Construction and Analysis of Geometric Networks

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften /
Doktors der Naturwissenschaften

der Fakultät für Informatik
der Universität Fridericiana zu Karlsruhe (TH)

vorgelegte

Dissertation

von

Marc Benkert

aus Berlin

Tag der mündlichen Prüfung: 3. Mai 2007

Erster Gutachter: Herr PD Dr. Alexander Wolff

Zweiter Gutachter: Herr Dr. Marc van Kreveld

Contents

Introduction

Each time we open a road map, we try to perceive some information about the location of the depicted places or connections by which one place can be reached starting from another place. In most of the cases our aim is to plan a route that should be as short as possible concerning its time consumption. We do this by analyzing the depicted geometric information in a simple way; by the color or width of a road we know its type. Hence, together with the length information about each road section we can—under the idealistic assumption that there will be no traffic jam—estimate how long we probably need for certain routes.

Figure 1: Road map of Schwenningen and its vicinity, 1939.

Each time a graph theorist opens a road map, he sees a network that is endowed with geometric information and other attributes of the objects contained. Well, indeed, the graph theorist sees the same as we do but in a different perspective. The reason for this is that he is aware of the theoretic concept that is used to illustrate this road network, for him, it is a *graph*.

A graph consists of *vertices* (also called *nodes*) and *links* (also called *edges*) that connect

the vertices. In the case of the road map network, the vertices are the depicted cities or junctions and the links are the roads that connect them. Additionally, the links can be endowed with weights, which, for the road map, indicate road lengths or traveling times. Other attributes, like the color of a road, record additional information. Naturally, attributes can also be set for the displayed vertices. When we plan our trip it is important whether the roads that we want to use are rural roads, highways or expressways because this significantly affects the traveling time. For the graph theorist our endeavors to find the best possible route by hand simply come down to performing a *shortest-path query* in an appropriate graph. If the tour should minimize the petrol costs, the links of the graph should be weighted by the amount of petrol needed for traveling along the corresponding road. If the tour should minimize the required time, the links should be weighted by the required traveling time.

However, the graph theorist does not need any explicit geometric information to perform a shortest-path query. For him, it is enough simply to know the underlying graph with all its vertices and links, i.e. the possibilities of going from any vertex to any other. A shortest-path query then finds the best possibility to go from vertex A to vertex B by checking intermediate vertices and possibly all links of the graph. In this sense the query can be dealt with using graph-theoretic concepts only. Graph theory in general and shortest-path queries in particular form a wide field, which is already quite well exploited and which will not be discussed in this thesis. We are interested in networks that are additionally endowed with geometric information. What makes the road map a geometric network is the fact that the real topology of the depicted places is preserved by its rendering. This helps us enormously in understanding the geometric information that should be conveyed.

This leads us to the first discipline when dealing with geometric networks: their *visualization*. Basically, there are three disciplines:

Visualization – Which geometric properties do we want to represent and how do we accomplish that?

Construction – How do we construct geometric networks that fulfill some desired properties?

Analysis – What information can we extract from a geometric network and how can we make use of this information?

When we are looking for a good visualization, the network itself is given and we look for a representation that maintains or emphasizes predefined geometric information. This representation is usually an embedding of the network in the plane, i.e. in 2-dimensional space. To accomplish that aim, one usually proceeds in two steps. Firstly, an embedding for the vertices is fixed, i.e. for each vertex its location in the plane is determined. Secondly, we decide on the type of geometric object with which the links shall be illustrated and the links are added to the drawing of the embedding. Typical link types are straight lines, polygonal lines, or curved lines. Which type we choose depends on the purpose that the link representation should fulfill and, often enough, on the beauty of the emerging rendering. For example, if we want to illustrate a graph that can be drawn in the plane such that no two links intersect, in the majority of the cases we want to find a representation that preserves this property, if no other properties we want to emphasize prohibit this. There may be applications which request an embedding of a geometric into 3-dimensional space, however, the scope of this thesis will only encompass networks in 2-dimensional space.

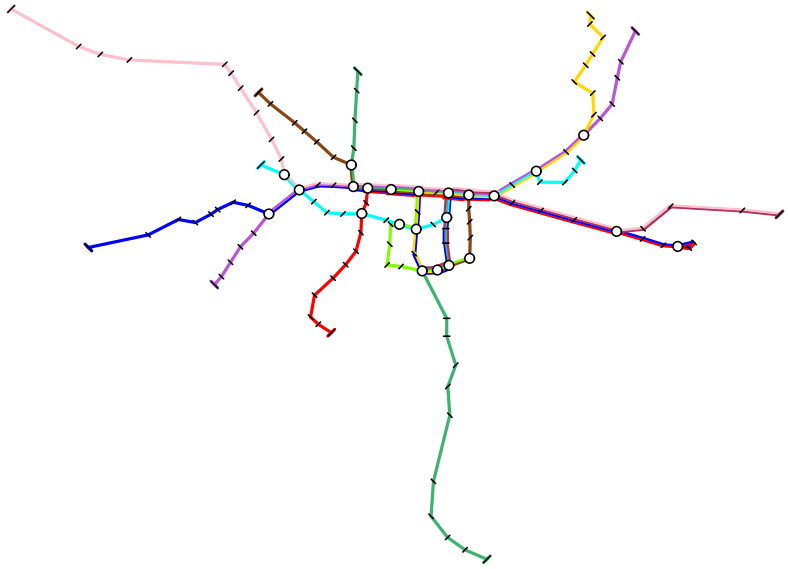
For the road map, finding a vertex embedding is in general a trivial issue: as there is no reason not to preserve the exact topology, the embedding is simply found by appropriately scaling the original data. However, for small scales there arises a different problem: we have to decide which dispensable vertices we skip. This is necessary simply because the map does not offer enough space anymore to display and label all vertices. After fixing the vertex embedding, we insert the roads into the network. This step also depends on the used scale and we first have to decide which roads we will incorporate. A road is then drawn as a polygonal or curved line that reasonably reflects its real geometric course, if there is enough space to do so. If the scale is too small to allow an authentic drawing we have to appropriately deform the original course such that the characteristic trait of the road is still maintained. This is, e.g., relevant for strongly winding mountain roads. Generating maps at different scales is all but a trivial task and cartographs devote a lot of work to these kind of *generalization* problems. For a survey on common generalization techniques see for example [Wei97].

A typical example for which it is not appropriate to heavily rely on the real geometric data, when we want to find a good visualization, is the metro map. First and foremost we look for a visualization of this transportation network that meets the customers demands as well as possible. Each single line should be nicely traceable, interchange facilities have to be marked in an understandable way and so on. However, additionally, the real topology should still be recognizable. Otherwise this would lead to confusions in the understanding of the map since the customer has, in most cases, at least a rough knowledge about the geography of the depicted region. A very common solution is to start with the original topology and deform it only slightly, see Figure 2 for an example. Figure 2a visualizes the transportation network of the city of Karlsruhe utilizing the real coordinates of the stations. Figure 2b depicts the resulting visualization of Nöllenburg and Wolff's approach [NW06]. They used a mixed-integer program to encode and optimize the criteria that have been touched on above.

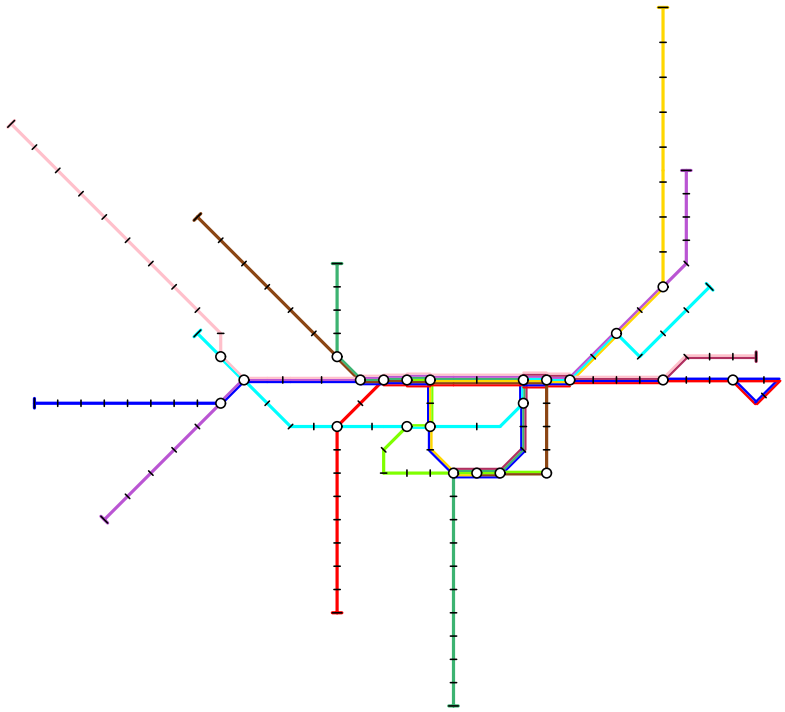
To summarize, a good visualization is always a visualization that meets the customer demands. This concludes my short overview on the discipline of visualization. In my thesis, visualization will only play a small role since the embedding of the vertices will already be fixed by the input. So, visualization will only be used for illustration purposes.

Network construction is the discipline that asks for the construction (or extension) of a network on a given set of vertices, or a given part of the network such that the outcome fulfills some predetermined properties. Almost always, these properties include the optimization of some quality criterion. Again, road maps provide an illustrative example. Assume for the moment that there haven't been built any roads and we want to set up a road network for a set of cities from scratch. This road network should connect the cities in a good way, but what does 'a good way' intrinsically mean? Well, an obvious requirement is the connectivity of the network, since it should be possible to get from any city to any other. But in most cases connectivity alone is not enough. Assume that we live in city A and want to visit a friend who lives in city B. Understandably, there is a limit on the detour that we are willing to accept compared to the air-line distance between A and B. Then why don't we just build a road between every pair of cities? Simply because this would be disastrously uneconomical, not to mention the unnecessary destruction of too much nature. This is best illustrated by two plausible examples; see Figure 3. For three nearly collinear cities it would make no sense to set up a direct connection between the outermost cities when there are roads connecting the outermost cities with the mid city. For two cluster of cities that lie far apart, one connecting road is enough to serve for low inter-cluster detours.

The technical term for the concept that was just described is *spanners with low weight*.



(a) Original topology.



(b) Map produced by the approach of Nöllenburg and Wolff [NW06].

Figure 2: Comparison between two representations of the Karlsruhe transportation network.

(a) Three nearly collinear cities. **(b)** Two city clusters.

Figure 3: Sparse road networks that serve low-detour connections.

And indeed, in practice, it also makes no sense to build a dense road network. For every arrangement of cities a surprisingly sparse network can be found that provides connections that do not let the maximum occurring detour get too big. A good overview to the theory of spanners is given in a survey paper by David Eppstein [Epp00]. Spanners play a role in the first to chapters of my thesis and for this reason a more formal introduction is given in Section 0.1.1.

Back to the (geometric) properties in our road-map example. The basic property is connectivity, the two other criteria are the building cost and the maximum occurring detour that shall be acceptable. In terms of the optimization criterion we have two choices to describe a 'good' network. Either we try to establish a network that offers connections between any two cities not exceeding a predefined detour value and minimize the building cost, or we fix the amount of money that we can afford and try to set up a network that minimizes the maximum occurring detour under this restriction, see e.g. [SZ04]. The latter scenario seems to be the more practical one. There are also examples of larger practical relevance than setting up a road network from scratch, e.g. the planning and building of bypass roads [FGG05]. Here, an existing network is to be extended in an optimal way. The prevailing optimization criterion is the ability of the resulting network to cope with the expected volume of traffic.

In Chapters 1, 2 and 3 we consider problems of the above kind. Together they constitute the first part of my thesis: *constructing geometric networks*. The considered problems all have in common that the embedding of the vertices (cities) is already fixed by the input and the task is to set up a network on these sets that fulfills given properties and optimizes a certain criterion. A more detailed description to each of these problems can be found in the outline of this thesis in Section 0.1.

The last of the three disciplines is the *analysis of geometric networks*. Obviously, this discipline encompasses analyzing a network for special—unknown—geometric properties. E.g., if we are given a road network, we want to know how big the maximum occurring detour is and for which pair of cities it is attained. Beyond analyzing for properties I want to classify a second model to this discipline. Assume that we do already know the geometric properties of the network we are dealing with. Now, we want to utilize these properties to support other analysis techniques, such as shortest-path queries. Indeed, in graph theory there are two approaches that use geometric information to speed-up the search for a shortest path from A to B. Both approaches share the same idea: the search is guided by geometric information provided by the embedding of the network. In the first approach [SV86], vertices that lie geometrically closer to the target B (Euclidean distance) are considered as intermediate vertices for a shortest A–B path before vertices that lie further away. In the second approach [WWZ06], *geometric containers* are used. A preprocessing step is used, computing for every edge $\{A, C\}$ the candidate vertices that can be reached by a shortest path starting from vertex A and using edge $\{A, C\}$. This information is stored in a geometric container,

represented by the smallest axis-aligned bounding box that contains all candidates. Obviously this information helps to guide the search; we only have to check edges that actually can lead to B when we are looking for a shortest path from A to B. So far I have concealed that both approaches make use of *Dijkstra's Algorithm*. Wagner et al. [WW07] give a brief introduction to this pioneering algorithm and a survey on speed-up techniques.

In my thesis, *analyzing geometric networks* constitutes the second part. In Chapter 4 the traces of moving objects are investigated and analyzed for their group behavior and in Chapter 5 a so-called *nearest-neighbor graph* is analyzed for its maximum degree and this information is used to bring the algorithm for a packing problem to work. For a detailed outline of these problems see Sections 0.2.4 and 0.2.5.

To summarize this introduction, geometric networks appear in many circumstances in everyday life. They are the most important data structure for modeling flows of traffic, goods or information. Constructing geometric networks that satisfy certain properties is a necessary discipline for the utilization of these properties, e.g. in railway and road network planning or in the design of VLSI layouts. An appropriate visualization is the tool for providing a good understanding of the information (and properties) that should be conveyed to an observer. Analyzing and understanding geometric networks helps to make use of the properties.

Two sections conclude this introduction. In Section 0.1 I introduce two basic concepts from computational geometry and in Section 0.2 I sketch the content of each chapter and summarize the results.

0.1 Concepts from Computational Geometry

On several occasions in this thesis *t-spanners* occur and the technique of *range searching* is used. These concepts are well known in Computational Geometry. I introduce them briefly in the next two sections.

0.1.1 t-spanners

One can explain the essence of a *t-spanner* in a single sentence: Given a set of points in the plane and a constant $t \geq 1$, a Euclidean *t-spanner* is a network in which, for any pair of points, the ratio of the network distance and the Euclidean distance of the two points is at most t.

As we will need the notion of a t-spanner on several occasions in this thesis, I will give a detailed definition. Let P be a set of n points in the plane. For two points $u, v \in P$ let $|u, v|$ denote the Euclidean distance between u and v. Let $G = (P, E)$ an undirected, connected graph on P having straight-line edges weighted by the Euclidean distance of the corresponding edge. Let $|u, v|_G$ denote the length of a shortest u–v path in G. The *stretch factor* $\delta(u, v)$ of a point pair u, v is defined as:

$$\delta(u, v) = \frac{|u, v|_G}{|u, v|}.$$

The *stretch factor* δ_G of the graph G is defined to be the maximum occurring stretch factor between any pair of points:

$$\delta_G = \max_{u,v \in G, u \neq v} \delta(u, v).$$

Frequently the term *stretch factor* is also referred to as *dilation*. A graph is said to be a *t-spanner* if its stretch factor is at most t, see Figure 4. The complete graph on P is obviously a 1-spanner, but as the complete graph has $\Theta(n^2)$ edges it is practically irrelevant. For most applications a good spanner with a small number of edges is desired, as seen in the example of a road network in the introduction. The *Yao-Graph* was the first graph with a linear number of edges that provably had a constant stretch factor [Yao82, Kei88, RS91]. There are a variety of other properties that a spanner is often desired to supply, which depend on the application, e.g. bounded degree, small total edge length, small spanning diameter and the required construction time of the spanner. The comparison criterion for small total edge length is the length of a minimum spanning tree on P. The *spanning diameter* of a graph is the length of the longest shortest path when all edges are weighted by 1. It gives the maximum number of hops that are required to get from one point to another.

