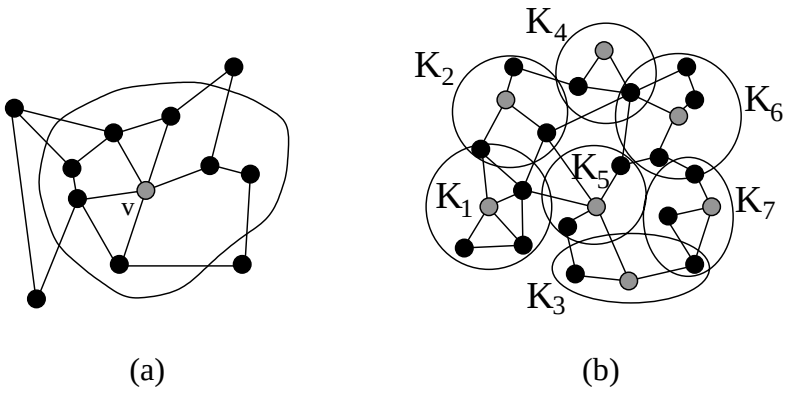

Figure 8.3: (a) A cluster with center at v and radius r. (b) The clusters $K_1, \ldots, K_7$ form a cluster cover.

The set C and the radius r will be chosen in such a way that the cluster cover always exists. In general, clusters in a cluster cover may overlap. We also modify the definition from [31] of a cluster graph so that it is a bit more general and it is defined for a given set of clusters and for an arbitrary weight function.

Definition 8.7 Cluster graph
Assume that $C = \{v_1, v_2, \ldots, v_m\} \subseteq V$ is a given set of cluster centers. For a given radius r, we assume that $K = \{K_1, K_2, \ldots, K_m\}$ is equal to CLUSTER-COVER(G, C, r, w). Given a second radius $R > r$, CLUSTER-GRAPH(G, w, C, r, R) is defined as a graph $H = (V, E_H)$ with a weight function w defined on its edges E_H. The weight of an edge $[u, v]$ in E_H is defined to be equal to $D_{\{G,w\}}(u, v)$ (we use square brackets to distinguish cluster graph edges from the edges of G). The edges of H are defined as follows.

Intra-cluster edges: For all K_i, and for all $u \in K_i$, $[u, v_i] \in E_H$.

Inter-cluster edges: For all $v_i, v_j \in C$, $[v_i, v_j]$ is an inter-cluster edge if either:

1. $v_i \notin K_j$ and $v_j \notin K_i$ and $D_{\{G,w\}}(v_i, v_j) \leq R$ (Type 1), OR
2. there exists $e = (u_i, u_j) \in E$ such that $u_i \in K_i, u_j \in K_j$ (Type 2).

Computing the cluster cover

Here we describe how the cluster cover is computed efficiently under some assumptions. Once a cluster cover is computed we show, in the next section, that it is straight-forward to construct the cluster graph.

Note that the input to the cluster cover computation is a weighted graph $G = (V, E)$ with a weight function w defined on its edges, a set $C \subseteq V$ of cluster centers and a radius R. We will assume that $|V| = n$, $|E| = O(n)$, the weight function w is integral, and the radius R is an integer. Since we do not have to deal with distances greater than R, we can safely assume that the weight of any edge is an integer value in the range $[0\,..\,R]$. We will further assume that the cluster centers are chosen in such a way that a cluster cover exists, which will be shown in Section 8.2.3. The obvious way to implement this algorithm is as was done in [31], i.e., to run Dijkstra's Single-Source-Shortest-Path algorithm from all the cluster centers and to compute the clusters in the cluster cover. However, this has a running time of $O(n \log n)$. In order to speed it up, we run Dijkstra's algorithm in **parallel** from all the cluster centers and use a simple and fast priority queue, which we denote by PQ. The priority queue we use is an array of size R, indexed from 1 to R, as shown in Fig. 8.4. This is sufficient for our purposes because of the following reasons. Firstly, the weight function is integral and the array contains all possible distance values from the cluster centers to vertices in the clusters. Secondly, it is well-known that in Dijkstra's algorithm, once a vertex has been extracted from the priority queue, its distance from the source will never be updated again and the distance from the source at the time of the extraction is the correct distance from the source. In other words, the minimum value of the items in the priority queue is monotonic. Since the priority queue is an array, EXTRACT-MIN can be implemented as a scan through the array for the "next" largest item.

One problem is that clusters can overlap and that vertices may have entries in the priority queue with distances from several cluster centers. Let c_v denote the maximal number of clusters that a vertex may belong to. The problem can be taken care of by augmenting the priority queue entries to be a pointer to a linked list where every entry in the list also stores information about the vertex as well as the corresponding cluster center. Since a vertex at most belongs to c_v clusters the space complexity of the priority queue will be $O(n \cdot c_v + R)$. Also, every vertex contains a list of the clusters it belongs to.

It should be noted that Dijkstra's algorithm, as shown below, needs to per-

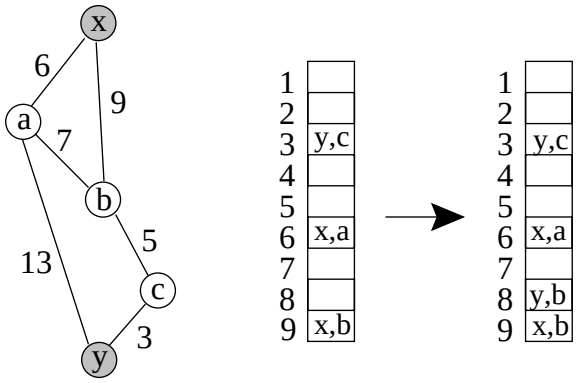

Figure 8.4: An example of how the cluster cover is computed with x and y as cluster centers and radius $R = 9$. The array to the left is the initial priority queue after all edges leading out of x and y have been processed. The array to the right shows the final priority queue. Note that after vertex c was processed, the edge (c, b) was relaxed and (y, b) was inserted into the priority queue with length 8.

form a number of RELAX steps and that in each such step the priority queue may need to be updated. The process of RELAXing an edge (u, v) consists of testing whether we can improve the shortest path to v found so far by going through u and, if so, updating the value for v, i.e., adding a new entry and removing an old entry. Since every vertex contains information about which clusters it belongs to and the distance to each cluster center, each update is performed in time $O(c_v)$. It should be pointed out that this is the only place where we are unable to eliminate the use of *Random Access* since it is critical that this update be performed efficiently, i.e., in constant time. Also note that an edge (u, v) may be RELAXed several times ($O(c_v)$ times), each time with respect to a different cluster center.

Algorithm PARALLELDIJKSTRA

Q=INITIALIZE(G', cluster centers)	$O(n)$
while $Q \neq \emptyset$ **do**	$O(n \cdot c_v)$
u =EXTRACTMIN(Q)	
for each vertex v adjacent to u **do**	$O(1)$
RELAX(u, v)	$O(c_v)$

end

Thus the time and space complexity of the algorithm is affected by the amount of overlap of the clusters in the cluster cover. The space complexity of

the data structure is $O(n \cdot c_v + R)$. Furthermore, the RELAX-operation has a running time of $O(c_v)$ and, the total number ($O(n \cdot c_v)$) of EXTRACTMIN-operations can be performed in total time $O(n \cdot c_v + R)$. A careful implementation of cluster cover can be made to run in time $O(n \cdot c_v^2 + R)$.

Computing the cluster graph

Now we are ready to describe how to compute the cluster graph. The input is a weighted graph G with a weight function w, a set of cluster centers $C = \{v_1, \ldots, v_m\}$, and two different radii r and R, where $R > r$. In order to compute the cluster graph, the algorithm computes a cluster cover from the same set of cluster centers but with the two radii, r and R. Let the cluster covers with radii r and R be denoted by K_r and K_R respectively. We augment the cluster cover procedure to also produce a data structure that supports the following queries for both the cluster covers:

FINDCENTERS(v, K)**:** Given $v \in V$, it returns all cluster centers v_i such that v is in a cluster from K centered at v_i, i.e., $D_{\{G,w\}}(v, v_i)$ is at most the radius of the clusters in K. It also returns $D_{\{G,w\}}(v, v_i)$ for these cluster centers.

COMPUTEDISTANCE(v_i, v)**:** Given $v \in V$, and a cluster center v_i, it returns $D_{\{G,w\}}(v, v_i)$ if $D_{\{G,w\}}(v, v_i) \leq R$; otherwise, it returns the value ∞.

Now the cluster graph $H = (V, E_H)$ is computed easily as follows. The intra-cluster edges of H are computed by performing FINDCENTERS queries for each vertex $v \in V$ in the cluster cover K_r and adding the corresponding edges. Recall that FINDCENTERS returns a set T of 2-tuples, where each tuple t consists of a vertex $t.w$ and its distance $t.d$ to v. The algorithm is described below in pseudo code, and has a running time of $O(n \cdot c_v)$.

Algorithm INTRAEDGES
  for every vertex $v \in V$ **do**
    $T :=$FINDCENTERS(v, K_r)
    for every element $t \in T$ **do**
      ADDINTRAEDGE$(v, t.w, t.d)$
end

From Definition 8.7 we have that the inter-cluster edges can be of two types, type 1 and 2. An edge $[v_i, v_j]$ of type 1 is added if $v_i \notin K_j$ and $v_j \notin K_i$ and $D_{\{G,w\}}(v_i, v_j) \leq R$, where $K_i, K_j \in K_r$ are clusters with centers at v_i and v_j. For every cluster center v_i, we use the FINDCENTERS query to list all the clusters from K_R that it is contained in. The centers v_j of these clusters satisfy the

condition that $D_{\{G,w\}}(v_i, v_j) \leq R$. Now we use the ComputeDistance queries to make sure that $v_i \notin K_j$ and $v_j \notin K_i$. Adding the inter-cluster edges of type 1 is done in time $O(n \cdot c_v^2)$.

Algorithm InterEdgesType1
  for every cluster center v in K_r **do**
    T :=FindCenters(v, K_R)
    for every element $t \in T$ **do**
      If ComputeDistance$(v, t.w) \geq r$ **then**
        AddInterEdge$(v, t.w, t.d)$
end

The time complexity of computing the cluster graph is $O(n \cdot c_v^2)$. Having the cluster centers selected before performing the clustering enables clusters to be grown in "parallel" and thus the above algorithm is able to use one common priority queue to grow all the clusters, and is consequently able to perform the clustering efficiently.

Maintaining the cluster graph

An edge $[v_i, v_j]$ of type 2 is added if there exists an edge $e = (u_i, u_j) \in E$ such that $u_i \in K_i$ and $u_j \in K_j$. During the computation of the cluster graph H, only intra-cluster edges and inter-cluster edges of type 1 are added. Additional edges may be added during a phase of the greedy algorithm. Every time the greedy algorithm decides to add an edge $e = (u, v)$ to the partial spanner graph, several inter-cluster edges of type 2 may be added to H. This is achieved as follows: for every edge $e = (u_i, u_j)$ that is to be added to G', perform FindCenters queries for u_i and u_j from K_r and join the corresponding cluster centers by inter-cluster edges in H. The weight of such edges are computed by performing two ComputeDistance queries for u_i and u_j with the corresponding cluster centers and adding it to the weight of (u_i, u_j). It is clear that the function below runs in $O(c_v^2)$ time, and it is performed $O(n)$ times.