However, the field of Euclidean spanners has already been well studied since Chew [Che89] introduced them. For the understanding of my thesis, only the definition is necessary. I conclude with two references that cover the theory of spanners as far as possible: *Euclidean Spanners: Short, Thin and Lanky* by Arya et al. [ADM+95], and *Spanning trees and spanners* by David Eppstein [Epp00].

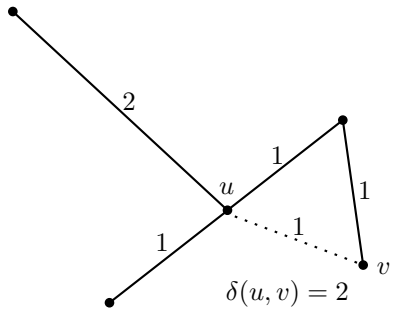


Figure 4: A 2-spanner and the point pair u, v that defines the stretch factor of the graph.

0.1.2 Range searching

In Chapter 2 and Chapter 4 performing range queries plays a crucial role for the algorithmic realization of the desired demands. For this reason, I introduce the basic idea of *range searching* here and give a very simple example showing how an approximate range query can be conducted.

Assume we want to answer queries of the following type: we are given a set P of points in the plane and a query object, e.g. a disk D, and we want to efficiently report the points in $P \cap D$. Obviously we can traverse all points and simply check whether each point lies in D or not, which would take linear time if the query object is of constant complexity. If we only have to deal with a single query object this would be the best we could do, but what happens if we have to process a whole series of different query objects? Then, it makes sense to preprocess the point set and establish a data structure that can answer queries in sublinear time. In the introduction of his survey paper on range searching [Aga97], Agarwal nicely sketched the technique that usually forms the basis for such a data structure: a set of disjoint canonical subsets of P is stored. Roughly speaking, the query will then make use of this information by deciding whether whole subsets lie inside or outside the query range. We will further

illustrate this in a small example below. The notion of storing point subsets immediately suggests a trade-off between space requirement and query time. A data structure suggested by Matoušek [Mat93] is a nice example of this. For a user-defined parameter $M \geqslant n$ he builds a data structure of size $O(M)$ in $O(n^{1+\delta} + M \operatorname{polylog} n)$ time, where δ is an arbitrary small positive constant, such that queries can be answered in $O((n/M^{1/3}) \operatorname{polylog} n)$ time.

Usually, the stored subsets are arranged in a tree-like structure. We show this for an *approximate* range query. If accuracy is not required we can approximate in the following sense. Say we have given a query object, e.g. a disk D, and the diameter of the object is d. We give the concession that any point outside D, but not further than $\varepsilon \cdot d$ distance away from D may or may not be counted for the query. Here, ε is some chosen, positive constant that determines the grade of exactness. Obviously, dealing with approximate range queries is algorithmically easier than with exact queries as we do not have to care about the exact location of points close to the boundary of D. For example, say that we want to query a disk D and we only want to output the (approximate) number of points in D (*range counting query*). We use a *split tree* to preprocess and store the points. The root of this tree is assigned a square that contains all input points. As additional information for each vertex, we store the number of points contained in its corresponding region. Hence, for the root, this number is $|P|$, see Figure 5. If the corresponding square R of a vertex in the split tree contains at most one point, the vertex is a leaf. Otherwise the vertex has four children that correspond to the four equally-sized squares that emerge when R is further partitioned, for more detailed information about split trees see for example [dBvKOS00].

Let $D_{1+\varepsilon}$ denote the region that contains D and the points not farther away from D than $\varepsilon \cdot d$, i.e. the enlarged query region. Recall that points in $D_{1+\varepsilon} \setminus D$ may or may not be counted. The query starts at the root of the split tree. For every vertex v that is encountered by the query, we check whether the corresponding square (i) lies completely within $D_{1+\varepsilon}$, (ii) lies completely outside of D or (iii) has non-empty intersection with D as well as with the outside of $D_{1+\varepsilon}$. For (i) we count the number of points in the square for the query; for (ii) we don't. Neither in case (i) nor in case (ii) do we have to descend further into the subtree of v. For (iii) we cannot make any decision and must descend further into the subtree of v.

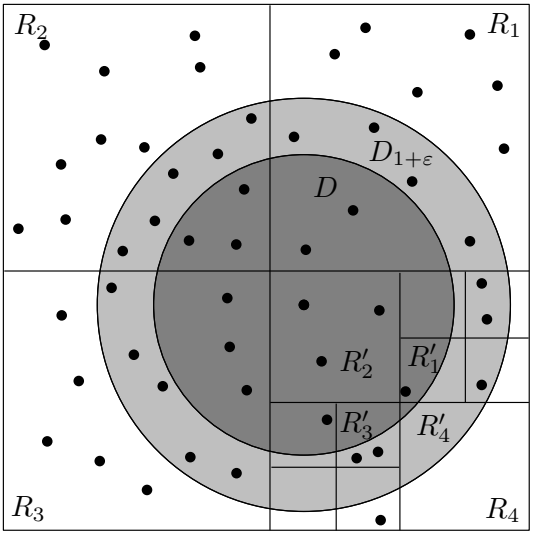
Note that the exact query time depends on many different matters, e.g. the height of the tree, additional information that is stored with the vertices and not least the time that is required to decide whether a point subset has non-empty intersection with the query object or not. The above example was only chosen to illustrate the functionality of a range query. Since a normal split tree, as described above, can have linear height, the worst-case complexity of a query in the example is still linear.

0.2 Thesis outline

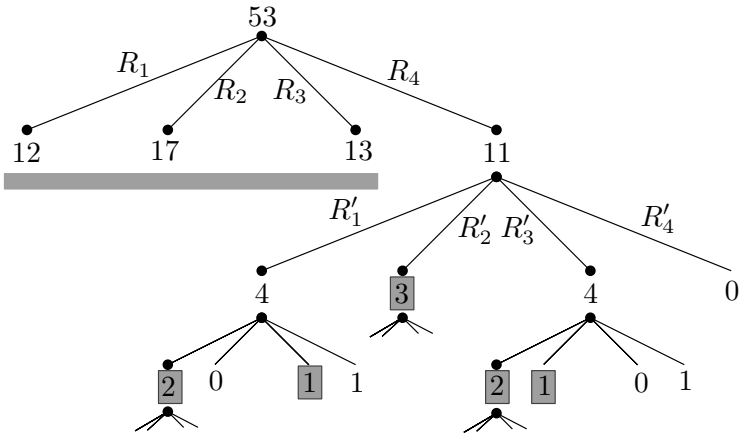
The contents of the five chapters of this thesis are outlined in the next sections. Chapters 1–3 constitute the *constructing* part and Chapters 4–5 constitute the *analyzing* part.

0.2.1 Manhattan Networks – Chapter 1

This work is concerned with the study of 1-spanners under the Manhattan (or L_1-) metric. The Manhattan metric differs significantly from the Euclidean metric, for which a shortest path is always realized by a straight-line. For the Manhattan metric a shortest path between two points is not uniquely defined: any shortest rectilinear path that connects the two points



(a) A point set P and (parts of) the subdivision into squares that builds the basis for the split tree.



(b) The split tree for P and the above subdivision. The subtrees of R_1, R_2 and R_3 are omitted. The query for D is illustrated in the subtree rooted at R_4. Shaded numbers are counted for the query; it is not necessary to descend further.

Figure 5: Performing an approximate range query using a split tree.

realizes the Manhattan distance. We call such a path a *Manhattan path*. The naming stems from the street system of Manhattan where almost all streets lead North–South (Avenues) or West–East (Streets).

A network that provides Manhattan paths between any two participating points is called a *Manhattan network*. More precisely, a Manhattan network for a set of points is a set of axis-parallel line segments whose union contains a Manhattan path for each pair of points. Two Manhattan networks for the same point set are depicted in Figure 6.

Finding a Manhattan network for a given point set is a trivial task. Take, for example, the complete rectilinear grid that is induced by the input points. However, the problem of finding a *minimum* Manhattan network (MMN), i.e. a Manhattan network of minimum total length, is more complicated. By now, it is still not known whether such a network can be computed in polynomial time or whether this problem is $\mathcal{NP}$-hard. In Chapter 1 we present an approximation algorithm for the problem. Given a set P of n points, our algorithm computes in $O(n \log n)$ time and linear space a Manhattan network for P whose length is at most 3 times the length of an MMN for P.

(a) minimum MN, length: 36.

(b) MN computed by our algorithm, length: 41.

Figure 6: Manhattan networks

We also establish a mixed-integer programming formulation for the MMN problem. With its help we extensively investigated the performance of our factor-3 approximation algorithm on random point sets. For two types of artificially generated point sets we found out that the practical performance of our algorithm was significantly better than the theoretical bound may suggest. On average our algorithm performed by a factor of approximately 1.3.

At the time our work was published [1] our algorithm had the provably best approximation factor, disregarding a 2-approximation algorithm (Kato et al. [KIA02]) whose correctness proof was incomplete. In [1] we also introduced a mixed-integer progam formulation of the problem in order to be able to solve instances optimally and to compare the results yielded by our approximation algorithm.

In the meantime the French researchers Chepoi, Nouioua and Vaxés [CNV05] have succeeded in generating a 2-approximation algorithm. They did this by relaxing the integer variables of our mixed-integer program and rounding them in an appropriate way. However, their algorithm must solve a linear program with $\Theta(n^3)$ variables and constraints. Thus, it is much slower than our algorithm.

The Manhattan network problem may have applications in city planning or VLSI design. Lam et al. [LAP03] also describe an application from computational biology that stems from the comparison of gene sequences. In Chapter 1 this application is described in more detail.

0.2.2 Interference-minimal Networks – Chapter 2

A wireless ad-hoc network can be represented as a graph in which the nodes represent wireless devices, and the links represent pairs of nodes that communicate directly by means of radio signals. The *interference* caused by a link between two nodes u and v can be defined as the number of other nodes that may be disturbed by the signals exchanged by u and v. This definition was first made by Burkhart et al. [BvRWZ04].

In this work we deal with the following task. Given the position of the nodes in the plane, links are to be chosen such that the maximum interference caused by any link is minimized, under the restriction that the network fulfills a certain desirable property. The properties that we are interested in are connectivity, bounded dilation and bounded link diameter. We give efficient algorithms to find networks in two different models. In the first model, the signal sent by u to v reaches exactly the nodes that are not farther from u than v is. In the second model, we assume that the boundary of a signal's reach is not known precisely and that our algorithms should therefore be based on acceptable estimations. More precisely this means that a signal sent from u to v reaches v in any case, but is definitely lost if distance to u exceeds $(1+\varepsilon)|u,v|$ where $|\cdot|$ denotes the Euclidean distance and ε is a small positive constant. The latter model yields faster algorithms. Furthermore, the running times of our algorithms in the exact model are output-sensitive. They depend on the smallest interference value k for which a network with the desired property exists. For example, we need $O(nk^2+n\log n\log k)$ time for finding the minimum-interference spanning tree in the exact model and $O(n/\varepsilon^2(1/\varepsilon+\log n))$ for finding it in the estimation model.

0.2.3 Boundary Labeling – Chapter 3

When endowing a map with information about the depicted places, it might happen that there is simply not enough space for all labels to lie directly at the point they refer to (e.g. metropolitan areas), or it is not desired that labels lie in the map because they would occlude other important information (e.g. in medical atlases). A common method for such cases is to place the labels around the map and to connect a point with its label by an arc. To ease the understanding of the point-label assignments it has turned out that equally-shaped arcs are convenient for visual perception. In Chapter 3 we give algorithms that compute such labelings with labels located on the boundary of the map. We formalize the setting as follows: n points are contained in a rectangle R. They are to be connected to rectangular labels that lie either on one or on two opposite sides of R. An assignment from a point p to a label ℓ is indicated by a polygonal line, which we call *leader*, leading from p to ℓ. A first obvious restriction that serves clarity is that no two leaders are allowed to intersect. We consider two types of shapes that we use for the leaders: rectilinear leaders, called *po-leaders*, that consist of a horizontal and a vertical segment, and leaders that consist of a horizontal and a diagonal segment, called *do-leaders*. Figures 7a and 7b show examples of the two types. Both types of leaders have at most one bend but *direct* leaders consisting of a single horizontal segment are explicitly allowed and desired for a nice labeling.

The algorithmic task now is to determine a good (bijective) assignment from the set of points to the set of labels. This assignment can be seen as a bipartite graph with the points and labels being the vertices and the leaders being the edges. Consequently, the problem is a mixture of a graph-drawing problem and a classical labeling problem.

To simplify visual perception we can optimize for two criteria: the total number of bends and the total leader length in a labeling. We first consider optimizing one criterion inde-

(a) *po*-leaders. **(b)** *do*-leaders.

Figure 7: Leader types

pendently from the other. The length-optimization variants of the described problems can basically be solved by sweep-line algorithms. The bend-minimization variants do not allow sweep-line algorithms in most cases; algorithms based on dynamic programming can be applied, however. This of course is reflected in the running times of the presented algorithms. For example, the version with labels on one side of R using *po*-leaders takes $O(n \log n)$ time for length minimization while our algorithm for bend minimization requires $O(n^3)$ time. By the $O(n \log n)$ algorithm for the *po*-leaders we improve a quadratic algorithm of Bekos et al. [BKSW07]. The new bound is tight.

Labeling a point set with *do*-leaders is more difficult than labeling with *po*-leaders. The primary reason for this is that there are points that cannot connect to any label, e.g. in Figure 7b the topmost point cannot connect to the bottommost label—indicated by the dotted line—by a *do*-leader. Hence, the running time for *do*-leaders is worse than for *po*-leaders. For one-sided length minimization our algorithm requires $O(n^2)$ time, and for bend minimization it requires $O(n^5)$ time.

We implemented our algorithms to evaluate their practical behavior. In addition to the practical evaluation we also discuss the advantages and disadvantages of the two leader types and the two optimization criteria.

0.2.4 Detecting Flocks – Chapter 4

Data representing moving objects is rapidly becoming more widely available, especially in the area of wildlife GPS tracking. It is a central belief that information is hidden in large data sets in the form of interesting patterns. One of the most commonly sought-after spatio-temporal patterns is the flock. A *flock* is a large subset of objects moving along paths close to each other for a certain pre-defined time. In Chapter 4 we give a new definition that we argue is more realistic than previous ones, and by the use of techniques from computational geometry we present fast algorithms to detect and report flocks.

Our definition of flock involves three parameters: the flock size $m \in \mathbb{N}$, the flock time span $k \in \mathbb{N}$ and the flock radius $r \in \mathbb{R}^+$. A flock must consist of at least m entities that move together for at least k consecutive time steps. We define *together* to mean that for each of the k time steps there is a disk of radius r that contains all flock entities, see Figure 8.

We assume that the trace of each entity participating in the input is given by a polygonal line (trajectory) with k vertices. The k vertices indicate the position of the entity at a set of predefined points in time. If more than k positions of the entities are given, our algorithm will look at each time span of k consecutive steps independently.

Our idea for the algorithmic accomplishment of detecting flocks in these sets of trajectories is to canonically map each trajectory to a point in $\mathbb{R}^{2k}$ and then apply range reporting and counting queries in $\mathbb{R}^{2k}$. If we use the right type of query objects, these queries detect all flocks

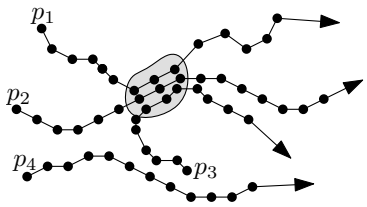


Figure 8: The entities p_1, p_2 and p_3 form a flock for a period of 3 time steps.

and can report the entities that belong to a detected flock. We give several approximation algorithms where the approximation is with respect to the radius r.

Apart from the total number of observed entities, the parameter k is crucial for the running time of our algorithms since, in general, range queries become significantly slower the higher the dimension is. We evaluated our algorithms and found that they are still practical for $k = 16$ time steps on meaningful input data.

0.2.5 Packing Disks – Chapter 5

A variant of a classical packing problem forms the basis of this chapter: in a rectangle ρ that already contains a set $\mathcal{R}$ of packed unit disks, further unit disks are to be packed such that neither of the newly packed disks intersects any other disk. Packing the maximum number m of non-intersecting unit disks into a polygonal area is known to be $\mathcal{NP}$-hard [FPT81]. On the positive side, Hochbaum and Maass [HM85] have proven the existence of a polynomial-time approximation scheme (PTAS). We approximate in a different sense. Let α be a fixed real in $(0, 1]$. Our task is to pack a set $\mathcal{B}_\alpha$ of at least m disjoint disks of radius α into ρ such that no disk in $\mathcal{B}_\alpha$ intersects a disk in $\mathcal{R}$. Note that m, the maximum number of unit disks that can be packed, is neither known a priori nor will it be known after our algorithm has finished.

Since this variant of the problem is similar to the approximation of the number of unit disks, it is somewhat surprising that there is no PTAS. Baur and Fekete [BF01] showed that for square objects the problem cannot be solved in polynomial time for any $\alpha > 13/14$, unless $\mathcal{P} = \mathcal{NP}$. Their inapproximability proof adapts to disks but with a constant much closer to 1 than 13/14. In Chapter 5 we present a polynomial-time algorithm for packing disks with $\alpha = 2/3$.

In our approximation, the analysis of a *nearest-neighbor* graph plays a crucial role. First, we make use of Hochbaum and Maass' PTAS to compute a set $\mathcal{B}_1$ of at least $8m/9$ disjoint unit disks. Next, we define a distance measure d for two disks in $\mathcal{B}_1$. Roughly speaking, d measures the length of the shortest path that translates the two disks without intersecting any of the predefined disks in $\mathcal{R}$. We compute the nearest-neighbor graph G on $\mathcal{B}_1$ with respect to d. The vertices in G correspond to the disks in $\mathcal{B}_1$ and an edge indicates a nearest-neighbor relation between the two incident disks. We prove that G has bounded maximum degree, which helps to find a lower bound on the number of matched disks in a maximum matching of G. Then, we assign a region to each matching pair such that all regions are pairwise disjoint and do not intersect any other disk in $\mathcal{B}_1$. Finally, we place three disks of radius 2/3 in the region of each matching pair and one disk of radius 2/3 into each unmatched disk in $\mathcal{B}_1$. We can show proceeding like this we place at least 9/8 many disks as there are in $\mathcal{B}_1$. Recall that $\mathcal{B}_1$ contains $8m/9$ unit disks. Hence, we have placed at least m disks of radius 2/3.

This dispersion problem may have applications in non-photo realistic rendering system,

where 3D models are to be rendered in an oil painting style. Another application is given by sample surveys of particular regions in the plane, e.g. soil ground. To serve a high quality of the sample the measuring should be nicely distributed.

Part I

Constructing Geometric Networks

Chapter 1

Manhattan Networks

Constructing a Manhattan network constitutes the first chapter of the *constructing* part of this thesis. The geometry element is given by the input, n points in the plane. The requirement on the network is the supply of shortest Manhattan paths between any point pair. The criterion that is to be minimized is the total length of the segments contained in the network.

The chapter is based on journal publication [2]: Marc Benkert, Alexander Wolff, Florian Widmann and Takeshi Shirabe: "The minimum Manhattan network problem: Approximations and exact solutions".

1.1 Introduction

Under the Euclidean metric, in a 1-spanner (which is the complete graph) the location of each edge is uniquely determined. This is not the case in the Manhattan (or L_1-) metric, where an edge $\{p, q\}$ of a 1-spanner is a *Manhattan p–q path*, i.e. an x- and y-monotone rectilinear path between p and q. A 1-spanner under the Manhattan metric for a finite point set $P \subset \mathcal{R}^2$ is called a *Manhattan network* and can be seen as a set of axis-parallel line segments whose union contains a Manhattan p–q path for each pair $\{p, q\}$ of points in P.

In this chapter we investigate how the extra degree of freedom in routing edges can be used to construct Manhattan networks of minimum total length, so-called *minimum Manhattan networks* (MMN). The MMN problem may have applications in city planning or VLSI layout, but Lam et al. [LAP03] also describe an application in computational biology. For aligning gene sequences they propose a three-step approach. In the first step, they use a local-alignment algorithm like BLAST [AGM$^+$90] to identify subsequences of high similarity, so-called *high-scoring pairs* (HSP). In the second step they compute a network for certain points given by the HSPs. They do not require that each point be connected by Manhattan paths to *all* other points, but only to those that have both a larger x- and a larger y-coordinate. A Manhattan path in their setting corresponds to a sequence of insertions, deletions, and (mis)matches that are needed to transform one point representing a gene sequence into another. Lam et al. show that modifying an algorithm by Gudmundsson et al. [GLN01] yields a $O(n^3)$-time factor-2 approximation for their problem. They state that the restriction to the network they compute helps to considerably reduce the size of the search space for a good alignment, which is computed by dynamic programming in the third step of their approach.

1.1.1 Previous work

The MMN problem has been considered before, but until now, its complexity status is unknown. Gudmundsson et al. [GLN01] have proposed an $O(n \log n)$-time factor-8 and an $O(n^3)$-time factor-4 approximation algorithm. Later Kato et al. [KIA02] have given an $O(n^3)$-time factor-2 approximation algorithm. However, the correctness proof of Kato et al. is incomplete. Both the factor-4 approximation and the algorithm by Kato et al. use quadratic space. After the journal version on which this chapter is based had been submitted, Chepoi et al. [CNV05] gave a factor-2 approximation algorithm based on linear programming. We now briefly sketch these algorithms.

Gudmundsson et al. [GLN01] considered each input point p separately. From p they established Manhattan paths to those points p' where the bounding box of p and p' contained no other input point. This yields a Manhattan network. In order to establish the paths from p to all points p', they considered the points p' in each of the four quadrants relative to p simultaneously. In each of the quadrants, these points define a staircase polygon. The points p' are connected to p by rectangulating the staircase polygon, minimizing the length of the segments used for the rectangulation. Solving this subproblem by a factor-2 approximation algorithm yields the factor-8 approximation algorithm for the MMN problem while using dynamic programming to solve the subproblem optimally yields the factor-4 approximation.

Kato et al. [KIA02] observed that it is not always necessary to connect p explicitly to all points p'. Instead, they came up with the notion of a *generating set*, i.e. a set of pairs of points with the property that each network that contains Manhattan paths between these point pairs is already a Manhattan network. In a first step they constructed a network N' whose length is bounded from above by the length of an MMN. Kato et al. designed the network N' such that it contains Manhattan paths for as many point pairs in the generating set as possible. They claimed that in a second step, they could rectangulate the facets of N' such that the remaining unconnected point pairs are connected and the total length of the new segments is again bounded from above by the length of an MMN. Both the details of this step and the proof of its correctness are missing in [KIA02].

Chepoi et al. [CNV05] use the relaxation of the mixed-integer program that we introduce in Section 1.7 and that we published before [1]. Their algorithm is based on cleverly rounding the solution of this linear program which uses $O(n^3)$ variables and constraints. Thus, their algorithm is much slower than all previous algorithms, including ours.

1.1.2 Our results

In this chapter we present an $O(n \log n)$-time factor-3 approximation algorithm. We use the generating set of [KIA02], and we also split the generating set into two subsets for which we incrementally establish Manhattan paths. However, our algorithm is simpler, faster and uses only linear (instead of quadratic) storage. The main novelty of our approach is that we partition the plane into two regions and compare the network computed by our algorithm to an MMN in each region separately. One region of the partition is given by the union of staircase polygons that have to be pseudo-rectangulated. For this subproblem a factor-2 approximation suffices. It runs in $O(n \log n)$ time and is similar to the factor-2 approximation for rectangulating staircase polygons that Gudmundsson et al. [GLN01] proposed.

We also establish a mixed-integer programming (MIP) formulation for the MMN problem. Our formulation is based on network flows. It yields an exact solver that finds MMNs for

small point sets within a bearable amount of time. We implemented our factor-3 approximation algorithm and used the exact solver to measure its performance on random point sets. Further, we make an extensive comparison with other algorithms including the factor-4 and -8 approximations of Gudmundsson et al. [GLN01]. It turns out that our algorithm usually finds Manhattan networks that are at most 50% longer than the corresponding MMN. However, for any $\varepsilon > 0$ there is a point set for which our algorithm returns a Manhattan network that is $(3 - \varepsilon)$ times as long as the corresponding MMN.

In Section 1.2 and 1.3 we give some basic definitions and show how helpful information for our network is computed. In Section 1.4 we detail how the backbone of our network is computed. We describe the algorithm precisely in Section 1.5 and analyze its approximation factor in Section 1.6. In Section 1.7 we give our MIP formulation. In Section 1.8 we use it to evaluate the practical performance of several algorithms. We conclude with some open problems in Section 1.9.

Our algorithm is available as Java applet under URL *http://i11www.ira.uka.de/manhattan*. The applet also features the factor-4 and factor-8 approximation algorithms by Gudmundsson et al. [GLN01].

1.2 Basic definitions

We use $|M|$ to denote the total length of a set M of line segments. For all such sets M we assume throughout this chapter that each segment of M is inclusion-maximal with respect to $\bigcup M$. It is not hard to see that for every Manhattan network M there is a Manhattan network M' with $|M'| \leq |M|$ that is contained in the grid induced by the input points, i.e. M' is a subset of the union U of the horizontal and vertical lines through the input points. Therefore we will only consider networks contained in U. It is clear that any meaningful Manhattan network of a point set P is contained in the bounding box $\mathrm{BBox}(P)$ of P. Finding a Manhattan network for given P is rather easy, e.g. the parts of U within $\mathrm{BBox}(P)$ yield a Manhattan network. However, the point set $\{(1,1),\ldots,(n,n)\}$ shows that this network is not always a good approximation, in this case it is n times longer than an MMN.

We will use the notion of a *generating set* that has been introduced in [KIA02]. A generating set Z is a subset of $P \times P$ with the property that a network containing Manhattan paths for all pairs in Z is already a Manhattan network of P.

The authors of [KIA02] defined a generating set Z with the nice property that Z consists only of a linear number of point pairs. We use the same generating set Z, but more intuitive names for the subsets of Z. We define Z to be the union of three subsets $Z_{\mathrm{hor}}, Z_{\mathrm{ver}}$ and Z_{quad}. These subsets are defined below. Our algorithm will establish Manhattan paths for all point pairs of Z—first for those in $Z_{\mathrm{hor}} \cup Z_{\mathrm{ver}}$ and then for those in Z_{quad}.

Definition 1.1 (Z_{ver}) *Let $P = \{p_1,\ldots,p_n\}$ be the set of input points in lexicographical order, where $p_i = (x_i, y_i)$. Let $x^1 < \cdots < x^u$ be the sequence of x-coordinates of the points in P in ascending order. For $i = 1,\ldots,u$ let $P^i = \{p_{a(i)}, p_{a(i)+1},\ldots,p_{b(i)}\}$ be the set of all $p \in P$ with x-coordinate x^i. Then*

$$\begin{aligned} Z_{\mathrm{ver}} \;=\; & \{(p_i,p_{i+1}) \mid x_i = x_{i+1} \text{ and } 1 \leq i < n\} \\ \cup\; & \{(p_{a(i)},p_{b(i+1)}) \mid y_{a(i)} > y_{b(i+1)} \text{ and } 1 \leq i < u\} \\ \cup\; & \{(p_{b(i)},p_{a(i+1)}) \mid y_{b(i)} < y_{a(i+1)} \text{ and } 1 \leq i < u\}. \end{aligned}$$

See Figure 1.3, where all pairs of Z_{ver} are connected by an edge. Note that Z_{ver} consists

of at most $n-1$ point pairs. If no points have the same x-coordinate, it holds that $Z_{\text{ver}} = \{(p_i, p_{i+1}) \mid 1 \leq i < n\}$, i.e. Z_{ver} is the set of neighboring pairs in the lexicographical order. The definition of Z_{hor} is analogous to that of Z_{ver} with the roles of x and y exchanged. Figure 1.4 shows that $Z_{\text{hor}} \cup Z_{\text{ver}}$ is not necessarily a generating set: Since $(p, h) \in Z_{\text{hor}}$ and $(p, v) \in Z_{\text{ver}}$, no network that consists only of Manhattan paths between pairs in $Z_{\text{hor}} \cup Z_{\text{ver}}$ contains a Manhattan p–q path. This shows the necessity of a third subset Z_{quad} of Z.

Definition 1.2 (Z_{quad}) *For a point $r \in \mathcal{R}^2$ denote its Cartesian coordinates by (x_r, y_r). Let $Q(r,1) = \{s \in \mathcal{R}^2 \mid x_r \leqslant x_s \text{ and } y_r \leqslant y_s\}$ be the first quadrant of the Cartesian coordinate system with origin r. Define $Q(r,2)$, $Q(r,3)$, $Q(r,4)$ analogously and in the usual order. Then Z_{quad} is the set of all ordered pairs $(p,q) \in P \times P$ with $q \in Q(p,t) \setminus \{p\}$ and $t \in \{1,2,3,4\}$ that fulfill*

(a) q is the point that has minimum y-distance from p among all points in $Q(p,t) \cap P$ with minimum x-distance from p, and

(b) there is no $q' \in Q(p,t) \cap P$ with (p,q') or (q',p) in $Z_{\text{hor}} \cup Z_{\text{ver}}$.

Obviously Z_{quad} consists of at most $4n$ point pairs. For the proof that Z_{quad} is in fact sufficient for $Z = Z_{\text{ver}} \cup Z_{\text{hor}} \cup Z_{\text{quad}}$ to be a generating set, see [KIA02].

For our analysis we need the following areas of the plane. Let $\mathcal{R}_{\text{hor}} = \{\text{BBox}(p,q) \mid \{p,q\} \in Z_{\text{hor}}\}$, where $\text{BBox}(p,q)$ is the smallest axis-parallel closed rectangle that contains p and q. Note that $\text{BBox}(p,q)$ is just the line segment $\text{Seg}[p,q]$ from p to q, if p and q lie on the same horizontal or vertical line. In this case we call $\text{BBox}(p,q)$ a *degenerate rectangle*. Define $\mathcal{R}_{\text{ver}}$ and $\mathcal{R}_{\text{quad}}$ analogously. Let $\mathcal{A}_{\text{hor}}$, $\mathcal{A}_{\text{ver}}$, and $\mathcal{A}_{\text{quad}}$ be the subsets of the plane that are defined by the union of the rectangles in $\mathcal{R}_{\text{hor}}$, $\mathcal{R}_{\text{ver}}$, and $\mathcal{R}_{\text{quad}}$, respectively.

Any Manhattan network has to bridge the vertical (horizontal) gap between the points of each pair in Z_{ver} (Z_{hor}). Of course this can be done such that at the same time the gaps of adjacent pairs are (partly) bridged. The corresponding minimization problem is defined as follows:

Definition 1.3 (cover [KIA02]) *A set of vertical line segments $\mathcal{V}$ is a* cover *of (or* covers*) $\mathcal{R}_{\text{ver}}$, if any $R \in \mathcal{R}_{\text{ver}}$ is covered, i.e. for any horizontal line ℓ with $R \cap \ell \neq \emptyset$ there is a $V \in \mathcal{V}$ with $V \cap \ell \cap R \neq \emptyset$. We say that $\mathcal{V}$ is a* minimum vertical cover (MVC) *if $\mathcal{V}$ has minimum length among all covers of $\mathcal{R}_{\text{ver}}$. The definition of a* minimum horizontal cover (MHC) *is analogous.*

Figure 1.5 shows an example of an MVC. Since any MMN covers $\mathcal{R}_{\text{ver}}$ and $\mathcal{R}_{\text{hor}}$, Kato et al. have the following result.