Algorithm InterEdgesType2(u_i, u_j)
  $T1$:=FindCenters(u_i, K_r)
  $T2$:=FindCenters(u_j, K_r)
  for every $t1 \in T1$ **do**
    for every $t2 \in T2$ **do**
      AddInterEdge$(t1.w, t2.w, t1.d + w(u_i, u_j) + t2.d)$
end

Selecting the cluster centers for a phase

In order for the CLUSTER-GRAPH function to be implemented efficiently, it needs to have the set of cluster centers as input. For the first phase, the cluster centers C_1 are identified in a greedy fashion using the weighted graph $G' = (V, E')$ with real-valued edge weights, and using a radius of $r = \delta W_1$. That is, select any point $v \in V$, not belonging to any clusters already computed, as a cluster center, compute the cluster with center at v. Continue this procedure until all points in V belong to a cluster. This is referred to as NAIVE-CENTERS in the algorithm given in Fig. 8.2. NAIVE-CENTERS runs in $O(n \log n)$ time, since this can be implemented using the standard Dijkstra's algorithm. For subsequent phases, cluster centers are identified (using UPDATECENTERS) in a different way. The set of cluster centers are always chosen as a subset of the cluster centers used in the previous phase.

At the end of each phase, the algorithm selects a set of cluster centers for the next phase. These centers are guaranteed to be sufficiently far apart from each other. More specifically, the cluster centers C_i used in phase i are guaranteed to be at a distance of at least $r_i/2$.

In phase i, the set of cluster centers for phase $i+1$ is computed as $C_{i+1} := C_i \backslash M_i$, i.e., a subset M_i of the cluster centers are deleted from the list of cluster centers. We now describe how the set M_i is chosen. M_1 is the empty set, implying that C_2 is identical to C_1. For $i > 1$, the algorithm picks a cluster center from C_i and marks all cluster centers that are within distance r from it. The cluster centers that are marked are inserted into M_i and hence, deleted in the next phase. This is easily implemented by calling the FINDCENTERS after the cluster cover for phase i has been computed. The next cluster center is then picked and the process continues until all centers have been processed. Finally, $C_{i+1} := C_i \backslash M_i$. Clearly this process runs in time $O(m \cdot c_v)$. (It is important to note that since the integralization changes in every iteration, vertices that are at distance r' in one iteration are at distance $r'/2$ in the next iteration.)

We now show that in phase i the cluster centers are guaranteed to be at a distance of at least $r/2$ from each other. In phase 1, since cluster centers are identified by using a radius of r, all cluster centers are at a distance of at least r from each other. In phase $i-1$, if two cluster centers are at a distance of r or less, then one of them will get marked, and will subsequently be deleted from the list C_i for phase i, as shown in Fig. 8.5. Lemma 8.8 specifies conditions under which vertices belong to at most a constant number of clusters. It follows from this lemma that no vertex of H is in more than a constant number of clusters of radius r or of radius R (since $\frac{R}{r} = \frac{1}{\delta}$).

Lemma 8.8 *Let $C = \{v_1, \ldots, v_m\} \subseteq V$ be a set of vertices such that for any*

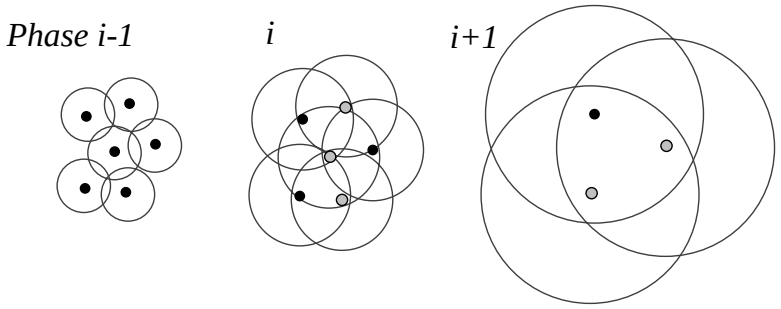

Figure 8.5: If a cluster contains another cluster center then this cluster center is marked for deletion in the next phase. The figure shows an example of a set of cluster centers where the cluster centers that are marked for deletion for the next phase are marked in grey.

pair of vertices $v_i, v_j \in C$, $D_{\{G,w\}}(v_i, v_j) > r'$. If $K = \{K_1, \ldots, K_m\}$ is returned by Cluster-Cover$(G, C, c' \cdot r')$*, where c' is a constant, then each vertex $v \in V$ is contained in at most a constant (which depends on the dimension d and c') number of clusters from K.*

The conditions of the lemma are true for the cluster graph as constructed above with $r' = r/2$ and $c' = 2$ or $c' = 4t/\delta$. Hence any vertex in H is part of at most a constant number of clusters in K_r or K_R. The proof follows from standard packing arguments, see also Section 8.1.1. Similar arguments also show that the number of inter-cluster edges incident to a cluster center is also a constant (although it might have a large number of intra-cluster edges). It follows that the degree of any vertex in H that is not a cluster center must be a constant, and the size of H is $O(n)$. Note that Improved-Greedy use integralized weights so the resulting clusters are approximate clusters; they are a little bit larger (since integers are always rounded up) than the exact clusters. It is clear that this does not affect the correctness of Lemma 8.8.

8.2.3 Answering shortest path queries

When the algorithm Improved-Greedy considers an edge $e = (u, v)$ for inclusion in the spanner graph, it needs to answer a shortest path query. It needs to check if $D_{\{G', wt\}}(u, v) \leq t \cdot d(u, v)$, where G' is the spanner graph constructed so far. As noted in [31], it is sufficient for this query to be answered approximately. So, it is sufficient to devise a procedure to efficiently check if $D_{\{G', wt\}}(u, v) \leq t(1+\epsilon') \cdot d(u, v)$, for some small $\epsilon' > 0$. In other words, it is sufficient to check if $D_{\{G', Iwt\}}(u, v) \leq t(1 + \epsilon'') \cdot d(u, v)/U_i$, for some small $\epsilon'' > 0$.

In fact, the algorithm will check if $D_{\{H,Iwt\}}(u,v) \leq t \cdot d(u,v)/U_i$. The time complexity of this test is constant if $D_{\{H,Iwt\}}(u,v) < c' \cdot r$, for some constant c'. Hence, we conclude this section with the following theorem, which follows from the above arguments.

Theorem 8.9 IMPROVED-GREEDY *runs in time* $O(n \log n)$.

Proof: Following the steps of the IMPROVED-GREEDY algorithm shown in Fig. 8.2, we have that step 1 in the initialization and steps 8 and 9 take time $O(n \log n)$; step 2 take constant time; step 6 run in $O(\log n)$ time, while step 3 takes $O(n)$ time. The integralization of the weight of the edges from steps 9, 12 and 13, takes a total of $O(\log n)$ time per edge. Since each edge is considered exactly once the total time spent on integralizing and re-integralizing the weight of the edges is $O(n \log n)$ according to Section 8.2.1. Step 14 requires linear time since c_v is bounded by a constant. On line 15 every edge in the input spanner is considered once. For each edge the algorithm performs one shortest path query in the cluster graph. As mentioned above, each query takes constant time. Hence the total time complexity for computing a linear number of shortest path queries is $O(n)$. Finally, updating the centers is easily done in linear time. From this it follows that IMPROVED-GREEDY runs in time $O(n \log n)$.

□

In 1999, Thorup [92] showed that single source shortest path queries could be answered in linear time for undirected graphs with integer edge weights. However this algorithm was not used in this paper since it does not visit the vertices in order of increasing distance, which is crucial for our algorithm. Also, it uses bit-shift for computing the floor function in constant time, which we do not allow in the real RAM model.

8.3 The graph produced by IMPROVED-GREEDY is a t-spanner

In order to show that the produced spanner graph G' is a t-spanner we need two main results. First, we need to show that the cluster graph H approximates the spanner graph G', i.e., $D_{\{G',wt\}}(v,u) \leq D_{\{H,Iwt\}}(v,u) \leq \alpha D_{\{G',wt\}}(v,u)$ for some constant α close to 1. This is done in Lemmas 8.12 and 8.13. Secondly, we need to show, Lemma 8.14, that H always is a valid cluster graph of G'. From these results we easily obtain Theorem 8.15, which says that the produced spanner G' is a t-spanner of the complete Euclidean graph.

Since the clusters are computed using the function $Iwt(\cdot)$ instead of $wt(\cdot)$, clusters are not as precise as they were in [31]. In this section we will assume

that the smaller radii r_i is δW_i and the larger radii R_i is W_i, where δ is a positive constant decided in Lemma 8.14 and 8.17. Finally, we set $\epsilon = \frac{nU_i}{\delta W_i}$. Some of the results in this section, and the next, are modified versions of analogous results in [31].

Lemma 8.10 *Let K be equal to* CLUSTER$(G', v, \delta W_i, Iwt_i)$, *i.e., a cluster with cluster center v and radius δW_i computed in iteration i of the algorithm. If u is a vertex in K, then $D_{\{G',wt\}}(v,u) \leq (1+\epsilon)\delta W_i U_i$. Otherwise, if $u \notin K$, then $D_{\{G',wt\}}(v,u) > \delta W_i U_i$.*

Proof: The lemma follows from Corollary 8.4 and the fact that a cluster from K with center at v consists of all vertices within integer distance δW_i from v. □

Consider the cluster graph H that results from the clustering performed on G' at the start of phase i. The following results apply to edges and paths in H.

Lemma 8.11 *If u is a cluster center and $[u,v]$ is an intra-cluster edge in H, then*

$$D_{\{G',wt\}}(u,v) \leq (1+\epsilon)\delta W_i U_i. \tag{1}$$

If $[v_j, v_k]$ is an inter-cluster edge in H, then

$$\delta W_i U_i < D_{\{G',wt\}}(u,v) \leq (1+\epsilon)\cdot(W_i + 2\delta W_i)U_i. \tag{2}$$

Proof: The first statement is a direct consequence of Lemma 8.10, the same holds for the left inequality in (2). The right inequality in the second statement follows from Definition 8.7 since an inter cluster edge of type 2 may be constructed to connect two cluster centers v_j and v_k since there exists an edge $(x,y) \in G'$ such that $x \in K_j$ and $y \in K_k$ and $Iwt(x,y) < W_i$. □

For simplicity in the rest of this section we will leave out the unit length U_i. The following lemma is straight-forward since H is an approximation of G'.

Lemma 8.12 *If there exists a path P_H in H between vertices u and v such that $Iwt(P_H) = L$, then there exists a path $P_{G'}$ in G' between vertices u and v such that $Iwt(P_{G'}) \leq L$. We first introduce some definitions. A vertex u is defined to* sufficiently far *from a vertex v if, (1) no single cluster contains both u and v, and (2) $D_{\{G',wt\}} \geq W_i$. Define a* cluster path *in H to be a path where the first and last edges may be intra-cluster edges, but all intermediate edges are inter-cluster edges.*

The next lemma is the approximate converse of Lemma 8.12.

Lemma 8.13 *Let u be sufficiently far from v. Let $P_{G'}$ be a path between u and v in G' such that $wt(P_{G'}) = L_1$. Then there exists a cluster path P_H between u and v in H such that*

$$Iwt(P_H) = L_2 < L_1 \cdot \frac{(1+\epsilon)(1+6\delta)}{1-2\delta(1+\epsilon)}.$$

Proof: Let the path from u to v having weight L_1 in G' be P. We shall use the notation $P(y,x)$ to denote the vertices of P between vertices y and x, not including y. We construct a cluster path Q from u to v in H with weight L_2 as follows. Let C_0 be any cluster, with center v_0, containing u. The first edge of Q is the intra-cluster edge $[u, v_0]$. Next, among all clusters with centers adjacent to v_0 in H, let C_1, with center v_1, intersect the furthest vertex along $P(u,v)$, say w_1. Add the inter-cluster edge $[v_0, v_1]$ to Q. Next, among all clusters with centers adjacent to v_1 in H, let C_2, with center v_2, intersect the furthest vertex along $P(w_1, v)$, say w_2. Add the inter-cluster edge $[v_1, v_2]$ to Q. This process continues until we reach a cluster center, v_m, whose cluster contains v. At this stage complete Q by adding the intra-cluster edge $[v_m, v]$, as shown in Fig 8.6. We now prove that the weight of Q satisfies the lemma statement.