Lemma 1.1 ([KIA02]) *The union of an* MVC *and an* MHC *has length bounded by the length of an* MMN*.*

To sketch our algorithm we need the following notations. Let N be a set of line segments. We say that N *satisfies* a set of point pairs S if N contains a Manhattan p–q path for each $\{p,q\} \in S$. We use $\bigcup N$ to denote the corresponding set of points, i.e. the union of the line segments in N. Let ∂M be the boundary of a set $M \subseteq \mathcal{R}^2$.

Our algorithm will proceed in four phases. In phase 0, we compute Z. In phase I, we construct a network N_1 that contains the union of a special MVC and a special MHC and

satisfies $Z_{\text{ver}} \cup Z_{\text{hor}}$. In phase II, we identify a set $\mathcal{R}$ of open regions in $\mathcal{A}_{\text{quad}}$ that do not intersect N_1, but need to be bridged in order to satisfy Z_{quad}. The regions in $\mathcal{R}$ are staircase polygons. They give rise to two sets of segments, N_2 and N_3, which are needed to satisfy Z_{quad}. For each region $A \in \mathcal{R}$ we put the segments that form $\partial A \setminus \bigcup N_1$ into N_2, plus, if necessary, an extra segment to connect ∂A to N_1. Finally, in phase III, we bridge the regions in $\mathcal{R}$ by computing a set N_3 of segments in the interior of the regions. This yields a Manhattan network $N = N_1 \cup N_2 \cup N_3$.

The novelty of our analysis is that we partition the plane into two areas and compare N to an MMN in each area separately. The area $\mathcal{A}_3$ consists of the interiors of the regions $A \in \mathcal{R}$ and contains N_3. The other area $\mathcal{A}_{12}$ is the complement of $\mathcal{A}_3$ and contains $N_1 \cup N_2$. For a fixed MMN N_{opt} we show that $|N \cap \mathcal{A}_{12}| \leq 3|N_{\text{opt}} \cap \mathcal{A}_{12}|$ and $|N \cap \mathcal{A}_3| \leq 2|N_{\text{opt}} \cap \mathcal{A}_3|$, and thus $|N| \leqslant 3|N_{\text{opt}}|$. The details will be given in Section 1.5.

1.3 Neighbors and the generating set

We now define vertical and horizontal neighbors of points in P. Knowing these neighbors helps to compute Z and $\mathcal{R}$.

Definition 1.4 (neighbors) *For a point $p \in P$ and $t \in \{1,2,3,4\}$ let* $p.\text{xnbor}[t] = \text{nil}$ *if $Q(p,t) \cap P = \{p\}$. Otherwise* $p.\text{xnbor}[t]$ *points at the point that has minimum y-distance from p among all points in $Q(p,t) \cap P \setminus \{p\}$ with minimum x-distance from p. The pointer* $p.\text{ynbor}[t]$ *is defined by exchanging x and y in the above definition.*

All pointers of types xnbor and ynbor can be computed by a simple plane sweep in $O(n \log n)$ time. The set Z_{ver} is then determined by going through the points in lexicographical order and examining the pointers of type xnbor. This works analogously for Z_{hor}. Note that by Definition 1.1 each point $q \in P$ is incident to at most three rectangles of $\mathcal{R}_{\text{ver}}$, at most two of which can be (non-) degenerate. We refer to points $p \in P$ with $(p,q) \in Z_{\text{ver}}$ as *vertical predecessors* of q and to points $r \in P$ with $(q,r) \in Z_{\text{ver}}$ as *vertical successors* of q. We call a predecessor or successor of q *degenerate* if it has the same x-coordinate as q. Note that each point can have at most one degenerate vertical predecessor and successor, and at most one non-degenerate vertical predecessor and successor. Horizontal predecessors and successors are defined analogously with respect to Z_{hor}. For each $t \in \{1,2,3,4\}$ the pair $(q, q.\text{xnbor}[t])$ lies in Z_{quad} if and only if $q.\text{xnbor}[t] \neq \text{nil}$ and no vertical or horizontal predecessor or successor lies in $Q(q,t)$. We conclude:

Lemma 1.2 *All pointers of type* xnbor *and* ynbor*, and the generating set Z can be computed in $O(n \log n)$ time.*

1.4 Minimum covers

In general the union of an MVC and an MHC does not satisfy $Z_{\text{ver}} \cup Z_{\text{hor}}$. Additional segments must be added to achieve this. To ensure that the total length of these segments can be bounded, we need covers with a special property. We say that a cover is *nice* if each cover segment contains an input point.

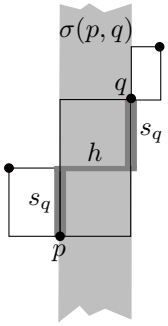


Figure 1.1: Illustration for Lemma 1.3.

Lemma 1.3 *For any nice* MVC $\mathcal{V}$ *and any nice* MHC $\mathcal{H}$ *there is a set* $\mathcal{S}$ *of line segments such that* $\mathcal{V} \cup H \cup S$ *satisfies* $Z_{\text{ver}} \cup Z_{\text{hor}}$ *and* $|\mathcal{S}| \leq W + H$*, where* W *and* H *denote width and height of* $\text{BBox}(P)$*, respectively. We can compute the set* $\mathcal{S}$ *in linear time if for each* $R \in \mathcal{R}_{\text{ver}}$ *(*$\mathcal{R}_{\text{hor}}$*) we have constant-time access to the segments in* $\mathcal{V}$ *(*$\mathcal{H}$*) that intersect* R*.*

Proof. We show that there is a set $\mathcal{S}_V$ of horizontal segments with $|\mathcal{S}_V| \leq W$ such that $\mathcal{V} \cup S_V$ satisfies Z_{ver}. Analogously it can be shown that there is a set $\mathcal{S}_H$ of vertical segments with $|\mathcal{S}_H| \leq H$ such that $\mathcal{H} \cup S_H$ satisfies Z_{hor}. This proves the lemma.

Let $(p, q) \in Z_{\text{ver}}$. If $R = \text{BBox}(p, q)$ is degenerate, then by the definition of a cover, there is a line segment $s \in \mathcal{V}$ with $R \subseteq s$, and thus $\mathcal{V}$ satisfies (p, q).

Otherwise R defines a non-empty vertical open strip $\sigma(p, q)$ bounded by p and q. Note that by the definition of Z_{ver}, R is the only rectangle in $\mathcal{R}_{\text{ver}}$ that intersects $\sigma(p, q)$. This yields that the widths of $\sigma(p, q)$ over all $(p, q) \in Z_{\text{ver}}$ sum up to at most W. Thus we are done, if we can show that there is a horizontal line segment h such that the length of h equals the width of $\sigma(p, q)$ and $\mathcal{V} \cup \{h\}$ satisfies (p, q).

Now observe that no line segment in $\mathcal{V}$ intersects $\sigma(p, q)$ since $\mathcal{V}$ is nice and $\sigma(p, q) \cap P = \emptyset$. Hence, the segments of $\mathcal{V}$ that intersect R in fact intersect only the vertical edges of R. We assume w.l.o.g. that $x_p < x_q$ and $y_p < y_q$ (otherwise rename and/or mirror P at the x-axis). This means that due to the definition of Z_{ver}, there is no input point vertically above p. Thus, if there is a segment s_p in $\mathcal{V}$ that intersects the left edge of R, then s_p must contain p. Analogously, a segment s_q in $\mathcal{V}$ that intersects the right edge of R must contain q. Since $\mathcal{V}$ covers R, s_p or s_q must exist. Let ℓ be the horizontal through the topmost point of s_p or the bottommost point of s_q. Then $h = \ell \cap R$ does the job, again due to the fact that $\mathcal{V}$ covers R, see Figure 1.1. Clearly h can be determined in constant time. ♣

In order to see that every point set has in fact a nice MVC, we need the following definitions. We restrict ourselves to the vertical case, the horizontal case is analogous.

For a horizontal line ℓ consider the graph $G_\ell(V_\ell, E_\ell)$, where V_ℓ is the intersection of ℓ with the vertical edges of rectangles in $\mathcal{R}_{\text{ver}}$, and there is an edge in E_ℓ if two intersection points belong to the same rectangle. We say that a point v in V_ℓ is *odd* if v is contained in a degenerate rectangle or if the number of points to the left of v that belong to the same connected component of G_ℓ is odd, otherwise we say that v is *even*. For a vertical line g let an *odd segment* be an inclusion-maximal connected set of odd points on g. Define *even segments* accordingly. For example, the segment s (drawn bold in Figure 1.6) is an even segment, while $f \setminus s$ is odd. We say that *parity changes* in points where two segments of different parity touch. We refer to these points as *points of changing parity*. The MVC with

the desired property will simply be the set of all odd segments. The next lemma characterizes odd segments, especially item (v) prepares their computation. Strictly speaking we have to state whether the endpoints of each odd segment are odd too, but since a closed segment has same length as the corresponding open segment, we consider odd segments closed.

Lemma 1.4 *Let $g : x = x_g$ be a vertical line through some point $p = (x_p, y_p) \in P$, meaning that $x_p = x_g$. It holds that:*

(i) Let e be a vertical edge of a rectangle $R \in \mathcal{R}_{\text{ver}}$. Then either all points on e are even or the only inclusion-maximal connected set of odd points on e contains an input point.

(ii) Let $R_1, \ldots, R_d$ and $R'_1, \ldots, R'_{d'}$ be the degenerate and non-degenerate rectangles in $\mathcal{R}_{\text{ver}}$ that g intersects, respectively. Then $d = |g \cap P| - 1$ and $d' \leq 2$. If $d = 0$ then $d' > 0$ and each R'_i has a corner in p. Else, if $d > 0$, there are $p_1, p_2 \in P$ such that $g \cap (R_1 \cup \cdots \cup R_d) = \text{Seg}[p_1, p_2]$. Then each R'_i has a corner in either p_1 or p_2.

(iii) There are $b_g < t_g \in \mathcal{R}$ such that $g \cap \mathcal{A}_{\text{ver}} = \{x_g\} \times [b_g, t_g]$.

(iv) The line g contains at most two points of changing parity and at most one odd segment. For each point c of changing parity there is an input point with the same y-coordinate.

(v) If g has no point of changing parity, there is either no odd segment on g or the odd segment is $\{x_g\} \times [b_g, t_g]$. If g has one point c of changing parity, then either $\{x_g\} \times [b_g, y_c]$ or $\{x_g\} \times [y_c, t_g]$ is the odd segment. If g has two points c and c' of changing parity, then $\{x_g\} \times [y_c, y_{c'}]$ is the odd segment.

Proof. For (i) we assume without loss of generality that e is the right vertical edge of $R = \text{BBox}(p, q)$ and that q is the topmost point of e. If R is degenerate it is clear that all points on e (including p and q) are odd, and we are done. Thus we can assume that $x_p < x_q$. Let $p_0 = q, p_1 = p, p_2 \ldots, p_k$ be the input points in order of decreasing x-coordinate that span the rectangles in $\mathcal{R}_{\text{ver}}$ that are relevant for the parity of e. Let $p_i = (x_i, y_i)$. For $2 \leqslant i \leqslant k$ define recursively $\overline{y_i} = \min\{y_i, \overline{y_{i-2}}\}$ if i is even, and $\overline{y_i} = \max\{y_i, \overline{y_{i-2}}\}$ if i is odd. Let $\overline{p_i} = (x_i, \overline{y_i})$, and let $\overline{\mathcal{L}}$ be the polygonal chain through $p_0, p_1, \overline{p_2}, \overline{p_3}, \ldots, \overline{p_k}$, see Figure 1.6. Note that the parity of a point v on e is determined by the number of segments of $\overline{\mathcal{L}}$ that the horizontal h_v through v intersects. If h_v is below $\overline{p_k}$, then it intersects a descending segment for each ascending segment of $\overline{\mathcal{L}}$, hence v is even. If on the other hand h_v is above $\overline{p_k}$, then it intersects an ascending segment for each descending segment—plus $\overline{p_1 p_0}$, hence v is odd. In other words, if $\overline{y_k} = y_0$, all points of e are even, if $\overline{y_k} = y_1$, all points of e are odd, and otherwise parity changes only in $(x_0, \overline{y_k})$ and q is odd. This settles (i).

(ii) follows directly from the definition of Z_{ver}, and (iii) follows from (ii), see also Figure 1.2.

For (iv) we first assume $d = 0$. Then (ii) yields $d' \in \{1, 2\}$ and $g \cap P = \{p\}$. By (i) we know that the only inclusion-maximal connected set of odd points on each vertical rectangle edge on g contains an input point, i.e. p. Thus there are at most two points of changing parity and there is at most one odd segment on g. Also according to the above proof of (ii), parity can change only in points of type $(x_0, \overline{y_k})$, and $\overline{y_k}$ is the y-coordinate of some input point in the set $\{p_0, \ldots, p_k\}$.

Now if $d > 0$ note that all degenerate rectangles consist only of odd points. By (ii) we have that $g \cap (R_1 \cup \cdots \cup R_d) = \text{Seg}[p_1, p_2]$ and that each of the at most two non-degenerate rectangles has a corner in either p_1 or p_2. Thus again the statement holds.

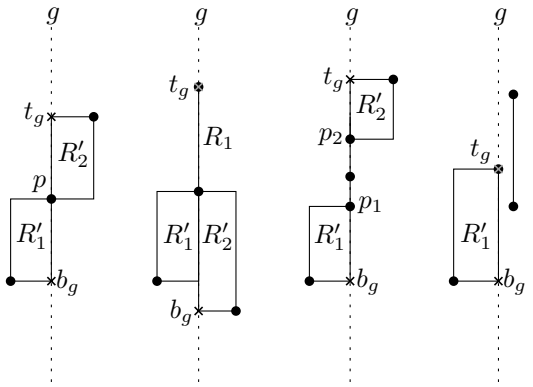


Figure 1.2: Illustration for Lemma 1.4 (ii) and (iii).

For the proof of (v) we make a case distinction depending on d'. If $d' = 0$, g intersects only degenerate rectangles and thus there is no point of changing parity on g and the odd segment is $\{x_g\} \times [b_g, t_g]$. Otherwise we assume w.l.o.g. that e is contained in g. If $e = \{x_g\} \times [b_g, t_g]$ holds, we are done. The argument of (i) shows that either e contains no point of changing parity and hence all points of e are of one parity, or $c = c_1$ is the only point of changing parity and the odd segment is $\{x_g\} \times [c_1, t_g = y_p]$. If $e \neq \{x_g\} \times [b_g, t_g]$, there is a further rectangle R_p in $\mathcal{R}_{\text{ver}}$ with $R_p = \text{BBox}(p, r)$ and $x_p \leqslant x_r, y_p < y_r$. If R_p is non-degenerate all points on $\{x_g\} \times [b_g, t_g] \setminus e$ are even, as there are no relevant rectangles to the right. In this case we have no odd segment on g if e is completely even, the odd segment is $\{x_g\} \times [b_g, y_p = c_1]$ if e is completely odd, and if c is a point of changing parity the odd segment is $\{x_g\} \times [c = c_1, y_p = c_2]$. If R_p is degenerate, $\{x_g\} \times [p, r]$ has to be added to the odd segments stated as before, besides the same argument holds with a possibly rectangle R_r connected to r. ♣

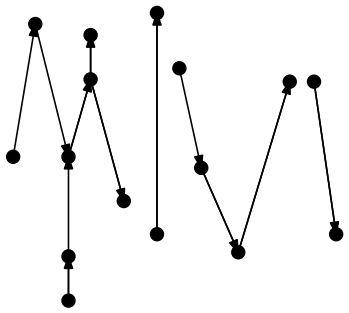


Figure 1.3: Point pairs in Z_{ver}.

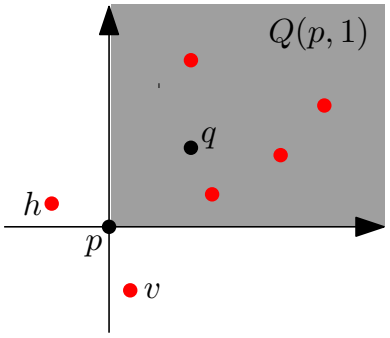


Figure 1.4: The pair (p, q) is in Z_{quad}.