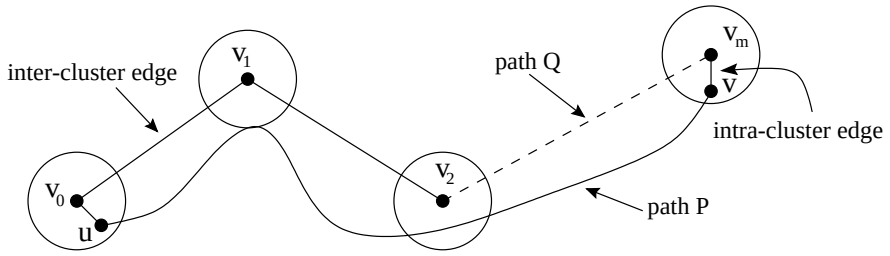

Figure 8.6: Paths in H approximate paths in G'.

Case 1. $[m = 1]$. In this case there is only one inter-cluster edge along Q. Since u is sufficiently far from v, we know that $L_1 > W_i - 2(1+\epsilon)\delta W_i$. Now $L_2 = Iwt([u,v_0]) + Iwt([v_0,v_1]) + Iwt([v_1,v])$. But $Iwt([u,v_0]) \leq (1+\epsilon)\delta W_i$ and $Iwt([v,v_1]) \leq (1+\epsilon)\delta W_i$, while $Iwt([v_0,v_1]) \leq 2(1+\epsilon)\delta W_i + D_{\{G',Iwt\}}(v,u) \leq 2(1+\epsilon)\delta W_i + (1+\epsilon)L_1$. This result follows from the procedure ADDINTER-EDGESTYPE2, since $wt([v_0,v_1])$ is at most $2(1+\epsilon)\delta W_i$ plus the length of the shortest edge connecting vertices of the two clusters to which u and v belong. So $L_2 \leq (1+\epsilon)L_1 + 4(1+\epsilon)\delta W_i$, and we have that $W_i < \frac{(1+\epsilon)L_1}{1-2(1+\epsilon)\delta}$. Combining these inequalities, we get

$$L_2 \leq (1+\epsilon)L_1 + \frac{4(1+\epsilon)\delta}{1-2(1+\epsilon)\delta} \cdot L_1 < \frac{(1+\epsilon)(1+2\delta)}{1-2(1+\epsilon)\delta} \cdot L_1.$$

Case 2. [$m \geq 2$ and m is even.] Suppose $[v_i, v_{i+1}]$ and $[v_{i+1}, v_{i+2}]$ are any two consecutive inter-cluster edges on Q. Observe that the sum of their weights is greater than W_i. If this were not so, then the edge $[v_i, v_{i+2}]$ would instead have been added to Q while Q was being constructed. Divide Q into portions $Q_0, Q_1, \ldots$, where Q_{2i} is the portion between v_{2i} and v_{2i+2}. Similarly, divide P into portions $P_0, P_1, \ldots$, where P_{2i} is the portion between the last vertex intersecting C_{2i} and the first vertex intersecting C_{2i+2}. We shall first prove that for any even i, the weight of Q_{2i} is no more than a constant times the weight of P_{2i}.

Let the weight of P_{2i} be p_{2i} and that of Q_{2i} be q_{2i}. Since there cannot be an inter-cluster edge between v_{2i} and v_{2i+2}, we have that $p_{2i} > W_i - 2\delta(1+\epsilon)W_i$. Select r to be any vertex of P_{2i} within the intermediate cluster C_{2i+1}. The vertex r splits P_{2i} into two portions. Let p^1_{2i} (respectively p^2_{2i}) be the initial (respectively final) portions; thus $p_{2i} = p^1_{2i} + p^2_{2i}$. From the procedure AddInterEdgesType2 we have $Iwt([v_{2i}, v_{2i+1}]) \leq (1+\epsilon)p^1_{2i} + 2\delta(1+\epsilon)W_i$, and similarly $Iwt([v_{2i+1}, v_{2i+2}]) \leq (1+\epsilon)p^2_{2i} + 2\delta(1+\epsilon)W_i$. Adding the two we get $q_{2i} \leq (1+\epsilon)p_{2i} + 4\delta(1+\epsilon)W_i$. We now have two inequalities relating p_{2i}, q_{2i} and W_i. We have that $W_i < \frac{p_{2i}}{1-2(1+\epsilon)\delta}$, which gives us

$$q_{2i} < (1+\epsilon)p_{2i} + 4\delta(1+\epsilon) \cdot \frac{p_{2i}}{1-2\delta(1+\epsilon)} < p_{2i} \cdot \frac{(1+\epsilon)(1+2\delta)}{1-2\delta(1+\epsilon)}.$$

Summing over all even values of i, and taking into account the two intra-cluster edges at either ends of Q, we get

$$L_2 < L_1 \cdot \frac{(1+\epsilon)(1+2\delta)}{1-2\delta(1+\epsilon)} + 2\delta(1+\epsilon)W_i.$$

Since u is sufficiently far from v, we know that $L_1 > W_i - 2\delta(1+\epsilon)W_i$. That is, $\frac{L_1}{1-2\delta(1+\epsilon)} > W_i$. Substituting this in the above inequality we obtain

$$L_2 < L_1 \cdot \frac{(1+\epsilon)(1+4\delta)}{1-2\delta(1+\epsilon)}.$$

Case 3. [$m \geq 3$ and m is odd.] The analysis will be exactly the same as in the previous case, except that we have to account for the last inter-cluster edge along Q and correspondingly the portion of P between the last two clusters. Let q_{m-1} be the integer weight of $[v_{m-1}, v_m]$, and let p_{m-1} be the weight of the portion of P between the last vertex intersecting C_{m-1} and the first vertex intersecting C_m. Clearly $q_{m-1} \leq (1+\epsilon)p_{m-1} + 2\delta(1+\epsilon)W_i$. This inequality does not change if we rewrite it as $q_{m-1} < (\frac{(1+\epsilon)(1+4\delta)}{1-2\delta(1+\epsilon)})p_{m-1} + 2\delta(1+\epsilon)W_i$. We then sum up as above, and get

$$L_2 < L_1 \cdot \frac{(1+\epsilon)(1+2\delta)}{1-2\delta(1+\epsilon)} + 4\delta(1+\epsilon)W_i.$$

Since $L_1 > W_i - 2\delta(1+\epsilon)W_i$, we have that $L_1 \cdot \frac{4\delta(1+\epsilon)}{1-2\delta(1+\epsilon)} > 4\delta(1+\epsilon)W_i$. Substituting this in the above inequality we obtain

$$L_2 < L_1 \cdot \frac{(1+\epsilon)(1+6\delta)}{1-2\delta(1+\epsilon)}.$$

□

Before processing any group E_i, the algorithm constructs a fresh cluster graph H. It uses a radius of δW_i, where $(W_i, 2W_i]$ is the interval which bounds the edge weights of E_i.

Lemma 8.14 *During the processing of any group E_i, the graph H always represents a valid cluster graph of G'.*

Proof: Let the edges in E_i be ordered by increasing weight as $e_{i1}, \ldots, e_{il}$. The proof is by induction. In the base case, when none of the edges have been processed, the lemma is obviously true. Now assume that the lemma is true just before the algorithm decides to examine edge $e_{ij}=(u,v)$. If this edge is not added to G', then the lemma still holds. Now, suppose this edge is added to G'. Since $Iwt(u,v) \geq wt(u,v) > W_i$ and $\delta<1/2$, the distance between any two previous cluster centers in the new G' will remain greater than δW_i, and thus the previous cluster cover will remain valid. Also the previous intra-cluster edges, and the inter-cluster edges of type 1 (see the definition of inter-cluster edges) will remain the same. We only have to make sure that we add new inter-cluster edges of type 2, and it is easily seen, that this is done by the algorithm. It remains to decide what weights are to be assigned to these new inter-cluster edges in H. Consider one such edge $[x,y]$, where x (respectively y) is the center of the cluster to which u (respectively v) belongs. The weight of this edge should be assigned $D_{\{G',wt\}}(x,y)$ (the shortest path in the new graph G' between x and y). However, it will be too time consuming to compute this directly. Instead the algorithm assigns the weight as $Iwt([x,u]) + Iwt(u,v) + Iwt([y,v])$, see Section 8.2.2. We now show that our choice of δ makes this acceptable.

Assume the contrary, that a shorter alternate path P exists between x and y. Since P cannot involve the edge (u,v), it only contains edges of the previous G'. But we know that (u,v) was selected to be added to G', thus no cluster path existed between u and v of weight within $\sqrt{tt'} \cdot Iwt(u,v)$ in H. Furthermore, since u is sufficiently far from v, we may use Lemma 8.13 to get

$$wt(P) + 2\delta W_i(1+\epsilon) > \frac{1-2\delta(1+\epsilon)}{(1+\epsilon)(1+6\delta)} \cdot \sqrt{tt'} \cdot Iwt(u,v).$$

We have that $wt(P) < Iwt([x,u]) + Iwt(u,v) + Iwt([v,y])$ which is at most

$Iwt(u,v) + 2\delta W_i(1+\epsilon)$. Putting these two results together we obtain

$$4\delta(1+\epsilon)W_i + Iwt(u,v) > \frac{1-2\delta(1+\epsilon)}{(1+\epsilon)(1+6\delta)} \cdot \sqrt{tt'} \cdot Iwt(u,v).$$

Using the fact that $Iwt(u,v) > W_i$ we get:

$$4\delta(1+\epsilon) > \frac{1-2\delta(1+\epsilon)}{(1+\epsilon)(1+6\delta)} \cdot \sqrt{tt'} - 1.$$

If we solve this inequality for δ, we see that the only positive solutions are

$$\delta > \frac{\sqrt{tt'} - (1+\epsilon)}{2(\sqrt{tt'}(1+\epsilon) + 5 + 7\epsilon + 2\epsilon^2)}.$$

According to algorithm IMPROVE-GREEDY the above inequality will never be satisfied, see Fig. 8.2. Hence, we have a contradiction. □

The following theorem now concludes this section.

Theorem 8.15 *The graph produced by* IMPROVED-GREEDY *is a t-spanner of the complete Euclidean graph.*

Proof: Follows from the fact that G' is a $\sqrt{tt'}$ spanner of G and that G is a $\sqrt{t/t'}$ spanner of the complete graph. □

8.4 The weight of G' is $O(wt(MST))$

In [22] it was shown that the greedy algorithm produces a spanner that has $O(n)$ edges and a total weight of $O(\log n) \cdot wt(MST(V))$, where $MST(V)$ is the minimum spanning tree of V. The analysis of the greedy algorithm was then improved in [30]. The proof relies on a property known as the *leapfrog property*. This property restricts how a set of line segments may be positioned in space. Here we provide a definition, which is technical and non-intuitive.

Let $t \geq t' > 1$. A set of line segments, denoted E', in d-dimensional space satisfy the (t', t)-*leapfrog* property if the following is true for every possible subset $S = \{(u_1, v_1), \ldots, (u_m, v_m)\}$ of E':

$$t' \cdot wt(u_1, v_1) < \sum_{i=2}^{m} wt(u_i, v_i) + t \cdot (\sum_{i=1}^{m-1} wt(v_i, u_{i+1}) + wt(v_m, u_1)).$$

Informally, this definition says that if there exists an edge between u_1 and v_1 then any path, not including (u_1, v_1) must have length greater than $t' \cdot wt(u_1, v_1)$. The following fact was shown by Das and Narasimhan [31].