Lemma 1.5 *The set $\mathcal{V}$ of all odd segments is a nice* MVC*, the* odd MVC*.*

Proof. Clearly $\mathcal{V}$ covers $\mathcal{R}_{\text{ver}}$. Let ℓ be a horizontal line that intersects $\mathcal{A}_{\text{ver}}$. Consider a connected component C of G_ℓ and let k be the number of vertices in C. If k is even then any cover must contain at least $k/2$ vertices of C, and $\mathcal{V}$ contains exactly $k/2$. On the other hand,

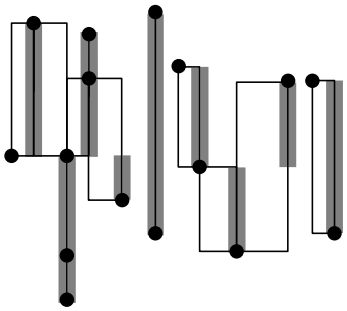


Figure 1.5: The odd MVC.

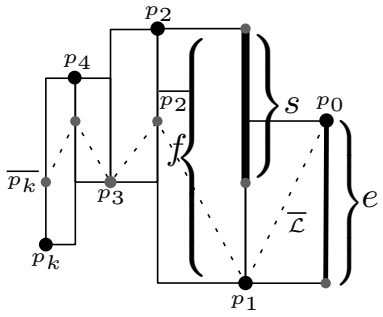


Figure 1.6: Proof of Lemma 1.4.

if $k > 1$ is odd then any cover must contain at least $(k-1)/2$ vertices of C, and $\mathcal{V}$ contains exactly $(k-1)/2$. If $k = 1$, any cover must contain the vertex, and so does $\mathcal{V}$ as the vertex belongs to a degenerate rectangle. Thus $\mathcal{V}$ is an MVC. Lemma 1.4 (i) shows that $\mathcal{V}$ is nice. ♣

Lemma 1.6 *The odd* MVC *can be computed in* $O(n \log n)$ *time using linear space.*

Proof. We compute the odd MVC by a plane sweep. Let $x^1 < \cdots < x^u$ be the ascending sequence of all distinct x-coordinates. For each vertical line $g_i : x = x^i$ we determine in a preprocessing step the points b_i and t_i such that $g_i \cap \mathcal{A}_{\text{ver}} = [b_i, t_i]$. For this it suffices to go through the input points in lexicographical order. For each g_i we introduce numbers β_i and τ_i which we initially set to ∞. After the sweep β_i and τ_i will determine the odd MVC in the following way: If $\beta_i = \tau_i = \infty$, then there is no odd segment on g_i, otherwise g_i contains the odd segment $x^i \times [\beta_i, \tau_i]$. These two variables are sufficient since according to Lemma 1.4(iv) there is at most one odd segment on g_i.

We use a sweep-line algorithm to compute the values β_i and τ_i. As usual, our sweep-line algorithm is supported by two data structures, the event-point queue and the sweep-line status. According to Lemma 1.4,(iv) there is an input point r with $y_c = y_r$ for each point c of changing parity and according to (v) we have to determine these points in order to get the odd segments. Thus, the event-point queue can be implemented as a sorted list of all y-coordinates of the input points. Note that the same y-value can occur more than once. This ensures that at each event point only one event takes place. The sweep-line status is a balanced binary tree in which each node corresponds to a connected components of G_ℓ, where ℓ is the current position of the horizontal sweep line. Our sweep line ℓ is a horizontal line sweeping all rectangles in $\mathcal{R}_{\text{ver}}$ from bottom to top.

While the sweep line moves from one event point to the next, the sweep-line status maintains the connected components of G_ℓ in a balanced binary tree $\mathcal{T}$. Initially $\mathcal{T}$ is empty. Whenever ℓ reaches an event point, we update $\mathcal{T}$. For each component C of G_ℓ we store two indices l_C and r_C with the property that the leftmost node of C lies on g_{l_C} and the rightmost node on g_{r_C}. The tree $\mathcal{T}$ is organized such that $r_{C'} < l_C$ for two components of G_ℓ if C' is a left child of C, while $r_C < l_{C'}$ if C' is a right child.

The following component modifications can occur on an event: a component appears or disappears, one component is replaced by a new one, a component is enlarged or reduced, two or three components are joined or a component is split into two or three components.

We can decide in constant time which type of event takes place, simply by evaluating b_i, t_i, and—if they exist—$b_{i\pm1}$ and $t_{i\pm1}$, where i is the index of the line which contains the input point that caused the current event. For each event we have to change the entries of at most three components and update $\mathcal{T}$ accordingly.

Each of these update operations takes $O(\log n)$ time. For example if a component is split into two, this component has to be found, its entries have to be updated and a new component has to be created and inserted to $\mathcal{T}$.

The correct values β_i, τ_i for each line g_i are computed during the sweep. At any point of time, the values β_i and τ_i indicate the information about the odd segment on g_i detected so far: $\beta_i = \infty$ means no odd segment has been found yet, while $\beta_i \neq \infty$ says that there is an odd segment on g_i with lower endpoint (x^i, β_i). If additionally $\tau_i \neq \infty$ then the upper endpoint has also been detected yet and the odd segment on g_i is $x^i \times [\beta_i, \tau_i]$. Thus, at each event we have to check whether there are odd segments that start or end at y_ℓ, the current y-value of the sweep line ℓ. According to Lemma 1.4(v), points of changing parity are always endpoints of odd segments, while bottom- or topmost points of $\mathcal{A}_{\text{ver}} \cap g_i$ may be endpoints. In order to find all endpoints, we have to consider the old and new entries of changing components whenever $\mathcal{T}$ is updated. Bottommost points occur, if a new components appears, a component is enlarged or components are joined. Topmost points occur, if a component disappears, is reduced or components are split. Points of changing parity can occur if the extent of a component changes, components are joined or split or one component is replaced by a new one. If we have found a bottommost point b_i, we check whether b_i is odd and hence the lower endpoint of the odd segment on g_i is b_i. We do this by examining l_C, r_C and i, where C is the component that contains b_i. If $l_C = r_C$ (degenerate rectangle) or the parities of l_C and i are different, b_i is odd and we set $\beta_i = b_i$. If we discover a point of changing parity, we check whether it is the lower or upper endpoint of the odd segment on g_i. If β_i is still ∞ the point of changing parity is the lower endpoint, otherwise the upper. We set accordingly $\beta_i = y_\ell$ or $\tau_i = y_\ell$. At a topmost point t_i we only have to check whether there is an odd segment on g_i and whether t_i is the upper endpoint of the odd segment. This is the case if $\beta_i \neq \infty$ and $\tau_i = \infty$, we then set $\tau_i = t_i$.

As there are at most $3n$ operations that change components during the sweep, we have to handle $O(n)$ of these checks. After sorting, each of the n events of our sweep takes $O(\log n)$ time. Thus, the total running time is $O(n \log n)$. ♣

The odd MHC can be computed analogously.

1.5 An approximation algorithm

Our algorithm ApproxMMN proceeds in four phases, see Figure 1.10. In phase 0 we compute all pointers of type xnbor and ynbor and the set Z. In phase I we satisfy all pairs in $Z_{\text{ver}} \cup Z_{\text{hor}}$ by computing the network N_1, the union of a nice MVC $\mathcal{C}_{\text{ver}}$, a nice MHC $\mathcal{C}_{\text{hor}}$, and at most one additional line segment for each rectangle in $\mathcal{R}_{\text{ver}} \cup \mathcal{R}_{\text{hor}}$. In phase II we compute the staircase polygons that were mentioned in Section 1.2. The union of their interiors is area $\mathcal{A}_3$. Network N_2 consists of the boundaries of these polygons and segments that connect the boundaries to N_1. In phase III we compute a network N_3 of segments in $\mathcal{A}_3$. The resulting network $N_1 \cup N_2 \cup N_3$ satisfies Z.

Phase 0. In phase 0 we compute all pointers of types xnbor and ynbor, and the set Z. We organize our data structures such that from now on we have constant-time access to all relevant information such as xnbor, ynbor, vertical and horizontal predecessors and successors from each point $p \in P$.

Phase I. First we compute the nice odd MVC and the nice odd MHC, denoted by $\mathcal{C}_{\text{ver}}$ and $\mathcal{C}_{\text{hor}}$, respectively. Then we compute the set $\mathcal{S}$ of additional segments according to Lemma 1.3. We compute $\mathcal{C}_{\text{ver}}$, $\mathcal{C}_{\text{hor}}$ and $\mathcal{S}$ such that from each point $p \in P$ we have constant-time access to the at most two *cover segments* (i.e. segments in $\mathcal{C}_{\text{ver}} \cup \mathcal{C}_{\text{hor}}$) that contain p and to the additional segments in the at most four rectangles incident to p.

Lemmas 1.1, 1.3, and 1.6 show that $N_1 = \mathcal{C}_{\text{ver}} \cup \mathcal{C}_{\text{hor}} \cup \mathcal{S}$ can be computed in $O(n \log n)$ time and that $|N_1| \leqslant |N_{\text{opt}}| + H + W$ holds. Recall that N_{opt} is a fixed MMN.

Phase II. In general N_1 does not satisfy Z_{quad}; further segments are needed. In order to be able to bound the length of these new segments, we partition the plane into two areas $\mathcal{A}_{12}$ and $\mathcal{A}_3$ as indicated in Section 1.2. We wanted to define $\mathcal{A}_3$ such that $|N_{\text{opt}} \cap \mathcal{A}_3|$ were large enough for us to bound the length of the new segments. However, we were not able to define $\mathcal{A}_3$ such that we could at the same time (a) satisfy Z_{quad} by adding new segments exclusively in $\mathcal{A}_3$ and (b) bound their length. Therefore we put the new segments into two disjoint sets, N_2 and N_3, such that $N_1 \cup N_2 \subseteq \mathcal{A}_{12}$ and $N_3 \subseteq \mathcal{A}_3$. This enabled us to bound $|N_1 \cup N_2|$ by $3|N_{\text{opt}} \cap \mathcal{A}_{12}|$ and $|N_3|$ by $2|N_{\text{opt}} \cap \mathcal{A}_3|$.

We now prepare our definition of $\mathcal{A}_3$. Recall that $Q(q,1), \ldots, Q(q,4)$ are the four quadrants of the Cartesian coordinate system with origin q. Let $P(q,t) = \{p \in P \cap Q(q,t) \mid (p,q) \in Z_{\text{quad}}\}$ for $t = 1,2,3,4$. For example, in Figure 1.12, $P(q,1) = \{p_1, \ldots, p_5\}$. Due to the definition of Z_{quad} we have $Q(p,t) \cap P(q,t) = \{p\}$ for each $p \in P(q,t)$. Thus the area $\mathcal{A}_{\text{quad}}(q,t) = \bigcup_{p \in P(q,t)} \text{BBox}(p,q)$ is a staircase polygon. The points in $P(q,t)$ are the "stairs" of the polygon and q is the corner opposite the stairs. In Figure 1.12, $\mathcal{A}_{\text{quad}}(q,1)$ is the union of the shaded areas. In order to arrive at a definition of the area $\mathcal{A}_3$, we will start from polygons of type $\mathcal{A}_{\text{quad}}(q,t)$ and then subtract areas that can contain segments of N_1 or are not needed to satisfy Z_{quad}.

Let $\Delta(q,t) = \text{int}\big(\mathcal{A}_{\text{quad}}(q,t) \setminus (\mathcal{A}_{\text{hor}} \cup \mathcal{A}_{\text{ver}})\big)$, where $\text{int}(M)$ denotes the interior of a set $M \subseteq \mathcal{R}^2$. In Figure 1.12, $\Delta(q,1)$ is the union of the three areas with dotted boundary. Let $\delta(q,t)$ be the union of those connected components A of $\Delta(q,t)$, such that $\partial A \cap P(q,t) \neq \emptyset$. In Figure 1.12, $\delta(q,1)$ is the union of the two dark shaded areas A and $\overline{A}$.

Due to the way we derived $\delta(q,t)$ from $\mathcal{A}_{\text{quad}}(q,t)$, it is clear that each connected component A of $\delta(q,t)$ is a staircase polygon, too. The stairs of A correspond to the input points on ∂A, i.e. $P(q,t) \cap \partial A$. Let q_A be the point on ∂A that is closest to q. This is the corner of A opposite the stairs. The next lemma is very technical, but it is crucial for the estimation of our network within the $\delta(q,t)$ regions.

Lemma 1.7 *Areas of type $\delta(q,t)$ are pairwise disjoint.*

Proof. For each pair $(p,q) \in Z_{\text{quad}}$ we define its *forbidden area* F_{pq} to be the union of $\text{BBox}(p,q)$ and the intersection of (a) the halfplane not containing p that is bounded by the horizontal through q and (b) the open strip between the verticals through p and q, see Figure 1.7. We have $F_{pq} \cap (P \setminus \{p,q\}) = \emptyset$ since the existence of a point $r \in F_{pq} \cap (P \setminus \{p,q\})$ would contradict $(p,q) \in Z_{\text{quad}}$.

Suppose there is a point $s \in \delta(q,t) \cap \delta(q',t')$ with $(q,t) \neq (q',t')$. Clearly $q \neq q'$ since $\delta(q,t) \subset \text{int}(Q(q,t))$ and $\delta(q,t') \subset \text{int}(Q(q,t'))$ and $\text{int}(Q(q,t)) \cap \text{int}(Q(q,t')) = \emptyset$ for $t \neq t'$. Since $\delta(q,t), \delta(q',t') \subseteq \mathcal{A}_{\text{quad}}$ we know that there are points p and p' with $(p,q),(p',q') \in Z_{\text{quad}}$ such that $s \in \text{BBox}(p,q) \cap \text{BBox}(p',q')$. Let $B = \text{BBox}(p,q)$ and $B' = \text{BBox}(p',q')$. Without loss of generality, we assume that p is to the right and above q.

We know that $p', q' \notin B$ since $B \subset F_{pq}$. Analogously $p, q \notin B'$. Let $\ell(x_q, y_p)$ and $r(x_p, y_q)$ be the other two corners of B, see Figure 1.8. There are three cases:

Case I: $B' \cap \{\ell, r\} = \emptyset$.

Recall that $B' \cap \{p,q\} = \emptyset$ and that $B \cap B' \neq \emptyset$. Thus B' lies in the vertically unbounded open strip $S_1 = (x_q, x_p) \times (-\infty, \infty)$ or in the horizontally unbounded open strip $S_2 = (-\infty, \infty) \times (y_q, y_p)$ determined by two opposite edges of B, see Figure 1.8. (Note that p' and q' cannot lie on the boundary of S_1 or S_2, otherwise (p,q) or (p',q') would not be in Z_{quad}.) Now if $B' \subset S_1$ (see the dashed rectangle in Figure 1.8), then p' or q' lies in F_{pq} contradicting $(p,q) \in Z_{\text{quad}}$. If on the other hand $B' \subset S_2$ (see the dotted rectangle in Figure 1.8) then p or q lies in $F_{p'q'}$ contradicting $(p',q') \in Z_{\text{quad}}$.

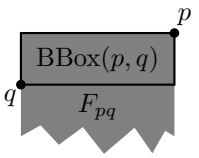


Figure 1.7: The forbidden area F_{pq} is shaded.

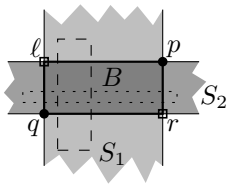


Figure 1.8: Case I.

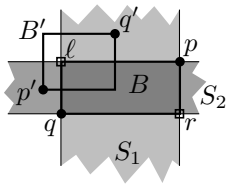


Figure 1.9: Case III.

Case II: $B' \cap \{\ell, r\} = \{r\}$.

Now the upper left corner of B' lies in B since again $B' \cap \{p,q\} = \emptyset$. Thus the lower left corner of B' is an input point (p' or q') but lies in F_{pq} contradicting $(p,q) \in Z_{\text{quad}}$.

Case III: $B' \cap \{\ell, r\} = \{\ell\}$.

In this case the lower right corner of B' lies in B and the upper right corner of B' lies above B in S_2. If p' was the upper right corner of B', we would have $q \in F_{p'q'}$, which contradicts $(p',q') \in Z_{\text{quad}}$. Thus p' lies in S_2 to the left of B and q' in S_1 above B, see Figure 1.9. Such a constellation is indeed possible. Note, however, that $B \cap B' \subset \text{BBox}(q,q')$. Furthermore $\{q,q'\} \in Z_{\text{ver}}$ since $\text{BBox}(q,q')$ and the open strip bounded by the verticals through q and q' are completely contained in $F_{pq} \cup F_{p'q'}$ and thus do not contain any input points except q and q'. These observations yield $s \in B \cap B' \subset \text{BBox}(q,q') \subset \mathcal{A}_{\text{ver}}$, which contradicts $s \in \delta(q,t)$ since $\delta(q,t)$ is contained in the complement of $\mathcal{A}_{\text{ver}}$.

♣

We are now sure that we can treat each connected component A of $\delta(q,t)$ independently. Finally we define $\mathcal{A}_3 = \bigcup_{t \in \{1,2,3,4\}} \bigcup_{q \in P} \delta(q,t)$ and $\mathcal{A}_{12} = \mathcal{R}^2 \setminus \mathcal{A}_3$. This definition ensures that $N_1 \subset \mathcal{A}_{12}$ as desired. The set N_2 will be constructed as follows: for each connected component A of $\mathcal{A}_3$, we put $\partial A \setminus \bigcup N_1$ into N_2 and test whether N_1 contains a Manhattan

path from q_A to q. If not, we add a further segment to N_2. This segment lies in $\mathcal{A}_{\text{hor}}$ and will be defined below. Since $\mathcal{A}_{\text{hor}}$ as well as ∂A are contained in $\mathcal{A}_{12}$, we have $N_2 \subset \mathcal{A}_{12}$. The set N_3 will be defined in phase III and will be arranged such that $N_3 \subset \mathcal{A}_3$.

We now describe how to compute $P(q,t)$ and how to find the connected components of $\delta(q,t)$. We compute all sets $P(q,t)$ by going through the input points and checking their Z_{quad}-partners. This takes linear time since $|Z_{\text{quad}}| = O(n)$. We sort the points in each set $P(q,t)$ according to their x-distance from q. This takes $O(n \log n)$ total time. The remaining difficulty is to decide which points in $P(q,t)$ are incident to the same connected component of $\delta(q,t)$. In Figure 1.12, $\{p_1, p_2\} \subset \partial A$ and $\{p_3, p_4, p_5\} \subset \partial\overline{A}$. For our description how to figure this out we assume $t = 1$ and $P(q,1) = (p_1, \ldots, p_m)$. Note that each connected component of $\delta(q,1)$ corresponds to a sequence of consecutive points in $P(q,1)$. By definition, for each connected component A of $\delta(q,1)$ and all $p_i, p_j \in A$ we have $p_i.\text{ynbor}[3] = p_j.\text{ynbor}[3]$.

We detect these sequences by going through $p_1, \ldots, p_m$. Let p_i be the current point and let A be the current connected component. If and only if $p_i.\text{ynbor}[3] \neq p_{i+1}.\text{ynbor}[3]$ there is a rectangle $R_A \in \mathcal{R}_{\text{hor}}$ that separates A from the next connected component of $\delta(q,1)$. The rectangle R_A is defined by the point $v_A = p_i.\text{ynbor}[3]$ and its horizontal successor w_A, which in this case is unique, see Figure 1.12. It remains to specify the coordinates of the corner point q_A of A. Let p_0 be the (unique) vertical successor of q. Then $x_{q_A} = x_{p_0}$ and $y_{q_A} = y_{w_A}$.

At last, we want to make sure that $N_1 \cup N_2$ contains a Manhattan q–q_A path. The reason for this is that in phase III we will only compute Manhattan paths from each $p_i \in \partial A$ to q_A. Concatenating these paths with the q–q_A path yields Manhattan p_i–q paths since $q_A \in \text{BBox}(q, p_i)$. Note that segments in N_3 lie in $\mathcal{A}_3$ and thus cannot help to establish a q–q_A path within $\text{BBox}(q, q_A) \subset \mathcal{A}_{12}$.

The set N_1 contains a Manhattan q–p_0 path $\mathcal{P}_{\text{ver}}$ and a Manhattan v_A–w_A path $\mathcal{P}_{\text{hor}}$, since $(q, p_0) \in Z_{\text{ver}}$ and $(v_A, w_A) \in Z_{\text{hor}}$. If $q_A \in \mathcal{P}_{\text{ver}}$, then clearly N_1 contains a Manhattan q–q_A path. However, N_1 also contains a Manhattan q–q_A path if $q_A \in \mathcal{P}_{\text{hor}}$. This is due to the fact that $\mathcal{P}_{\text{ver}}$ and $\mathcal{P}_{\text{hor}}$ intersect. If $q_A \notin \mathcal{P}_{\text{ver}} \cup \mathcal{P}_{\text{hor}}$, then $\mathcal{P}_{\text{hor}}$ contains the point $c_A = (x_{q_A}, y_{v_A})$, which lies on the vertical through q_A on the opposite edge of R_A. Thus, to ensure a Manhattan q–q_A path in $N_1 \cup N_2$, it is enough to add the segment $s_A = \text{Seg}[q_A, c_A]$ to N_2. We refer to such segments as *connecting segments*.

The algorithm ApproxMMN does not compute $\mathcal{P}_{\text{ver}}$ and $\mathcal{P}_{\text{hor}}$ explicitly, but simply tests whether $q_A \notin \bigcup N_1$. This is equivalent to $q_A \notin \mathcal{P}_{\text{ver}} \cup \mathcal{P}_{\text{hor}}$ since our covers are minimum and the bounding boxes of $\mathcal{P}_{\text{ver}}$ and $\mathcal{P}_{\text{hor}}$ are the only rectangles in $\mathcal{R}_{\text{ver}} \cup \mathcal{R}_{\text{hor}}$ that contain s_A. Due to the same reason and to the fact that cover edges are always contained in (the union of) edges of rectangles in $\mathcal{R}_{\text{ver}} \cup \mathcal{R}_{\text{hor}}$, we have that $s_A \cap \bigcup N_1 = \{c_A\}$. This shows that connecting segments intersect N_1 at most in endpoints. The same holds for segments in N_2 that lie on $\partial\mathcal{A}_3$. This is important as later on, in Section 1.6 we need that a segment in N_1 and a segment in N_2 intersect at most in their endpoints. We summarize:

Lemma 1.8 *In $O(n \log n)$ time we can compute the set N_2, which has the following properties: (i) $N_2 \subset \mathcal{A}_{12}$, (ii) a segment in N_1 and a segment in N_2 intersect at most in their endpoints, and (iii) for each region $\delta(q,t)$ and each connected component A of $\delta(q,t)$, $N_1 \cup N_2$ contains ∂A and a Manhattan q–q_A path.*

Proof. The properties of N_2 follow from the description above. The runtime can be seen as follows. Let A be a connected component of $\mathcal{A}_3$ and $m_A = |P \cap \partial A|$. Note that $\sum m_A = O(n)$ since each point is adjacent to at most four connected components of $\mathcal{A}_3$, according

to Lemma 1.7. After sorting $P(q,t)$ we can compute in $O(m)$ time for each A the segment s_A and the set $\partial A \setminus \bigcup N_1$. This is due to the fact that we have constant-time access to each of the $O(m)$ rectangles in $\mathcal{R}_{\text{hor}} \cup \mathcal{R}_{\text{ver}}$ that intersect ∂A and to the $O(m)$ segments of N_1 that lie in these rectangles. ♣

Phase III. Now, we finally satisfy the pairs in Z_{quad}. Due to Lemma 1.8 for each connected component A of $\mathcal{A}_3$ it is enough to compute a set of segments $B(A)$ such that the union of $B(A)$ and ∂A contains Manhattan paths from any input point on ∂A to q_A. We say that such a set $B(A)$ *bridges* A. The set N_3 will be the union over all sets of type $B(A)$. The algorithm BRIDGE that we use to compute $B(A)$ is similar to the "thickest-first" greedy algorithm for rectangulating staircase polygons, see [GLN01]. However, we cannot use that algorithm since the segments that it computes do not lie entirely in $\mathcal{A}_3$.

For our description of algorithm BRIDGE we assume that A lies in a region of type $\delta(q,1)$. Let again $(p_1,\ldots,p_m)$ denote the sorted sequence of points on ∂A. Note that ∂A already contains Manhattan paths that connect p_1 and p_m to q_A. Thus we are done if $m \leq 2$. Otherwise let $p'_j = (x_{p_j}, y_{p_{j+1}})$, $a_j = \text{Seg}[(x_{q_A}, y_{p'_j}), p'_j]$ and $b_j = \text{Seg}[(x_{p'_j}, y_{q_A}), p'_j]$ for $j \in \{1,\ldots,m-1\}$, see Figure 1.13. We denote $|a_j|$ by α_j and $|b_j|$ by β_j. From now on we identify staircase polygon A with the tuple $(q_A, p_1, \ldots, p_m)$. Let B be the set of segments that algorithm BRIDGE computes. Initially is $B = \emptyset$. The algorithm chooses an $i \in \{1,\ldots,m-1\}$ and adds—if they exist—a_{i-1} and b_{i+1} to B. This satisfies $\big\{(p_i,q),(p_{i+1},q)\big\}$. In order to satisfy $\big\{(p_2,q),\ldots,(p_{i-1},q)\big\}$ and $\big\{(p_{i+2},q),\ldots,(p_{m-1},q)\big\}$, we solve the problem recursively for the two staircase polygons $\big((x_{q_A}, y_{p_i}), p_1, \ldots, p_{i-1}\big)$ and $\big((x_{p_{i+1}}, y_{q_A}), p_{i+2}, \ldots, p_m\big)$.

Our choice of i is as follows. Note that $\alpha_1 < \cdots < \alpha_{m-1}$ and $\beta_1 > \cdots > \beta_{m-1}$. Let $\Lambda = \{j \in \{1,\ldots,m-1\} \mid \alpha_j \leqslant \beta_j\}$. If $\Lambda = \emptyset$, we have $\alpha_1 > \beta_1$, i.e. A is flat and broad. In this case we choose $i = 1$, which means that only b_2 is put into B. Otherwise let $i' = \max \Lambda$. Now if $i' < m-1$ and $\alpha_{i'} \leqslant \beta_{i'+1}$, then let $i = i'+1$. In all other cases let $i = i'$. The idea behind this choice of i is that it yields a way to balance α_{i-1} and β_{i+1}, which in turn helps to compare $\alpha_{i-1}+\beta_{i+1}$ to $\min\{\alpha_i, \beta_i, \alpha_{i-1}+\beta_{i+1}\}$, i.e. the length of the segments needed by any Manhattan network in order to connect p_i and p_{i+1} to q, see also the proof of Theorem 1.9.

To avoid expensive updates of the α- and β-values of the staircase polygons in the recursion, we introduce offset values x_{off} and y_{off} that denote the x- respectively y-distance from the corner of the current staircase polygon to the corner q_A of A. In order to find the index i in a recursion, we compare $\alpha_j - x_{\text{off}}$ to $\beta_j - y_{\text{off}}$ instead of α_j to β_j as in the definition of Λ above. Figure 1.11 shows the pseudo code of algorithm BRIDGE for a staircase polygon A of type $\delta(q,1)$.

Running time and performance of algorithm BRIDGE(A) are as follows:

Lemma 1.9 *Given a connected component A of $\mathcal{A}_3$ with $|P \cap \partial A| = m$, algorithm* BRIDGE *computes in $O(m \log m)$ time a set B of line segments with $|B| \leqslant 2|N_{\text{opt}} \cap A|$ and $\bigcup B \subset A$ that bridges A.*

Proof. As for the running time, note that the monotone orders of $\alpha_1,\ldots,\alpha_{m-1}$ and $\beta_1,\ldots,\beta_{m-1}$ permit to find i by binary search in $O(\log m)$ time. The recursion tree has $O(m)$ nodes. Thus the algorithm runs in $O(m \log m)$ time.

As for the performance, note that according to Lemma 1.7, A does not intersect any other connected component of $\mathcal{A}_3$. The performance proof is similar to the analysis of the greedy algorithm for rectangulation, see Theorem 10 in [GLN01].

ApproxMMN(P)

Phase 0: *Neighbors and generating set*
for each $p \in P$ **and** $t \in \{1,2,3,4\}$ **do**
  compute p.xnbor[t] and p.ynbor[t]
compute $Z = Z_{\text{ver}} \cup Z_{\text{hor}} \cup Z_{\text{quad}}$.

Phase I: *Compute* N_1

compute odd MVC $\mathcal{C}_{\text{ver}}$ and MHC $\mathcal{C}_{\text{hor}}$
compute set $\mathcal{S}$ of additional segments
$N_1 \leftarrow \mathcal{C}_{\text{ver}} \cup \mathcal{C}_{\text{hor}} \cup \mathcal{S}$, $N_2 \leftarrow \emptyset$, $N_3 \leftarrow \emptyset$

Phase II: *Compute* N_2

compute $\mathcal{A}_3$
for each connected component A of $\mathcal{A}_3$ **do**
  $N_2 \leftarrow N_2 \cup (\partial A \setminus \bigcup N_1)$
  if $q_A \notin \bigcup N_1$ **then**
    $N_2 \leftarrow N_2 \cup \{s_A\}$

Phase III: *Compute* N_3

for each connected component A of $\mathcal{A}_3$ **do**
  $N_3 \leftarrow N_3 \cup$ Bridge(A)

return $N = N_1 \cup N_2 \cup N_3$

Figure 1.10: ApproxMMN(P)

Bridge$\big(A = (q_A, p_1, \ldots, p_m)\big)$

for $i = 1$ **to** $m-1$ **do**
  compute α_i and β_i
return SubBridge$\big(1, m, 0, 0\big)$

SubBridge$\big(k, l, x_{\text{off}}, y_{\text{off}}\big)$

$A_{\text{curr}} = (q_A + (x_{\text{off}}, y_{\text{off}}), p_k, \ldots, p_l)$
if $l - k < 2$ **return** $\emptyset$
$\Lambda = \big\{j \in \{k, \ldots, l-1\} : \ \alpha_j - x_{\text{off}} \leqslant \beta_j - y_{\text{off}}\big\}$
$i = \max \Lambda \cup \{k\}$
if $i < l-1$ **and** $\alpha_i - x_{\text{off}} \leqslant \beta_{i+1} - y_{\text{off}}$
  then $i = i + 1$
$B = \emptyset$
if $i > 1$ **then**
  $B = B \cup \{a_{i-1} \cap A_{\text{curr}}\}$
if $i < l-1$ **then**
  $B = B \cup \{b_{i+1} \cap A_{\text{curr}}\}$
$x_{\text{new}} = x_{p_{i+1}} - x_{q_A}$
$y_{\text{new}} = y_{p_i} - y_{q_A}$
return $B \cup$ SubBridge$(l, i-1, x_{\text{off}}, y_{\text{new}})$
  $\cup$ SubBridge$(i+2, l, x_{\text{new}}, y_{\text{off}})$

Figure 1.11: Bridge

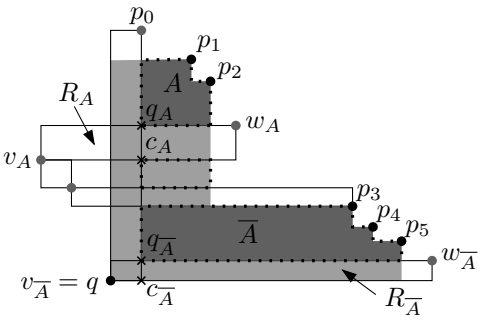


Figure 1.12: Notation: $\mathcal{A}_{\text{quad}}(q,1)$ shaded, $\Delta(q,1)$ with dotted boundary, and $\delta(q,1) = A \cup A'$ with dark shading.

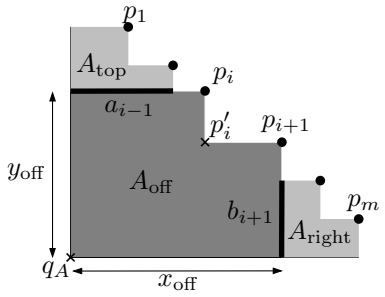


Figure 1.13: Notation for algorithm BRIDGE.