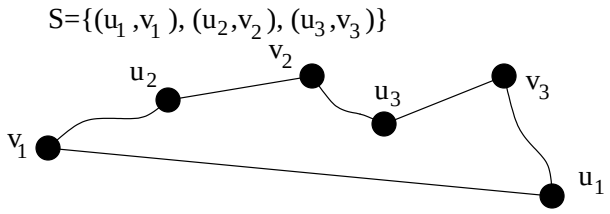

Figure 8.7: Illustration of the definition of the leapfrog property.

Fact 8.16 *(Theorem 3 in [31])*
There exists a constant $0 < \phi < 1$ such that the following holds: if a set of line segments E' in d-dimensional space satisfies the (t', t)-leapfrog property where $t \geq t' \geq \phi t + 1 - \phi > 1$, then $wt(E') = O(wt(MST))$, where MST is a minimum spanning tree connecting the endpoints of E'. The constant implicit in the O-notation depends on t and d.

Now, suppose that we construct a t-spanner such that for every spanner edge (u, v), the second shortest path is not necessary longer than $t \cdot wt(u, v)$, but longer than $t' \cdot wt(u, v)$, for some t' such that $t \geq t' > 1$. In this case the t-spanner satisfies the (t, t')-leapfrog property, as can be proved by using arguments similar to those used in Lemma 2.4 in [30]. Hence, the produced spanner will then have total weight $O(wt(MST(V)))$.

So, it remains to prove that the weight of the second shortest path between u and v is greater than $t' \cdot wt(u, v)$. First note that the edges in E_0 do not contribute much, because their total length is at most equal to the length of the longest edge ($< n \cdot D/n$), which is less than the weight of the minimum spanning tree. We estimate $wt(E' \backslash E_0)$, where E' is the set of edges produced by the algorithm.

Lemma 8.17 *Let $e=(u, v) \in E' \backslash E_0$. The weight of the second shortest path between u and v is greater than $t' \cdot wt(u, v)$.*

Proof: Let C be the shortest simple cycle in G' containing e. We have to estimate $wt(C) - wt(u, v)$. Let $e_1 = (u_1, v_1)$ be the longest edge on the cycle. Then $e_1 \in E' \backslash E_0$, and among the cycle edges it is examined last by the algorithm. What happens while the algorithm is examining e_1?

Assume that e_1 is examined in phase i. There is an alternate path in G' from u_1 to v_1 of weight $wt(C) - wt(u_1, v_1)$. But since the algorithm eventually decides to add e_1 to the spanner, at that moment the weight of each cluster path from u_1 to v_1 is larger than $\sqrt{tt'} \cdot Iwt(u_1, v_1) \cdot U_i$. Notice that $Iwt(u_1, v_1) \cdot U_i$

and $wt(u_1, v_1)$ is larger than W. This implies that u_1 and v_1 are not contained in the same cluster. Thus u_1 is sufficiently far from v_1. Lemma 8.13 implies that the weight of each path in G' between u_1 and v_1 is large, i.e.,

$$wt(C) - wt(u_1, v_1) > \sqrt{tt'} \cdot Iwt(u_1, v_1) \cdot U_i > \sqrt{tt'} \cdot wt(u_1, v_1) \cdot \frac{1 - 2\delta(1+\epsilon)}{(1+\epsilon)(1+6\delta)}.$$

But we know, according to the algorithm, that $\delta \leq \frac{\sqrt{tt'}-(1+\epsilon)t'}{2(1+\epsilon)(\sqrt{tt'}+3t')}$. Substituting, we obtain $wt(C) - wt(u_1, v_1) > t' \cdot wt(u_1, v_1)$. □

Finally, since Lemma 8.17 holds, we can use the following observation which together with Fact 8.16 concludes the proof of Theorem 8.1.

Observation 8.18 *E'/E_0 satisfies the (t', t)-leapfrog property.*

Proof: Consider any subset of the edges, $S = \{(u_1, v_1), \ldots, (u_m, v_m)\}$ of E'. By Lemma 8.17 we know that $t' \cdot d(u_1, v_1)$ is smaller than the weight of the second shortest path between u_1 and v_1 in G'. Consider a path P from v_1 to u_1, composed of the shortest path from v_1 to u_2 (of weight $\leq t \cdot d(v_1, u_2)$), the edge (u_2, v_2), the shortest path from v_2 to u_3 (of weight $\leq t \cdot d(v_2, u_3)$), and so on, until the final portion is the shortest path from v_m to u_1. Clearly $wt(P)$ is at least as large as the weight of the second shortest path between u_1 and v_1. But $wt(P)$ is also equal to the right hand side of the definition of the leapfrog property. The fact follows. □

This concludes the proof of Theorem 8.1.

8.5 Open problem

The main open problem that remains unsolved is if it's possible to construct a sparse t-spanner in time $O(n \log n)$ in the algebraic decision tree model of computation.

8.6 Acknowledgments

We are grateful to Professor Michiel Smid for helpful discussions and for pointing out errors in an earlier draft.

Chapter 9

Higher order Delaunay triangulations

One of the most well-known and useful structures studied in computational geometry is the Delaunay triangulation [32, 38, 75]. It has applications in spatial interpolation between points with measurements, because it defines a piecewise linear interpolation function. The Delaunay triangulation also has applications in mesh generation for finite element methods. In both cases, the usefulness of the Delaunay triangulation as opposed to other triangulations is the fact that the triangles are well-shaped. It is well-known that the Delaunay triangulation of a set P of points maximizes the smallest angle, over all triangulations of P.

One specific use of the Delaunay triangulation for interpolation is to model elevation in Geographic Information Systems. The so-called *Triangulated Irregular Network,* or *TIN,* is one of the most common ways to model elevation. Elevation is used for hydrological and geomorphological studies, for site planning, for visibility impact studies, for natural hazard modeling, and more.

Because a TIN is a piecewise linear, continuous function which is generally not differentiable at the edges, these edges play a special role. In elevation modeling, one usually tries to make the edges of the TIN coincide with the ridges and valleys of the terrain. Then the rivers that can be predicted from the elevation model are a subset of the edges of the TIN. When one obtains a TIN using the Delaunay triangulation of a set of points, the ridges and valleys in the actual terrain will not always be as they appear in the TIN. The so-called 'artificial dam' in valleys is a well-known artifact in elevation models, Fig. 9.1. It appears when a Delaunay edge crosses a valley from the one hillside to the other hillside, creating a local minimum in the terrain model slightly higher up in the valley. It is known that in real terrains such local minima are quite rare [48]. These artifacts

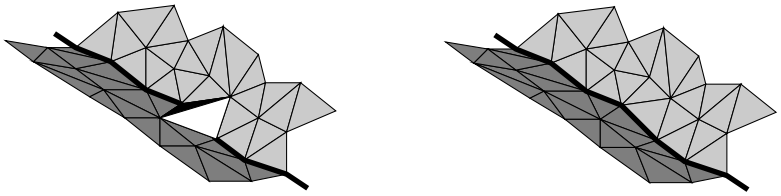

Figure 9.1: Artificial dam that interrupts a valley line (left), and a correct version obtained after one flip (right).

need to be repaired, if the TIN is to be used for realistic terrain modeling [86], in particular for hydrological purposes [67, 68, 91]. If the valley and ridge lines are known, these can be incorporated by using the constrained Delaunay triangulation [26, 36, 66]. The cause of problems like the one mentioned above may be that the Delaunay triangulation is a structure defined for a planar set of points, and does not take into account the third dimension at all. One would like to define a triangulation that is both well-shaped and has some other properties as well, like avoiding artificial dams. This lead us to define *higher-order Delaunay (HOD) triangulations,* a class of triangulations for any point set P that allows some flexibility in which triangles are actually used. The Delaunay triangulation of P has the property that for each triangle, the circle through its vertices has no points of P inside. A k-order Delaunay (k-OD) triangulation has the relaxed property that at most k points are inside the circle. The idea is then to develop algorithms that compute some HOD triangulation that optimizes some other criterion as well. Such criteria could, for example be, minimizing the number of local minima, and minimizing the number of local extrema. The former criterion deals with the artificial dam problem, and the latter criterion also deals with interrupted ridge lines. For finite element method applications, criteria like minimizing the maximum angle, area triangle, and degree of any vertex may be of use [17, 18, 19].

In Section 9.1 we define HOD triangulations and show some basic properties. In Section 9.2 we give an algorithm to compute which edges can be included in a k-OD triangulation. The algorithm runs in $O(nk \log n + n \log^3 n)$ expected time. In Section 9.3 we consider 1-OD triangulations, and prove more specific, useful results in this case. In Section 9.4 we give the applications. We show that for 1-OD triangulations, most of the criteria we study can be optimized in $O(n \log n)$ time. We also give a general heuristic for the case of k-OD triangulations. Directions for further research are given in Section 9.5.

9.1 Definitions and preliminaries

We first define higher-order Delaunay edges, higher-order Delaunay triangles, and higher-order Delaunay triangulations. Given two vertices u and v we will denote by $\overline{uv}$ the edge between u and v and by $\vec{uv}$ the directed line segment from u to v. Furthermore, the unique circle through three vertices u, v and w is denoted $C(u, v, w)$, and the triangle defined by u, v and w is denoted $\triangle uvw$. We will throughout this article assume that P is non-degenerate, that is, no three points of P lie on a line and no four points of P are co-circular.

Definition 9.1 *Let P be a set of points in the plane. For $u, v, w \in P$:*

- *An edge $\overline{uv}$ is a* k-order Delaunay edge *(or* k-OD edge*) if there exists a circle through u and v that has at most k points of P inside, Fig 9.2;*
- *A triangle $\triangle uvw$ is a* k-order Delaunay triangle *(or* k-OD triangle*) if the circle through u, v, and w has at most k points of P inside.*
- *A triangulation of P is a* k-order Delaunay triangulation *(or* k-OD triangulation*) of P if every triangle of the triangulation is a k-OD triangle.*

For $k = 0$, the definitions above match the usual Delaunay edge and triangle definitions.

Lemma 9.2 *Let P be a set of points in the plane.*
(a) *Every edge of a k-OD triangle is a k-OD edge.*
(b) *Every edge of a k-OD triangulation is a k-OD edge.*
(c) *Every k-OD edge with $k > 0$ that is not a 0-OD edge intersects a Delaunay edge.*

Proof: For the first part, consider the circle through the three vertices of the k-OD triangle. This circle contains at most k other points of P. Since no four points are co-circular, we can slightly change the circle by letting one of the vertices loose and leaving it to the outside, while acquiring no other point of P. This shows that the edge connecting the two other vertices of the triangle is a k-OD edge. The second part of the lemma follows immediately from the first part, and the third part is trivial. □

Note that the converse of Lemma 9.2(a) is not true. Not every triangle consisting of three k-OD edges is a k-OD triangle. Figure 9.3a shows an example where three 1-OD edges form a 3-OD triangle. A natural question to ask is whether any k-OD edge or any k-OD triangle can be part of some k-OD triangulation. Put differently, can k-OD edges exist that cannot be used in any

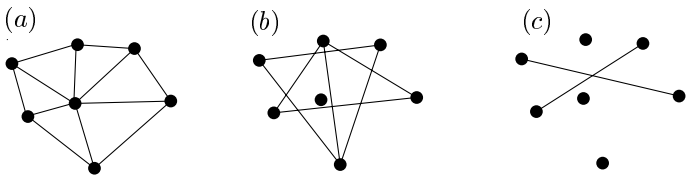

Figure 9.2: (a) The 0-OD edges (b) extended with the new 1-OD edges, and (c) the new 2-OD edges.

k-OD triangulation? Indeed, such edges (and triangles) exist. In the next section we'll give a method to test for any k-OD edge if it can be extended to a k-OD triangulation. Figure 9.3b shows an example where $\triangle uvx$ is a 1-OD triangle that cannot be included in a 1-OD triangulation since $\triangle uvy$ is not a 1-OD triangle. To distinguish between "useful" and "non-useful" k-OD edges we use the following definition.