Let i be the index determined in the first call to algorithm SUBBRIDGE, see Figure 1.11. If $i > 1$, let A_{top} be the part of A properly above a_{i-1}, otherwise let $A_{\text{top}} = \emptyset$. If $i < m-1$, let A_{right} be the part of A properly to the right of b_{i+1}, otherwise let $A_{\text{right}} = \emptyset$. Now let $A_{\text{off}} = A \setminus (A_{\text{top}} \cup A_{\text{right}})$. Note that $a_{i-1} \cup b_{i+1} \subset A_{\text{off}}$.

By induction we can assume that $|B(A_{\text{top}})| \leqslant 2|N_{\text{opt}} \cap A_{\text{top}}|$ and $|B(A_{\text{right}})| \leqslant 2|N_{\text{opt}} \cap A_{\text{right}}|$. Thus, we are done if we can show that $\alpha_{i-1} + \beta_{i+1} \leqslant 2|N_{\text{opt}} \cap A_{\text{off}}|$ (*). The network N_{opt} has to contain segments in A_{off} in order to satisfy $\{(p_i, q), (p_{i+1}, q)\}$, more precisely $|N_{\text{opt}} \cap A_{\text{off}}| \geqslant \min\{\alpha_i, \beta_i, \alpha_{i-1} + \beta_{i+1}\}$. Obviously (*) holds if N_{opt} contains segments of length at least $\alpha_{i-1} + \beta_{i+1}$ in A_{off}. Therefore, it remains to show that $\alpha_{i-1} + \beta_{i+1} \leqslant 2\min\{\alpha_i, \beta_i\}$. We make a case distinction depending on how i was derived. If $\Lambda = \emptyset$, then $i = 1$, $\alpha_1 > \beta_1$ and $A_{\text{top}} = \emptyset$. In this case only b_2 is added to B and $\beta_2 < \min\{\alpha_1, \beta_1\} = \beta_1$. If $i' = \max \Lambda = m-1$, an analogous argument holds. Next we analyze the case $i' < m-1$ and $\alpha_{i'} > \beta_{i'+1}$, where i is set to i'. This yields that $\beta_{i+1} < \alpha_i$ and thus $\alpha_{i-1} + \beta_{i+1} < 2\alpha_i$. On the other hand, by the definition of Λ, we have $\alpha_i \leq \beta_i$. Hence $2\alpha_i \leq 2\min\{\alpha_i, \beta_i\}$. It remains to analyze the case $i' < m-1$ and $\alpha_{i'} \leq \beta_{i'+1}$, where i is set to $i'+1$. This yields $\alpha_{i-1} \leqslant \beta_i$ and thus $\alpha_{i-1} + \beta_{i+1} < 2\beta_i$. On the other hand, by the definition of Λ, we now have $\alpha_i > \beta_i$. Hence $2\beta_i \leq 2\min\{\alpha_i, \beta_i\}$. ♣

We conclude this section by analyzing the running time of APPROXMMN.

Theorem 1.1 APPROXMMN *runs in $O(n \log n)$ time and uses $O(n)$ space.*

Proof. Each of the four phases of our algorithm takes $O(n \log n)$ time: for phase 0 refer to Lemma 1.2, for phase I to Lemmas 1.3 and 1.6, for phase II to Lemma 1.8 and for phase III to Theorem 1.9. APPROXMMN outputs $O(n)$ line segments. ♣

1.6 The approximation factor

As desired we can now bound the length of N in $\mathcal{A}_{12}$ and $\mathcal{A}_3$ separately. Theorem 1.9 and Lemma 1.7 directly imply that $|N \cap \mathcal{A}_3| = |N_3| \leqslant 2|N_{\text{opt}} \cap \mathcal{A}_3|$. Note that by $|N_{\text{opt}} \cap \mathcal{A}_3|$ we actually mean $|\{s \cap \mathcal{A}_3 : s \in N_{\text{opt}}\}|$. It remains to show that $|N \cap \mathcal{A}_{12}| = |N_1 \cup N_2|$ is bounded by $3|N_{\text{opt}} \cap \mathcal{A}_{12}|$.

Recall that by Lemmas 1.1 and 1.3, $|N_1| \leqslant |N_{\text{opt}}| + H + W$. Since the segments of N_{opt} that were used to derive the estimation of Lemma 1.1 lie in $\mathcal{A}_{\text{ver}} \cup \mathcal{A}_{\text{hor}} \subset \mathcal{A}_{12}$, even the

stronger bound $|N_1| \leqslant |N_{\text{opt}} \cap \mathcal{A}_{12}| + H + W$ holds. It remains to analyze the length of N_2 segments. Let N_2^{ver} (N_2^{hor}) denote the set of all vertical (horizontal) segments in N_2. We will compare the length of N_2^{ver} to the length of $\mathcal{C}_{\text{ver}}$ and the length of N_2^{hor} to the length of $\mathcal{C}_{\text{hor}}$. Lemma 1.12 will yield the desired length bounds. In the following we show how the length bound for N_2^{ver} is obtained, this is the more complicated case as the connecting segments are vertical. First, we need to distinct the connecting segments and all other segments of N_2. We call the non-connecting segments in N_2 *boundary segments* as they lie on $\partial\mathcal{A}_3$. Due to Lemma 1.8, segments in N_2^{ver} and segments in $\mathcal{C}_{\text{ver}}$ intersect at most in segment endpoints. Thus, a horizontal line ℓ with $\ell \cap P = \emptyset$ does not contain any point that lies at the same time in $\bigcup \mathcal{C}_{\text{ver}}$ and in $\bigcup N_2^{\text{ver}}$. We restrict ourselves to such lines, this makes no difference in terms of overall length as we exclude only a finite number of lines. In order to obtain Lemma 1.12, we will characterize the sequences that are obtained by the intersection of such a line ℓ with cover and boundary segments and cover and connecting segments, see Lemma 1.10 and Lemma 1.11, respectively.

Lemma 1.10 *Let ℓ be a horizontal line with $\ell \cap P = \emptyset$ and $\ell \cap \text{BBox}(P) \neq \emptyset$. Consider the sequence of boundary and cover segments intersected by ℓ. Then*

(i) No more than two boundary segments are consecutive.

(ii) The left- and the rightmost segments are cover segments.

Proof. We show that each boundary segment s is (directly) preceeded or succeeded by a cover segment. This implies immediately (i). The kind of cover segment (predecessor or successor) that is assigned to a boundary segment shows that no boundary segments can be left- or rightmost and thus (ii) follows.

We show the above statement for boundary segments that lie on the boundary of a connected component A of $\mathcal{A}_3$. W.l.o.g. we assume that A is part of a region of type $\delta(q, 1)$. Let $p_1, \ldots, p_m$ be the input points on ∂A ordered according to x-distance from q. As earlier, let $v_A = p_1.\text{ynbor}[3]$ and let w_A be the horizontal successor of v_A. Let R denote the rectangle in $\mathcal{R}_{\text{ver}}$ defined by the point q and its vertical successor p_0, see Figure 1.14. Let $p_\ell = s \cap \ell$ and let y_ℓ be the y-coordinate of ℓ. Note that $p_\ell \in \bigcup N_2^{\text{ver}}$ and thus $p_\ell \notin \bigcup \mathcal{C}_{\text{ver}}$. There are two cases for the type of s.

First, s could be the boundary segment to the left of A. In this case s lies on the right vertical edge of R. Let $q_\ell = (x_q, y_\ell)$ be the point opposite of p_ℓ on the left vertical edge of R. Then $q_\ell \in \bigcup \mathcal{C}_{\text{ver}}$ since $R \in \mathcal{R}_{\text{ver}}$ and $p_\ell \notin \bigcup \mathcal{C}_{\text{ver}}$. Due to $\text{int}(R) \subset \text{int}(\mathcal{A}_{12})$, no boundary segments intersects the relative interior of $\text{Seg}[p_\ell, q_\ell]$, and thus p_ℓ is preceeded by $q_\ell \in \bigcup \mathcal{C}_{\text{ver}}$ on ℓ.

Second, s could be a vertical "staircase segment" to the right of A. In this case we show that s is succeeded by a segment in $\mathcal{C}_{\text{ver}}$. There are two subcases: either s is the left edge of $\text{BBox}(p_i, p_{i+1})$ for some $i \in \{1, \ldots, m-1\}$ or the left edge of $\text{BBox}(p_m, w_A)$. For the first subcase let β denote $\text{BBox}(p_i, p_{i+1})$. We show that ℓ intersects a vertical cover segment in β. At the same time we show that $\beta \cap \mathcal{A}_3 = \emptyset$, and hence there is no boundary segment in the interior of β. This is done by characterizing the point pairs $(p', q') \in Z_{\text{quad}}$ with $\text{BBox}(p', q') \cap \text{int}(\beta) \neq \emptyset$ and showing that the connected component of $\mathcal{A}_3$ that is incident to p' does not intersect β. Let σ and τ be the vertical and horizontal strips, respectively, that are induced by β, see Figure 1.14. The strip τ does not contain any input point to the left of β since this would contradict p_i and p_{i+1} lying in the same connected component of $\delta(q, 1)$. The

strip σ does not contain any input point below β since this would contradict $(p_{i+1}, q) \in Z_{\text{quad}}$. Let β' be β minus its right and top edge. There is no input point in β', otherwise there would be a point $p \in \beta'$ with $(p, q) \in Z_{\text{quad}}$ contradicting p_i and p_{i+1} being consecutive. Let r be the rightmost input point on the top edge of β and let t be the topmost input point on the right edge of β. (Possibly $p_i = r$ and $p_{i+1} = t$.) Since there is a point $r' \in Q(r, 4)$ with $(r', r) \in Z_{\text{hor}}$ and a point $t' \in Q(t, 2)$ with $(t', t) \in Z_{\text{ver}}$, we must have that $q' = t$ and $p' \in Q(q', 2)$, otherwise $\text{BBox}(p', q')$ would not intersect $\text{int}(\beta)$. Observe that the rectangle $\text{BBox}(r, r') \in \mathcal{R}_{\text{hor}}$ splits $\text{BBox}(p', q')$ into two connected components. However, the component incident to p' does not intersect $\text{int}(\beta)$, and thus $\beta \cap \mathcal{A}_3 = \emptyset$. Since $\text{BBox}(t, t') \in \mathcal{R}_{\text{ver}}$ and $y_{t'} \geq y_{p_i}$, it is clear that ℓ intersects a vertical cover segment in β, either the one that is induced by the non-degenerate rectangle $\text{BBox}(t, t')$ if $y_\ell \geqslant y_t$ or by the degenerate rectangle $\text{BBox}(p_{i+1}, t)$ itself if $y_\ell < y_t$.

Last, we examine the subcase that ℓ intersects $\beta = \text{BBox}(p_m, w_A)$. We have to proceed differently as we lose the property that no input point lies in the vertical strip below β. Consider $b = p_m.\text{xnbor}[4]$ (allowing $b = w_A$). We assume w.l.o.g. $y_{p_m} > y_b$ otherwise let $b = b.\text{xnbor}[4]$ until this is the case. Now, b could lie in $\text{int}(\beta)$, but only if there is a point b' with $x_{b'} = x_b$ and $y_{b'} < y_{w_A}$ otherwise there would be a point $p \in \text{int}(\beta)$ with $(p, q) \in Z_{\text{quad}}$. We discard this case for a moment and assume that already $y_b < y_{w_A}$ holds. Now, there is a point $p' \in Q(b, 2)$ with $(p', b) \in Z_{\text{ver}}$. By the construction, it is clear that $y_{p'} \geqslant y_{p_m}$ and thus the vertical line through b splits β into two connected components. For the component β' incident to p_m we can use the same argument as above to show that $\beta' \cap \mathcal{A}_3 = \emptyset$ since the vertical strip below β' does not contain any input points by construction. Hence, s is succeeded by a vertical cover segment in $\text{BBox}(p', b)$. Now, back to the discarded case: if $y_\ell < y_b$, s is succeeded by the degenerate rectangle $\text{BBox}(b, b')$, otherwise the same argument holds with $p' \in Q(b, 2)$ and $(p', b) \in Z_{\text{ver}}$. ♣

For the following characterization of connecting segments note that such segments lie only in non-degenerate rectangles of $\mathcal{R}_{\text{hor}}$.

Lemma 1.11 *Let ℓ be a horizontal line that intersects the interior of a rectangle $R_\ell \in \mathcal{R}_{\text{hor}}$. Consider the sequence of connecting and cover segments in R_ℓ. Then*

(i) No connecting segment lies on a vertical edge of R_ℓ.

(ii) No more than two connecting segments are consecutive.

(iii) At least one of the two leftmost segments is a cover segment.

(iv) At least one of the two rightmost segments is a cover segment.

(v) The left- or rightmost segment is a cover segment.

Proof. In order to show (i), we show that no connecting segment is incident to an input point. By construction, each connecting segment $s_A = \text{Seg}[q_A, c_A]$ lies on a vertical edge of a rectangle $R = \text{BBox}(q, p_0) \in \mathcal{R}_{\text{ver}}$ and in a rectangle $R_A = \text{BBox}(v_A, w_A) \in \mathcal{R}_{\text{hor}}$. By construction must R be non-degenerate, otherwise $q_A \in \bigcup \mathcal{C}_{\text{ver}}$. Thus, $c_A \neq q$. Clearly $q_A \neq q$. Now $\{c_A, q_A\} \cap P \setminus \{q\} \neq \emptyset$ would contradict $(p, q) \in Z_{\text{quad}}$ for any point $p \in \partial A \cap P$. Hence, s_A is not incident to an input point.

Now, since a connecting segment s_A is not in $\mathcal{C}_{\text{ver}}$ and lies on a vertical edge of a rectangle $R \in \mathcal{R}_{\text{ver}}$ it is pre- or succeeded by the cover segment on the opposite edge of R. This directly shows (ii), (iii) and (iv).

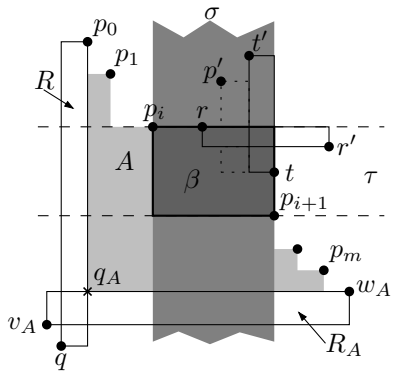


Figure 1.14: The area $\mathrm{int}(\tau \cap \beta)$ does not intersect any boundary segment, but a segment in $\mathcal{C}_{\mathrm{ver}}$.

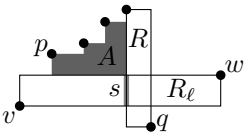


Figure 1.15: An impossible constellation: $(w, p) \in Z_{\mathrm{hor}}$ excludes $(p, q) \in Z_{\mathrm{quad}}$.

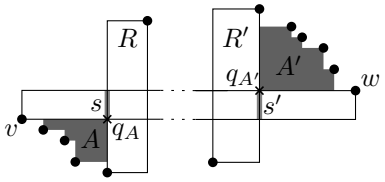


Figure 1.16: Not both s and s' lie in N_2.