Definition 9.3 *Let P be a set of points in the plane. A k-OD edge $\overline{uv}$ with $u, v \in P$ is* useful *if there exists a k-OD triangulation that includes $\overline{uv}$. A k-OD triangle $\triangle uvw$ with $u, v, w \in P$ is* valid *if it does not contain any point of P inside and its three edges are useful.*

There is a close connection between k-OD edges and higher-order Voronoi diagrams.

Lemma 9.4 *Let P be a set of n points in the plane, let $k \leq n/2 - 2$, and let $u, v \in P$. The edge $\overline{uv}$ is a k-OD edge if and only if there are two incident faces, F_1 and F_2, in the order-$(k+1)$ Voronoi diagram such that u is in the set of points that induces F_1 and v is in the set of points that induces F_2.*

Proof: For two points u and v in P, let m be the smallest integer such that the bisector of u and v appears in the order-m Voronoi diagram. Since on the one

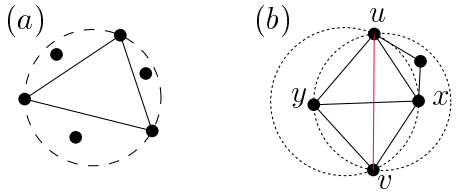

Figure 9.3: (a) Not every triangle with three 1-OD edges is a 1-OD triangle. (b) Not every 1-OD triangle ($\triangle uvx$) can be included in a 1-OD triangulation.

side of the line through u and v there are at least $n/2-1$ points of P, the bisector of u and v will appear in all higher-order Voronoi diagrams from order-m up to order-$(n/2-1)$. This bisector separates two faces representing subsets of P as stated in the lemma. □

Since the worst case complexity of order-$(k+1)$ Voronoi diagrams is $O((k+1)(n-k-1))$ [57], it follows that $O(n+nk)$ pairs of points can give rise to a k-OD edge. These pairs can be computed in $O(nk\log n + n\log^3 n)$ expected time [1].

9.2 Higher order Delaunay triangulations

In this section we show some properties of k-OD edges. Next we give an efficient way to compute all useful k-OD edges of a point set P. In Section 9.4.2 we give a heuristic to compute k-OD triangulations that take into account some other criterion.

9.2.1 Properties of k-OD edges

There are always Delaunay edges that must be present in a k-OD triangulation, for example the convex hull of the point set. For the 1-OD case we will show later that the Delaunay edges that must be in a 1-OD triangulation partition the interior of the convex hull into triangles and quadrilaterals. The *k-OD triangulation skeleton* is defined as the set of Delaunay edges that does not intersect any useful k-OD edges. A first step of completing the k-OD skeleton would be to decide which k-OD edges are possible to insert into a k-OD triangulation. The main result in this section is Lemma 9.10, which allows us to perform a simple and fast test to check if a k-OD edge is useful or not. The result also implies heuristics for computing a k-OD triangulation; this given in Section 9.4.2. We next study the properties of an arbitrary k-OD edge $\overline{uv}$. The following observation is a reformulation of Lemma 9.4 in the Dutch textbook [32].

Observation 9.5 *For any k-OD edge $\overline{uv}$ and any Delaunay edge $\overline{sp}$ that intersects $\overline{uv}$, the circle $C(u,v,s)$ contains p.*

The following corollary is a direct consequence of Observation 9.5.

Corollary 9.6 *Consider a k-OD edge $\overline{uv}$. Any circle through u and v that does not contain any vertices to the left (right) of $\vec{vu}$ contains all vertices to the right (left) of $\vec{vu}$ that are incident to Delaunay edges that intersect $\overline{uv}$.*

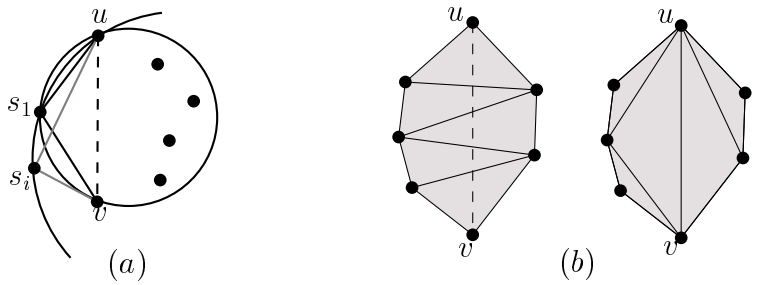

Figure 9.4: (a) If $\overline{uv}$ is useful then $\triangle us_1v$ is a useful k-OD triangle. (b) The hull of k-OD edge $\overline{uv}$ (shaded) and the completion of $\overline{uv}$.

9.2.2 Testing a k-OD edge for usefulness

To decide if a k-OD edge, $\overline{uv}$ can be included in any k-OD triangulation one has to check if the point set can be k-OD triangulated with the edge $\overline{uv}$. A naive strategy to decide if the edge is useful or not would be to search through all possible k-OD triangulations to see if the edge is included in any of these. This is obviously both space and time consuming, since there is an exponential number of possible k-OD triangulations. We will show that it suffices to check only two circles through the points u, v, and one other point, and count how many points they contain. For simplicity, we will assume, without loss of generality, that $\overline{uv}$ is vertical and that u is above v.

Lemma 9.7 *Let $\overline{uv}$ be a k-OD edge and let s_1 be the point to the left (right) of $\vec{vu}$, such that the circle $C(u, s_1, v)$ contains no points to the left (right) of $\vec{vu}$. If $\triangle us_1v$ is not a k-OD triangle then $\overline{uv}$ is not useful.*

Proof: Let us assume that $\triangle us_1v$ is not a k-OD triangle. It follows that the circle $C(u, s_1, v)$ contains more than k points to the right of $\vec{vu}$. Suppose that still a k-OD triangulation T exists that includes $\overline{uv}$. Let $\triangle us_iv$ be the triangle in T to the left of $\vec{vu}$. Then point s_i must lie either to the left of $\vec{vs}_1$ or to the right of $\vec{us}_1$, otherwise s_1 would lie inside $\triangle us_iv$ and this contradicts that $\triangle us_iv$ is a face in T. By symmetry we may assume that s_i lies to the left of $\vec{vs}_1$, see Figure 9.4a. Let p_1 and p_2 be two points in P such that $\triangle s_1p_1p_2$ is in T and it intersects the triangle $\triangle uvs_1$. Possibly, $p_1 = u$, or $p_2 = s_i$, or both. The circle $C(s_1, p_1, p_2)$ includes the part of the circle $C(u, v, s_1)$ to the right of $\vec{vu}$ since p_1 and p_2 lie outside $C(u, v, s_1)$ (one of them may lie on the circle). Hence, $C(s_1, p_1, p_2)$ contains at least $k + 2$ vertices: $k + 1$ points that are also inside $C(u, s_1, v)$, and furthermore the point v. This implies that $\triangle us_iv$ cannot be a

k-OD triangle either, contradicting the assumption that a k-OD triangulation exists. □

We would like to go a step further than the result of Lemma 9.7, namely, prove that if the "first" triangle on the left side of $\vec{uv}$, $\triangle us_1v$, is valid, and the symmetrically defined triangle on the right side of $\vec{vu}$ is valid, then $\overline{uv}$ is useful. We show this result constructively, by giving a method that gives a k-OD triangulation that includes $\overline{uv}$. It appears that we only have to compute a triangulation of a small region near $\overline{uv}$, called the hull of $\overline{uv}$. The hull is defined as follows.

Definition 9.8 *The* hull *of a k-OD edge $\overline{uv}$ is the closure of the union of all Delaunay triangles whose interior intersects $\overline{uv}$, Fig. 9.4b. We say that the hull of a k-OD edge $\overline{uv}$ is* complete *if it is triangulated. The additional edges needed to triangulate the hull are called the* completion *of $\overline{uv}$, Fig. 9.4b. Furthermore, $\overline{uv}$ is said to be* isolated *if the hull of $\overline{uv}$ is in the triangulation.*

The following algorithm computes a triangulation of the hull. Let $\overline{uv}$ be a k-OD edge. Let p_1 be the point to the right of $\vec{vu}$ such that the part of $C(u, v, p_1)$ to the right of $\vec{vu}$ is empty. Note that p_1 must be a vertex on the boundary of the hull of $\overline{uv}$. Add the two edges $\overline{up_1}$ and $\overline{vp_1}$ to the graph. Continue like this recursively for the two edges $\overline{up_1}$ and $\overline{vp_1}$ until the hull of $\overline{uv}$ to the right of $\vec{vu}$ is completely triangulated. The same procedure is then performed on the left side of $\vec{vu}$. The obtained triangulation is called the *greedy triangulation* of the hull of $\overline{uv}$, see Figure 9.5. The next corollary, which is a direct consequence of Corollary 9.6, shows that the hull is a simple polygon consisting of at most $2k + 2$ vertices.

Corollary 9.9 *The Delaunay edges intersecting one useful k-OD edge $\overline{uv}$ are connected to at most k vertices on each side of the k-OD edge.*

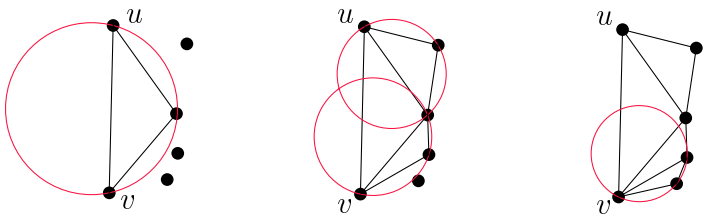

Figure 9.5: Greedy triangulation to the right of $\vec{uv}$.

Lemma 9.10 *Let $\overline{uv}$ be a k-OD edge, let s be the point to the left of $\vec{vu}$, such that the circle $C(u,s,v)$ contains no points to the left of $\vec{vu}$. Let s' be defined similarly but to the right of $\vec{vu}$. Edge $\overline{uv}$ is a useful k-OD edge if and only if $\triangle uvs$ and $\triangle uvs'$ are valid.*

Proof: The "only if" part is given in Lemma 9.7. For the "if" part, consider the greedy triangulation of $\overline{uv}$. It is given that $\triangle uvs$ and $\triangle uvs'$ are valid. Since the recursive continuation for the edges $\overline{vs}$, $\overline{vs'}$, $\overline{us}$, and $\overline{us'}$ yields circles to be tested that can only contain fewer points than $C(u,s,v)$ or $C(u,s',v)$, the corresponding triangles must be valid too. □

After preprocessing, we can test efficiently whether an edge $\overline{uv}$ is a useful k-OD edge. First, locate u and v in the Delaunay triangulation, and traverse it from u to v along $\overline{uv}$ from Delaunay triangle to Delaunay triangle. Collect all intersected Delaunay edges. Determine the endpoints of intersected Delaunay edges left of $\vec{vu}$ and right of $\vec{vu}$. Next, we determine s and s' as in the lemma just given. Now we must test how many points lie in the circles $C(u,s,v)$ and $C(u,s',v)$ to determine usefulness. To this end, we use a data structure for storing a set of points that can report the $(k+1)$st closest point for a given query point, i.e., the center of the query circle. If the distance between the center and the $(k+1)$st closest point is at most the radius of the query circle then there are at most k points within the circle. Note that the $(k+1)$-order Voronoi diagram returns the $(k+1)$st closest point. The $(k+1)$-order Voronoi diagram can be preprocessed in time $O(kn \log kn)$ such that queries can be answered in time $O(\log kn)$. Thus, it takes $O(kn \log n)$ time in total to check all k-OD edges for usefulness, since there are $O(kn+n)$ edges to be tested.

In Section 9.1 we showed that all k-OD edges can be determined in expected time $O(kn \log n + n \log^3 n)$ using an algorithm for higher-order Voronoi diagrams [1]. Thus, it takes $O(kn \log n + n \log^3 n)$ expected time overall to determine all useful k-OD edges.

9.3 One-higher order Delaunay triangulations

We examine the special structure of any 1-OD edge $\overline{uv}$ further. We already observed in Corollary 9.9 that if $\overline{uv}$ is a useful 1-OD edge, and not a 0-OD edge, then it intersects exactly one Delaunay edge. We again assume without loss of generality that $\overline{uv}$ is vertical and that u is above v.

Lemma 9.11 *Every useful 1-OD edge intersects at most one useful 1-OD edge.*

Proof: From Corollary 9.9 we already know that any Delaunay edge at most intersect one 1-OD edge. It remains to prove the lemma for any non-Delaunay,

useful 1-OD edge $\overline{uv}$. Let $\overline{sy}$ be the Delaunay edge that $\overline{uv}$ intersects. If there exists a useful 1-OD edge that intersects the Delaunay edge $\overline{su}$ it must be connected to y, according to Corollary 9.9; see Figure 9.6. Denote this edge $\overline{xy}$. Since $\overline{xy}$ is a 1-OD edge x must be connected to u and s by Delaunay edges. Consider the circle $C(u, v, y)$. This circle contains only s by Observation 9.5 and from the fact that $\overline{uv}$ is useful. The circle $C(u, y, x)$ can be obtained by expanding $C(u, v, y)$ until it hits x while releasing v. Since we let go of v, both v and s is contained in $C(u, y, x)$. Thus $\triangle xuy$ cannot be a 1-OD triangle and hence $\overline{xy}$ cannot be a useful 1-OD edge, which contradicts our assumption. By symmetry this holds for any edge intersecting $\overline{uy}$, $\overline{sv}$, or $\overline{vy}$. □

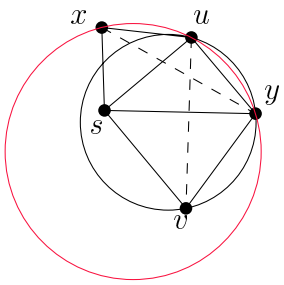

Figure 9.6: There are no non-Delaunay, useful 1-OD edges that intersect.

The lemma just given shows that if $\overline{uv}$ is a useful 1-OD edge that is not Delaunay, then the four Delaunay edges $\overline{us}$, $\overline{uy}$, $\overline{sv}$, and $\overline{vy}$ must be in every 1-OD triangulation. Given a triangulation $\mathcal{T}$ and two edges e_1 and e_2 in $\mathcal{T}$, we say that e_1 and e_2 are independent if they are not incident to the same triangle in $\mathcal{T}$. From Corollary 9.9 and Lemma 9.11 we obtain the main result of this section.

Corollary 9.12 *Every 1-OD triangulation can be obtained from a Delaunay triangulation by flips of independent Delaunay edges.*

It is easy to see that—given the Delaunay triangulation—all 1-OD edges can be determined in linear time. In $O(n \log n)$ time, we can find out which ones are useful.

9.4 Triangulations with additional criteria

Recall from the introduction that Delaunay triangulations are often used in terrain modeling, because they give well-shaped triangles. However, artifacts

like artificial dams may arise. Since the Delaunay triangulation is completely specified by the input points (in non-degenerate cases), there is no flexibility to incorporate other criteria into the triangulation, which is why higher-order Delaunay triangulations were introduced. In this section we show how to avoid artificial dams in higher-order Delaunay triangulations, and deal with a number of other criteria as well. Many of these criteria can be optimized for 1-OD triangulations, which is what we will show first. Then we give a general heuristic to incorporate such criteria in k-OD triangulations. Unfortunately, we currently cannot optimize nor approximate these criteria yet.

9.4.1 Applications for 1-OD triangulations

Minimizing the number of local minima

To minimize the number of local minima is straight-forward if the domain is the class of 1-OD triangulations.

Lemma 9.13 *Removing a local minimum by adding 1-OD triangles never prevents any other local minimum from being removed.*

Proof: Consider a convex quadrilateral with vertices A, B, C, D with heights a, b, c, d in clockwise order with $a, c < b, d$, and assume that the Delaunay triangulation connects B and D. If a local minimum in A or C can be removed by the flip which connects A and C (if it gives two 1-OD triangles) then this flip is always good. The only two vertices that lose a neighbor are B and D—they lose each other as neighbor— but neither will become a local minimum because they remain connected to the vertices A and C, which are lower. □

Theorem 9.14 *An optimal 1-OD triangulation with respect to minimizing the number of local minima can be obtained by flips of independent Delaunay edges in $O(n \log n)$ time.*

Minimizing the number of local extrema

The number of local extrema—minima and maxima—can also be efficiently minimized over all 1-OD triangulations. In the previous subsection we could choose the edge in any quadrilateral that connects to the lowest of the four points. But if we want to minimize local minima and maxima we get conflicts: it can be that the one edge of a convex quadrilateral gives an additional local minimum and the other edge gives a local maximum. Consider the subdivision S consisting of all edges that must be in any 1-OD triangulation, so S contains triangular and convex quadrilateral faces only. Consider the set of points that either have no

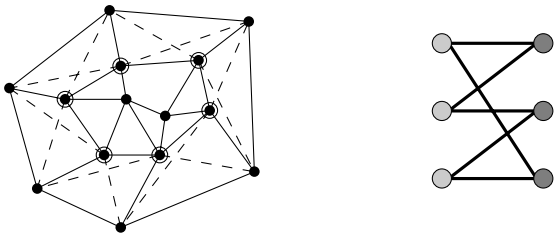

Figure 9.7: Local extrema appearing in an even cycle.

lower neighbors or no higher neighbors; they are extremal in S. Consider the graph $G = (M, A)$, where M is the set of nodes representing the local extrema, and two nodes m, m' are connected if they represent points on the same quadrilateral face and the one triangulating edge makes that m is not a local extremum and the other triangulating edge makes that m' is not a local extremum.

Lemma 9.15 *Any quadrilateral face of S defines at most one arc in G, and this arc connects a local minimum to a local maximum.*

Proof: Any triangulating edge of a quadrilateral face can only avoid a local extremum if the highest two points are opposite and the lowest two points are opposite in the quadrilateral. One triangulating edge can only make the second-highest point non-maximal, and the other triangulating edge can only make the second-lowest point non-minimal. □

From the lemma it follows that G is bipartite, because every arc connects a local minimum to a local maximum. G may contain many isolated nodes; these extreme points can be removed without affecting the other extrema. For any node incident to only one arc, we can choose to make the point representing that node to be non-extremal, without giving up optimality (minimum number of local extrema). If there are no nodes connected to only one other node, all nodes appear in cycles. Since the graph is bipartite, every cycle has even length. Take any cycle (of even length). Now all nodes in the cycle can be made non-extremal: we assign one quadrilateral (represented by the arc) to one incident extremum of S and choose the triangulating edge to make it non-extremal. We can repeat to treat nodes with only one incident arc, and even cycles, until no more extrema can be removed by triangulating edges. Then we complete the triangulation of S in any manner. This greedy, incremental method completes the subdivision S to a 1-OD triangulation that minimizes the number of local minima and maxima. The algorithm can be implemented to run in linear time when S is given.

Theorem 9.16 *An optimal* 1*-OD triangulation with respect to minimizing the number of local extrema can be determined in* $O(n \log n)$ *time.*

Other criteria

In visualization applications it is sometimes important to construct planar drawings with small degree and large angles between the edges. Thus, a natural optimization criteria for a 1-OD triangulation would be to minimize the maximum degree, since the Delaunay triangulation already maximizes the minimum angle. Besides visualization applications [33], constructing drawing with high angular resolution is important in the design of optical communication networks [37]. The problem of minimizing the maximum degree have been studied in several papers [42, 49, 51]. We know of no polynomial-time algorithm that gives an optimal solution to this optimization problem. The more general problem to complete the triangulation of a biconnected planar graph while minimizing the maximum degree is known to be NP-hard. There are efficient approximation algorithms though, as the following fact shows. Let $\Delta(G)$ be the maximum degree of a graph G.

Fact 9.17 *[Kant & Bodlander [51]] There is a linear time algorithm to triangulate a biconnected planar graph* G *such that for the triangulation* G' *of* G $\Delta(G') \leq \lceil \frac{3}{2}\Delta(G) \rceil + 11$.

We cannot use the fact directly since it does not work on biconnected embedded straight-line graphs such as the 1-OD triangulation skeleton. But every quadrilateral face is convex, hence the interior of the 1-OD triangulation skeleton can always be triangulated using the above fact. The problem is the convex hull of P, which will be triangulated by edges that are not straight lines. Let h be the number of points in P on the convex hull of P. Recursively we add points until the convex hull contains three points as follows: Construct an exterior hull H with $\lceil h/2 \rceil$ points that entirely includes P. Connect each point in H with three points in the convex hull of P, keeping the resulting graph planar. Note that this recursive procedure gives a triangulation of the region outside P, where each point originally not in P has degree at most 7, and each point on the convex hull of P is connected to at most two additional edges. This gives us the following theorem.

Theorem 9.18 *There is an* $O(n \log n)$ *time algorithm to triangulate the* 1*-OD triangulation skeleton* S *such that the degree of the triangulation is at most* $\lceil \frac{3}{2}\Delta(S) \rceil + 13$.

Note that the Delaunay triangulation itself is a 2-approximation of the optimal 1-OD triangulation.

As was pointed out in the introduction such criteria as minimizing the maximum angle and minimizing the maximum area triangle may be of use for finite element method applications. These criteria (together with a number of other criteria not mentioned above) are trivial to optimize if the edges in the domain are useful 1-OD edges. This follows from the fact that these are all local optimization criteria, thanks to the nice property of 1-OD triangulations.

Theorem 9.19 *For a Delaunay triangulation of a set P of n points in the plane, an optimal 1-OD triangulation can be obtained by flips of independent Delaunay edges in $O(n \log n)$ time for each one of the following criteria: (i) minimizing the maximal area triangle, (ii) minimizing the maximal angle, (iii) maximizing the minimum radius of a circumcircle, (iv) maximizing the minimum radius of an enclosing circle, (v) minimizing the sum of inscribed circle radii, (vi) minimizing the number of obtuse angles, and (vii) minimizing the total edge length.*

9.4.2 Applications for k-OD triangulations

It appears to be difficult to obtain general optimization results for all of the criteria listed before, given a value of $k \geq 2$. When k is so large that every pair of points gives a useful edge (like $k = n - 3$), then certain criteria can be optimized. For example, when minimizing the number of local minima, we can choose an edge from every point to the global minimum (in non-degenerate cases), so that there is only one local minimum. For minimizing the maximum angle and some other criteria, various results are known [17].

To develop approximation algorithms for k-OD triangulations, we need to determine how many hulls a single hull can intersect. To this end, we first prove an upper bound on the maximum number of useful k-OD edges that intersect a given Delaunay edge. Figure 9.8 shows that $\Omega(n)$ 2-OD edges can intersect

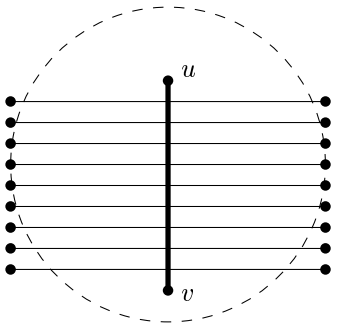

Figure 9.8: Many 2-OD edges can intersect a Delaunay edge $\overline{uv}$.

a given Delaunay edge. But these 2-OD edges cannot all be useful. The next lemma shows that the maximum number of useful k-OD edges intersecting a given Delaunay edge does not depend on n, but only on k.

Lemma 9.20 *Let $\overline{uv}$ be any Delaunay edge. The number of useful k-OD edges in a triangulation $\mathcal{T}$ that intersect $\overline{uv}$ is $O(k)$.*

Proof: For simplicity we assume that u is vertically above v. Let Q_l and Q_r be the left endpoints respectively the right endpoints of the useful k-OD edges of $\mathcal{T}$ that intersect $\overline{uv}$. Assume without loss of generality that $|Q_l| \leq |Q_r|$. Let p_r be the point in Q_r such that $C(u, v, p_r)$ includes all points in Q_r, except p_r. There exists a point $p_l \in Q_l$ such that $\overline{p_l p_r}$ intersects $\overline{uv}$.

It is easily seen that $C(u, p_l, p_r)$ together with $C(v, p_l, p_r)$ will contain all the points in $Q_r \setminus \{p_r\}$, see Fig. 9.9. Now we may apply Lemma 9.7. If $\overline{p_l p_r}$ is useful then the "first" triangles to left, denoted $\triangle p_l p_r x$, and right, denoted $\triangle p_l p_r y$, of $\vec{p_r p_l}$ must be valid, i.e., the two circles $C(p_l, p_r, x)$ and $C(p_l, p_r, y)$ includes at most k points. But, $C(p_l, p_r, x)$ and $C(p_l, p_r, y)$ will together include $C(u, p_l, p_r)$ and $C(v, p_l, p_r)$ which we know include at least $|Q_r| - 1$ points. Hence, it follows that $|Q_r| - 1 \leq 2k$, and since $\mathcal{T}$ is a planar triangulation the total number of useful k-OD edges intersecting $\overline{uv}$ will be $O(k)$. □

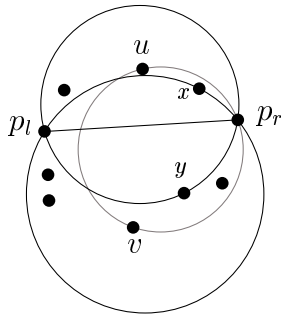

Figure 9.9: Illustration of the proof of Lemma 9.20.

Lemma 9.21 *Let $\overline{uv}$ be a useful k-OD edge and let H be its hull. The number of hulls of useful k-OD edges included in a triangulation $\mathcal{T}$ that intersect the interior of H is $O(k^2)$.*

Proof: By Corollary 9.9, the hull H of $\overline{uv}$ contains $O(k)$ Delaunay edges of P in its interior. Any useful k-OD edge intersecting H must intersect at least one of these, or itself be the Delaunay edge. By Lemma 9.20, $O(k)$ useful k-OD

edges included in a triangulation $\mathcal{T}$ can intersect a common Delaunay edge. The bound of $O(k^2)$ follows. □

To keep track of which hulls intersect we define a hull intersection graph G as follows. There is a node for the hull of every useful k-OD edge. Two nodes are connected by an arc if their hulls intersect, that is, there exists a point that is interior to both hulls. The choice of one useful k-OD edge e possibly prohibits the choice of any other useful k-OD edge whose hull intersects the hull of e. In any case, if we choose an independent set of nodes in graph G, we get a set of hulls of useful k-OD edges that can be used together in a k-OD triangulation.

Minimizing the number of local extrema (revisited) Suppose that all useful k-OD edges and their hulls have been computed, and the hull intersection graph G as well. Choose any useful k-OD edge e that removes a local minimum. Mark the node for the hull of e in G, and also the adjacent nodes. Choose the greedy triangulation of the hull of e. The choice of e prevents at most $O(k^2)$ other useful k-OD edges to be chosen for the triangulation according to Lemma 9.21. Therefore, at most $O(k^2)$ points can be prevented from not being a local minimum. Continue to choose a useful k-OD edge that avoids another local minimum, provided its node in G is unmarked, until no such choice exists.

The same approach can be used to minimize the number of extrema.

Theorem 9.22 *Let m be the smallest number of local minima (or extrema) in any k-OD triangulation of a set P of points. There is a polynomial time algorithm that computes a k-OD triangulation of P with at most $O(m \cdot k^2)$ local minima (or extrema).*

Minimizing the number of obtuse angles We consider minimizing the number of obtuse angles in k-OD triangulations in the case that k is constant. Consider the Delaunay triangulation. For any obtuse angle at a point p, it can be avoided if there is a useful k-OD edge that splits it into two non-obtuse angles. It can also be avoided if there are two useful k-OD edges that split it into three non-obtuse angles. We need not explicitly look at more than two edges, because there will always be two that already work. The completion to a k-OD triangulation may have more useful k-OD edges incident to p.

It is easy to test if a single useful k-OD edge can avoid an obtuse angle at a point p. To test whether two useful k-OD edges are needed, we can test any pair e, e' at point p. When we remove the Delaunay edges intersecting e or e', we get a region—the union of the hulls of e and e'—that must be triangulated to a k-OD triangulation. Since k is constant, we can try every possible triangulation of the region. This leads to:

Theorem 9.23 *Let k be a constant, and let m be the smallest number of obtuse angles in any k-OD triangulation of a set P of points. There is a polynomial time algorithm that computes a k-OD triangulation of P with at most $O(m \cdot k^2)$ obtuse angles.*

9.5 Directions for further research

For the case when $k \geq 2$ we just have some initial result. Obviously, optimization or approximation results are a topic of future research. Another issue we would like to address is experimental: how much benefit can one obtain for the criterion of optimization with respect to the Delaunay triangulation?

Bibliography

[1] P. K. Agarwal, M. de Berg, J. Matoušek, and O. Schwarzkopf. Constructing levels in arrangements and higher order Voronoi diagrams. *SIAM Journal of Computing*, 27:654–667, 1998.

[2] A. Aggarwal, H. Edelsbrunner, P. Raghavan, and P. Tiwari. Optimal time bounds for some proximity problems in the plane. *Information Processing Letters*, 42(1):55–60, 1992.

[3] A. Aggarwal, M. Hansen, and T. Leighton. Solving query-retrieval problems by compacting Voronoi diagrams. In *Proceedings of the 22nd Annual ACM Symposium on Theory on Computing*, pages 331–340, 1990.

[4] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, MA, 1983.

[5] M. Albertson and C. J. O'Keefe. Covering regions with squares. *SIAM Journal on Discrete Mathematics*, 2(3):240–243, 1981.

[6] I. Althöfer, G. Das, D. P. Dobkin, D. Joseph, and J. Soares. On sparse spanners of weighted graphs. *Discrete Computational Geometry*, 9:81–100, 1993.

[7] E. M. Arkin and R. Hassin. Approximation algorithms for the geometric covering salesman problem. *Discrete Applied Math*, 55:197–218, 1994.

[8] E. M. Arkin, S. Khuller, and J. S. B. Mitchell. Geometric knapsack problems. *Algoritmica*, 10:399–427, 1993.

[9] S. Arora. Nearly linear time approximation schemes for Euclidean TSP and other geometric problems. In *Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science*, pages 554–563, 1997.

[10] S. Arora, M. Grigni, D. Karger, P. Klein, and A. Woloszyn. A polynomial-time approximation scheme for weighted planar graph TSP. In *Proceedings*

of the 9th ACM-SIAM Symposium on Discrete Algorithms, pages 33–41, 1998.

[11] S. Arya, G. Das, D. M. Mount, J. S. Salowe, and M. Smid. Euclidean spanners: short, thin, and lanky. In *Proceedings of the 27th Annual ACM Symposium on Theory on Computing*, pages 489–498, 1995.

[12] L.J. Aupperle, H.E. Conn, J.M. Keil, and J. O'Rourke. Covering orthogonal polygons with squares. In *Proceedings of the 26th Annual Allerton Conference on Communication, Control and Computation*, pages 97–106, 1988.

[13] R. Bar-Yehuda and E. Ben-Hanoch. A linear-time algorithm for covering simple polygons with similar rectangles. *International Journal of Computational Geometry & Applications*, 6(1):79–102, 1996.

[14] M. Ben-Or. Lower bounds for algebraic computation trees. In *Proceedings of the 15th ACM Annual Symposium on Theory on Computing*, pages 80–86, 1983.

[15] P. Berman and B. DasGupta. On the complexities of efficient solutions of the rectilinear polygon cover problems. *Algoritmica*, 17:331–356, 1997.

[16] P. Berman and M. Karpinski. On some tighter inapproximability results. Technical Report TR98-029, ECCC, 1998.

[17] M. Bern. Triangulations. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 22, pages 413–428. CRC Press LLC, Boca Raton, FL, 1997.

[18] M. Bern, D. Dobkin, and D. Eppstein. Triangulating polygons without large angles. *International Journal of Computational Geometry & Applications*, 5:171–192, 1995.

[19] M. Bern and D. Eppstein. Mesh generation and optimal triangulation. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, volume 4 of *Lecture Notes Series on Computing*, pages 47–123. World Scientific, Singapore, 2nd edition, 1995.

[20] P. B. Callahan and S. R. Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *Journal of the ACM*, 42:67–90, 1995.

[21] S. Chaiken, D. Kleitman, M. Saks, and J. Shearer. Covering regions by rectangles. *SIAM Journal of Algebraic Discrete Methods*, 2:394–310, 1981.

[22] B. Chandra, G. Das, G. Narasimhan, and J. Soares. New sparseness results on graph spanners. *International Journal of Computational Geometry & Applications*, 5:125–144, 1995.

[23] Bernard Chazelle. Triangulating a simple polygon in linear time. *Discrete Compututational Geometry*, 6(5):485–524, 1991.

[24] B.M. Chazelle. *Computational Geometry and Convexity.* PhD thesis, Department of Computer Science, Yale University, New Haven, CT, 1979. Carnegie-Mellon University Report CS-80-150.

[25] D. Z. Chen, G. Das, and M. Smid. Lower bounds for computing geometric spanners and approximate shortest paths. In *Proceedings of the 8th Canadian Conference on Computational Geometry*, pages 155–160, 1996.

[26] L. P. Chew. Constrained Delaunay triangulations. *Algorithmica*, 4:97–108, 1989.

[27] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[28] J.C. Culberson and R.A. Reckhow. Covering polygon is hard. *Journal of Algorithms*, 17:2–44, 1994.

[29] A. Czumaj and A. Lingas. Linear-time heuristics for minimum weight rectangulation. In *Proceedings of the 28th International Colloquium on Automata, Languages, and Programming*, volume 1853 of *LNCS*, 2000.

[30] G. Das, P. Heffernan, and G. Narasimhan. Optimally sparse spanners in 3-dimensional Euclidean space. In *Proceedings of the 9th Annual ACM Symposium on Computational Geometry*, pages 53–62, 1993.

[31] G. Das and G. Narasimhan. A fast algorithm for constructing sparse Euclidean spanners. *International Journal of Computational Geometry & Applications*, 7:297–315, 1997.

[32] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications.* Springer-Verlag, Berlin, 1997.

[33] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. Algorithms for drawing graphs: an annotated bibliography. *Computational Geometry - Theory and Applications*, 4:235–282, 1994.

[34] D. Z. Du, L. Q. Pan, and M. T. Shing. Minimum edge length guillotine rectangular partition. Technical Report 02418-86, Mathematical Science Research Institute, University of California, Berkeley, CA, 1986.

[35] P. Eades and D. Rappaport. The complexity of computing minimum separating polygons. *Pattern Recognition Letters*, 14:715–718, 1993.

[36] B. Falcidieno and C. Pienovi. Natural surface approximation by constrained stochastic interpolation. *Computer Aided Design*, 22(3):167–172, 1990.

[37] M. Formann, T. Hagerup, J. Haralambides, M. Kaufmann, F. T. Leighton, A. Simvonis, E. Welzl, and G. Woeginger. Drawing graphs in the plane with high resolution. *SIAM Journal of Computing*, 22:1035–1052, 1993.

[38] S. Fortune. Voronoi diagrams and Delaunay triangulations. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 20, pages 377–388. CRC Press LLC, Boca Raton, FL, 1997.

[39] D. Franzblau and D. Kleitman. An algorithm for constructing regions with rectangles. In *Proceedings of the 16th Annual ACM Symposium on Theory on Computing*, pages 167–174, 1984.

[40] D. S. Franzblau. Performance guarantees on a sweep-line heuristic for covering rectilinear polygons with rectangles. *SIAM Journal of Discrete Mathemathics*, 2(3):307–321, 1989.

[41] M. R. Garey, R. L. Graham, and D. S. Johnson. Some NP-complete geometric problems. In *Proceedings of the 8th Annual ACM Symposium on Theory on Computing*, pages 10–21, 1976.

[42] A. Garg and R. Tamassia. Planar drawings and angular resolution: Algorithms and bounds. In *Proceedings of the 2nd Annual European Symposium on Algorithms*, volume 855 of *LNCS*, pages 12–23. Springer-Verlag, 1994.

[43] T. Gonzalez and S. Q. Zheng. Bound for partitioning rectilinear polygons. In *Proceedings of the 1st Annual ACM Symposium on Computational Geometry*, 1985.

[44] J. Gudmundsson and C. Levcopoulos. Approximation algorithms for covering polygons with squares and similar problems. In *Proceedings of the 1st International Workshop on Randomization and Approximation Techniques in Computer Science*, volume 1269 of *LNCS*, 1996.

[45] S. Hart and M. Sharir. Nonlinearity of Davenport-Schinzel sequences and of generalized path compression schemes. *Combinatorica*, 6:313–319, 1986.

[46] A. Hegedus. Algorithms for covering polygons by rectangles. *Computer Aided Design*, 14(5), 1982.

[47] J. Hershberger and S. Suri. A pedestrian approach to ray shooting: Shoot a ray, take a walk. *Journal of Algorithms*, 18:403–431, 1995.

[48] M.F. Hutchinson. Calculation of hydrologically sound digital elevation models. In *Proceedings of the 3rd International Symposium on Spatial Data Handling*, pages 117–133, 1988.

[49] K. Jansen. One strike against the min-max degree triangulation problem. *Computational Geometry - Theory and Applications*, 3:107–120, 1993.

[50] V. Kann. Personal communication, 1999.

[51] G. Kant and H. L. Bodlaender. Triangulating planar graphs while minimizing the maximum degree. *Information and Computation*, 135:1–14, 1997.

[52] J. M. Keil and C. A. Gutwin. Classes of graphs which approximate the complete Euclidean graph. *Discrete Computational Geometry*, 7:13–28, 1992.

[53] J.M. Keil. Minimally covering a horizontally convex orthogonal polygon. In *Proceedings of the 2nd Annual ACM Symposium on Computational Geometry*, pages 43–51, 1986.

[54] D.G. Kirkpatrick. Efficient computation of continuous skeletons. In *Proceedings of the 20th Annual IEEE Symposium on Foundation of Computer Science*, pages 18–27, 1979.

[55] D. E. Knuth. Big omicron and big omega and big theta. *SIGACT News*, 8(2):18–24, 1976.

[56] V. S. Anil Kumar and H. Ramesh. Covering rectilinear polygons with axis-parallel rectangles. In *Proceedings of the 31st Annual ACM Symposium on Theory on Computing*, 1999.

[57] D. T. Lee. On k-nearest neighbor Voronoi diagrams in the plane. *IEEE Transanctions on Computers*, C-31:478–487, 1982.

[58] C. Levcopoulos. A fast heuristic for covering polygons by rectangles. In *Proceedings of Fundamentals of Computation Theory*, volume 199 of *LNCS*, 1985.

[59] C. Levcopoulos. Fast heuristics for minimum length rectangular partitions of polygons. In *Proceedings of the 2nd Annual ACM Symposium on Computational Geometry*, pages 100–108, 1986.

[60] C. Levcopoulos. Improved bounds for covering general polygons with rectangles. In *Proceedings of the 7th Foundations of Software Technology and Theoretical Computer Science*, volume 287 of *LNCS*, 1987.

[61] C. Levcopoulos and A. Lingas. There are planar graphs almost as good as the complete graphs and almost as cheap as minimum spanning trees. *Algorithmica*, 8:251–256, 1992.

[62] C. Levcopoulos, G. Narasimhan, and M. Smid. Efficient algorithms for constructing fault-tolerant geometric spanners. In *Proceedings of the 30th Annual ACM Symposium on Theory Computing*, pages 186–195, 1998.

[63] C. Levcopoulos and A. Östlin. Linear-time heuristics for minimum weight rectangulation. In *5th Scandinavian Workshop on Algorithm Theory*, volume 1097 of *LNCS*, pages 271–283, 1996.

[64] A. Lingas, R. Pinter, R. Rivest, and A. Shamir. Minimum edge length partitioning of rectilinear polygons. In *Proceedings of the 20th Annual Allerton Conference on Communication, Control and Computation*, pages 53–63, 1982.

[65] W. T. Liou, J. J. M. Tan, and R. C. T. Lee. Minimum partitioning simple rectilinear polygons in $O(n \log \log n)$ time. In *Proceedings of the 5th Annual ACM Symposium on Computational Geometry*, pages 344–353, 1989.

[66] J.J. Little and P. Shi. Structural lines, TINs, and DEMs. In T.K. Poiker and N. Chrisman, editors, *Proceedings of the 8th International Symposium on Spatial Data Handling*, pages 627–636, 1998.

[67] D.R. Maidment. GIS and hydrologic modeling. In M.F. Goodchild, B.O. Parks, and L.T. Steyaert, editors, *Environmental modeling with GIS*, pages 147–167. Oxford University Press, New York, 1993.

[68] D.M. Mark. Automated detection of drainage networks from digital elevation models. *Cartographica*, 21:168–178, 1984.

[69] W.J. Masek. Some NP-complete set covering problems. Manuscript, 1979.

[70] C. S. Mata and J. S. B. Mitchell. Approximation algorithms for geometric tour and network design problems. In *Proceedings of the 11th Annual ACM Symposium on Computational Geometry*, pages 360–369, 1995.

[71] J. S. B. Mitchell. Guillotine subdivisions approximate polygonal subdivisions: A simple polynomial-time approximation scheme for geometric TSP, k-MST, and related problems. *SIAM Journal of Computing*, 28(4):1298–1309, 1998.

[72] L. Monk. *Elementary-recursive Decision Procedures*. PhD thesis, University of California, Berkley, 1975.

[73] D. Morita. Finding a minimal cover for binary images: an optimal parallel algorithm. Technical Report 88-946, Deptartment of Computer Science, Cornell University, 1988.

[74] D. Mount. Personal communication, 1998.

[75] A. Okabe, B. Boots, and K. Sugihara. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. John Wiley & Sons, Chichester, UK, 1992.

[76] E. Olofsson. En snabb approximations algoritm för handelsresandeproblemet med grannskap (*Eng. transl.* A fast approximationalgorithm for the traveling salesman problem with neighborhoods). Master thesis LUNDFD6/NFCS-5124, Department of Computer Science, Lund University, 1999.

[77] J. O'Rourke. The complexity of computing minimum convex covers for polygons. Technical Report JHU-EE 82-1, Department of Electrical Engineering and Computer Science, The Johns Hopkins University, Baltimore, MD, 1982.

[78] J. O'Rourke. *Computational Geometry in C (2nd ed.)*. Cambridge University Press, 1998.

[79] J. O'Rourke and K.J. Supowit. Some NP-hard polygon decomposition problems. *IEEE Transactions on Information Theory*, IT-29:181–190, 1983.

[80] C. H. Papadimitriou. Euclidean TSP is NP-complete. *Theoretical Computer Science*, 4:237–244, 1977.

[81] F.P. Preparata and M.I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, NY, 1985.

[82] S. B. Rao and W. D. Smith. Approximating geometrical graphs via "spanners" and "banyans". In *Proceedings of the 30th Annual ACM Symposium on Theory on Computing*, 1998.

[83] S. B. Rao and W. D. Smith. Improved approximation schemes for traveling salesman tours. In *Proceedings of the 30th Annual ACM Symposium on Theory on Computing*, 1998.

[84] G. Reich and P. Widmayer. Beyond Steiner's problem: A VLSI oriented generalization. In *Proceedings of the 15th International Workshop Graph-Theoretical Concepts in Computer Science*, volume 411 of *LNCS*, pages 196–210, 1989.

[85] J. S. Salowe. Construction of multidimensional spanner graphs with applications to minimum spanning trees. In *Proceedings of the 7th ACM Symposium on Computational Geometry*, pages 256–261, 1991.

[86] B. Schneider. Geomorphologically sound reconstruction of digital terrain surfaces from contours. In T.K. Poiker and N. Chrisman, editors, *Proceedings of the 8th International Symposium on Spatial Data Handling*, pages 657–667, 1998.

[87] D.S. Scott and S.S. Iyengar. TID: a translation invariant data structure for storing images. *Communications of the ACM*, 29(5), 1986.

[88] M. I. Shamos and D. Hoey. Closest-point problems. In *Proceedings of the 16th Annual IEEE Symposium on Foundations of Computer Science*, pages 151–162, 1975.

[89] M. Sharir. Davenport-Schinzel sequences and their geometric applications. In R. A. Earnshaw, editor, *Theoretical Foundations of Computer Graphics and CAD*, volume F40 of *NATO ASI*, pages 253–278. Springer-Verlag, Berlin, Germany, 1988.

[90] V. Soltan and A. Gorpinevich. Minimum dissection of rectilinear polygon with arbitrary holes into rectangles. *Discrete Computational Geometry*, 9:57–79, 1993.

[91] D.M. Theobald and M.F. Goodchild. Artifacts of TIN-based surface flow modelling. In *Proceedings of the Annual Conference on GIS/LIS*, pages 955–964, 1990.

[92] M. Thorup. Undirected single-source shortest path with positive integer weights in linear time. *Journal of the ACM*, 46(3):362–394, 1999.

[93] P. M. Vaidya. A sparse graph almost as good as the complete graph on points in K dimensions. *Discrete Computational Geometry*, 6:369–381, 1991.

[94] A. Wiernik and M. Sharir. Planar realization of nonlinear Davenport-Schinzel sequences by segments. *Discrete Computational Geometry*, 3:15–47, 1988.

[95] M. Yamashita, T. Ibaraki, and N. Honda. The minimum number cover problem of a rectilinear region by rectangles. *Bulletin of the European Association for Theoretical Computer Science*, 24:138, 1984.

[96] C. K. Yap. An $O(n \log n)$ algorithm for the Voronoi diagram of a set of simple curve segments. *Discrete Computational Geometry*, 2:365–393, 1987.