Our proof for (v) is by contradiction: we assume that the leftmost segment s and the rightmost segment s' in R_ℓ are connecting segments. Let $R_\ell = \mathrm{BBox}(v, w)$. Let w.l.o.g. v be the lower left point and w be the upper right point of R_ℓ, see Figure 1.15. Let A and A' be the connected components of $\mathcal{A}_3$ with $s = s_A$ and $s' = s_{A'}$. Note that $R_A = R_{A'} = R_\ell$. Let R and R' be the rectangles in $\mathcal{R}_{\mathrm{ver}}$ whose vertical edges contain s and s', respectively. Clearly s must lie on the left edge of R and s' on the right edge of R'. Thus, A must be a region of type $\delta(q, 2)$ or $\delta(q, 3)$. First, assume $A \subseteq \delta(q, 2)$ for some $q \in P$. Then would A lie above R and q below R, see Figure 1.15. However, this is impossible. Let p be the leftmost point in $P(q, t) \cap \partial A$. Then p has a Z_{hor} partner in $Q(p, 4)$ which contradicts (p, q) being in Z_{quad}. Thus, $A \subseteq \delta(q, 3)$ and analogously $A' \subseteq \delta(q', 1)$ for some $q' \in P$, see Figure 1.16. Now, the Manhattan v–w path in N_1 contains at least one of the corner points q_A or $q_{A'}$. This contradicts s and s' both being connecting segments. ♣

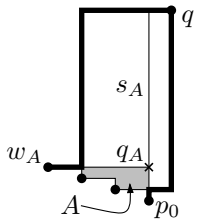
Combining Lemma 1.10 and Lemma 1.11 yields:

Lemma 1.12 $|N_2^{\mathrm{ver}}| \leqslant 2|\mathcal{C}_{\mathrm{ver}}| - H$ *and* $|N_2^{\mathrm{hor}}| \leqslant 2|\mathcal{C}_{\mathrm{hor}}| - W$.

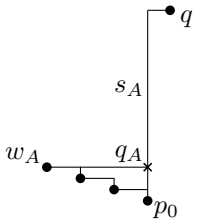
Proof. For a horizontal line ℓ with $\ell \cap P = \emptyset$ we want to compare the numbers $\#N_2^{\mathrm{ver}}$ and $\#\mathcal{C}_{\mathrm{ver}}$ of segments in N_2^{ver} and $\mathcal{C}_{\mathrm{ver}}$ intersected by ℓ, respectively. If we show that $\#N_2^{\mathrm{ver}} \leqslant 2\#\mathcal{C}_{\mathrm{ver}} - 1$, $|N_2^{\mathrm{ver}}| \leqslant 2|\mathcal{C}_{\mathrm{ver}}| - H$ follows. (Sweep $\mathrm{BBox}(P)$ from bottom to top. The at most n lines that we have to exclude draw no distinction in terms of length.) It remains to show that $\#N_2^{\mathrm{ver}} \leqslant 2\#\mathcal{C}_{\mathrm{ver}} - 1$. Observe that due to Lemma 1.11 (i), ℓ intersects connecting segments at most within the interior of a rectangle in $\mathcal{R}_{\mathrm{hor}}$. On the other hand, due to the definition of $\mathcal{A}_3$, ℓ does not intersect any boundary segments within the interior of such a rectangle. We investigate three cases.

First, consider the case that ℓ intersects no connecting segment. Thus, only cover and boundary segments are intersected. By Lemma 1.10 at most two boundary segments are consecutive and both the left- and rightmost intersected segments are cover segments. By a simple counting argument, this even yields $\#N_2^{\mathrm{ver}} \leqslant 2\#\mathcal{C}_{\mathrm{ver}} - 2$.

Second, consider the case that ℓ intersects no boundary segments. Then, by Lemma 1.11 (ii) and (v), at most two connecting segments are consecutive and the left- or rightmost segment is a cover segment. Now, further using Lemma 1.11 (iii) and (iv) yields $\#N_2^{\mathrm{ver}} \leqslant 2\#\mathcal{C}_{\mathrm{ver}} - 1$ as desired.



(a) Network computed by ApproxMMN.



(b) An MMN of the same point set.

Figure 1.17: A bad example.

Third, consider the case that ℓ intersects both boundary and connecting segments. Lemmas 1.10 (ii) and 1.11 (v) yield that the left- or rightmost intersected segment is a cover segment. Thus if in the sequence of segments intersected by ℓ at most two segments in N_2^{ver} are consecutive, we are in the same situation as in the second case. Hence $\#N_2^{\text{ver}} \leqslant 2\#\mathcal{C}_{\text{ver}} - 1$.

However, there is a case in which more than two N_2^{ver} segments are consecutive: two consecutive boundary segments are succeeded (or preceeded) by a rectangle $R \in \mathcal{R}_{\text{hor}}$. Due to Lemma 1.11 (iii) and (iv) at most one of the following two segments within R is a connecting segment. Hence, no more than three segments in N_2^{ver} are consecutive. If there are three consecutive segments in N_2^{ver}, then one of them is a connecting segment that is left- or rightmost in R. W.l.o.g. we assume that the connecting segment is leftmost in R. Then by Lemma 1.11 (v) the rightmost segment in R is a cover segment. From this we deduce two things: (a) since ℓ intersects at most one rectangle in $\mathcal{R}_{\text{hor}}$, three consecutive segments in N_2^{ver} occur at most once. (b) If there are three such segments, then by Lemma 1.10 (ii) both the left- and rightmost segments intersected by ℓ are cover segments. Hence, we again have $\#N_2^{\text{ver}} \leqslant 2\#\mathcal{C}_{\text{ver}} - 1$.

To bound the length of N_2^{hor} segments is easier since connecting segments are vertical. An analogous, simpler argument holds. ♣

This finally settles the approximation factor of ApproxMMN.

Theorem 1.2 $|N| \leqslant 3|N_{\text{opt}}|$.

Proof. By Lemma 1.12 and $|\mathcal{C}_{\text{ver}} \cup \mathcal{C}_{\text{hor}}| \leqslant |N_{\text{opt}} \cap \mathcal{A}_{12}|$ we have $|N_2| \leqslant 2|N_{\text{opt}} \cap \mathcal{A}_{12}| - H - W$. Together with $|N_1| \leqslant |N_{\text{opt}}| + H + W$ this yields $|N_1 \cup N_2|/|N_{\text{opt}} \cap \mathcal{A}_{12}| \leqslant 3$. Theorem 1.9 and Lemma 1.7 show that $|N_3|/|N_{\text{opt}} \cap \mathcal{A}_3| \leqslant 2$. Then, the disjointness of $\mathcal{A}_{12}$ and $\mathcal{A}_3$ yields $|N|/|N_{\text{opt}}| \leq \max\{|N_1 \cup N_2|/|N_{\text{opt}} \cap \mathcal{A}_{12}|, |N_3|/|N_{\text{opt}} \cap \mathcal{A}_3|\} \leq 3$. ♣

In Figure 1.17 a network computed by ApproxMMN and an MMN of the same point set are depicted. The example indicates that there are point sets P for which the ratio $|N|/|N_{\text{opt}}|$ is arbitrarily close to 3, where N is the Manhattan network that ApproxMMN computes for P. The reason for the particularly bad performance of ApproxMMN on this point set is that neither the w_A–q path nor the p_0–q path (bold solid line segments) contain the point q_A. This forces ApproxMMN to use the connecting segment s_A.

However, the example of Figure 1.17 is rather artificial. We were sure that, like most approximation algorithms, ApproxMMN performs significantly better in practice. In Section 1.8 we evaluate how ApproxMMN behaves on randomly generated point sets. To be able to compare the network computed by ApproxMMN with an MMN, we established a mixed-integer programming (MIP) formulation which is detailed in the next section.

1.7 Mixed-integer program

In this section we give a MIP formulation of the MMN problem that first appeared in [1]. It is based on network flows. For each pair of points (p, q) in Z we guarantee the existence of a Manhattan p–q path by requiring an integer flow from p to q.

We need some notation: For the set P of n input points with $p_1 = (x_1, y_1), \ \ldots, p_n = (x_n, y_n)$ let $x^1 < \cdots < x^u$ and $y^1 < \cdots < y^w$ be the ascending sequences of x- and y-coordinates of the input points, respectively. The grid Γ induced by P consists of the *grid points* (x^i, y^j) with $i = 1, \ldots, u$ and $j = 1, \ldots, w$. In this section we assume that all pairs $(p, q) \in Z$ are directed such that $x_p \leq x_q$ holds. Now, for each pair $(p, q) \in Z$ let $V(p, q) = \Gamma \cap \mathrm{BBox}(p, q)$ and let $A(p, q)$ be the set of arcs between horizontally or vertically adjacent grid points in $V(p, q)$. Horizontal arcs are always directed from left to right, vertical arcs point upwards if $y_p < y_q$ and downwards otherwise. Our formulation is based on the *grid graph* $G_P(V, A)$, where $V = \bigcup_{(p,q)\in Z} V(p, q)$ and $A = \bigcup_{(p,q)\in Z} A(p, q)$. Let $E = \{\{g, g'\} \mid (g, g') \in A \text{ or } (g', g) \in A\}$ be the set of undirected edges.

For each pair $(p, q) \in Z$ we enforce the existence of a p–q Manhattan path by a flow model as follows. We introduce a 0–1 variable $f(p, q, g, g')$ for each arc (g, g') in $A(p, q)$, which encodes the size of the flow along arc (g, g') from p to q. For each grid point g in $V(p, q)$ we introduce the following constraint:

$$\sum_{(g,g')\in A(p,q)} f(p, q, g, g') \; - \sum_{(g',g)\in A(p,q)} f(p, q, g', g) \quad = \quad \begin{cases} +1 & \text{if } g = p, \\ -1 & \text{if } g = q, \\ 0 & \text{else.} \end{cases} \tag{1.1}$$

This constraint enforces flow conservation at point g, as the first sum represents the total outflow and the second sum represents the total inflow at g. In total, there are $O(n^3)$ constraints and variables of this type, since $|Z| \in O(n)$ and $|V(p, q)|, |A(p, q)| \in O(|\Gamma|) = O(n^2)$. Next we introduce a continuous variable $F(g, g')$ for each edge $\{g, g'\}$ in E. This variable will in fact be forced to take a 0–1 value by the objective function and the following constraints. The MMN that we want to compute will consist of all grid edges $\{g, g'\}$ with $F(g, g') = 1$. We now add a constraint for each $\{g, g'\}$ in E and each $(p, q) \in Z$ with $\overline{gg'} \subseteq \mathrm{BBox}(p, q)$:

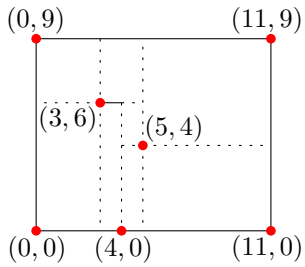
$$F(g, g') \quad \geq \quad \begin{cases} f(p, q, g, g') & \text{if } (g, g') \in A(p, q), \\ f(p, q, g', g) & \text{if } (g', g) \in A(p, q). \end{cases} \tag{1.2}$$

This constraint forces $F(g, g')$ to be 1 if the arc (g, g') or the arc (g', g) carries flow in any $A(p, q)$. Clearly we have $O(n^2)$ new variables, and accordingly $O(n^3)$ new constraints, again since $|Z| \in O(n)$. Our objective function expresses the total length of the selected grid edges:

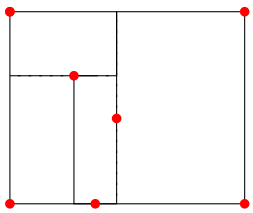
$$\min \sum_{\{g,g'\}\in E} |gg'| \cdot F(g, g'), \tag{1.3}$$

where $|gg'|$ is the Euclidean distance of g and g'. The objective function drives each $F(g, g')$ to be as small as possible. Thus, Constraint 1.2 forces $F(g, g')$ to be 0 or 1.

In total this MIP formulation uses $O(n^3)$ variables and constraints. By treating pairs in Z_{quad} more carefully, a reduction to $O(n^2)$ is possible. We implemented exact solvers based on both formulations, but it turned out that the variant with a quadratic number of constraints



(a) The result of the LP solver.



(b) The result of the MIP solver (an MMN).

Figure 1.18: A gap instance.

and variables was slower than the one with a cubic number. Therefore, we omit the details of the quadratic formulation here.

It is not hard to see that the MIP formulation (1.1)–(1.3) always yields an MMN:

Theorem 1.3 *Let P be a set of points and let A and E be defined as above. Let $F : E \to \mathcal{R}_0^+$ and $f : Z \times A \to \{0, 1\}$ be functions that fulfill (1.1) & (1.2) and minimize (1.3). Then the set of line segments $\{\overline{gg'} \mid \{g, g'\} \in E,\ F(g, g') \geq 1\}$ is an* MMN *of P.*

Due to our objective function (1.3), Equation (1.1) can be replaced by an inequality (with direction $\geq$). If the resulting constraint matrix was totally unimodular (i.e. every square submatrix has determinant in $\{-1, 0, +1\}$), every vertex of the solution polyhedron would be integral. This would mean that the relaxation of the MIP formulation yielded an MMN and thus, the MMN problem would be solvable in polynomial time. Unfortunately it turned out that this is not the case. There are instances with fractional vertices that minimize our objective function. There are even instances for which the objective value of the LP is strictly less than that of the MIP. Figures 1.18a and 1.18b show such an instance with optimal fractional and integral solution, respectively. The dotted segments in Figure 1.18a have flow 1/2. The value of the objective function is 58.5, while the length of an MMN for this point set is 60, see Figure 1.18b. For a while we hoped that we could at least prove half-integrality of the solution polyhedron. Then, rounding the LP solution would give a very simple polynomial-time factor-2 approximation. However, it was not obvious to us how to prove half-integrality of the solution polyhedron.

1.8 Experiments

To show that our algorithm performs better on average instances, we implemented ApproxMMN and the MIP formulation described in Section 1.7. Then we generated two classes of random point sets. We used the MIP solver *Xpress-Optimizer* (2003) [Das03] by Dash Optimization with the C++ interface of the BCL library to compute optimal solutions at least for small instances.

1.8.1 Experimental set-up

We implemented ApproxMMN in C++ using the compiler gcc-3.3. The two classes of random point sets, Square and HalfCircle, were generated as follows.

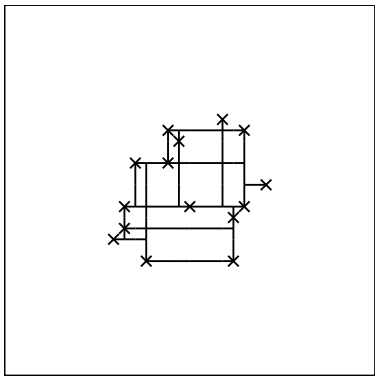


Figure 1.19: An MMN for a SQUARE-01 instance with 15 points.

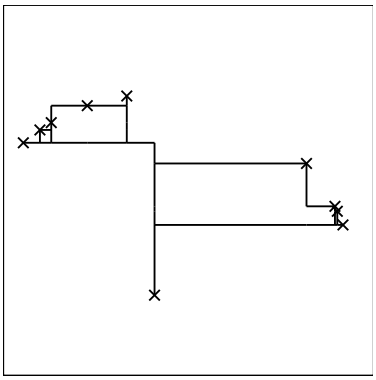


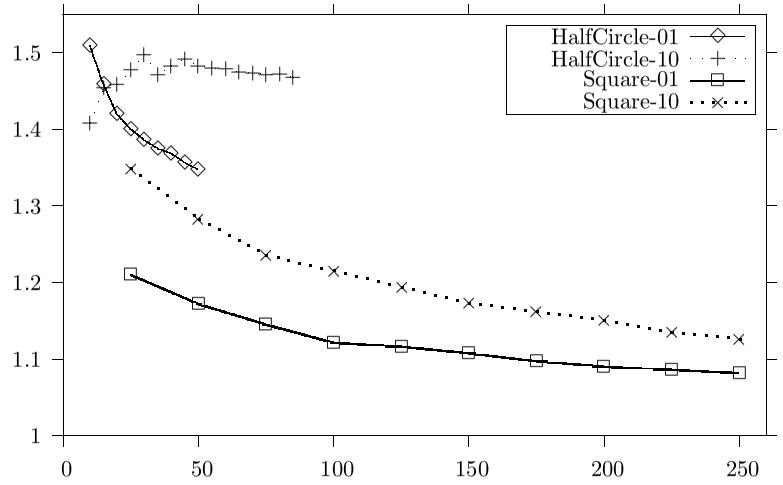
Figure 1.20: An MMN for a HALFCIRCLE-02 instance with 10 points.

SQUARE-k instances were generated by drawing n different points with uniform distribution from a $kn \times kn$ integer grid. We wanted to see the effects of having more (k small) or less (k large) points with the same x- or y-coordinate. If a pair of points shares a coordinate, the Manhattan path connecting them is uniquely determined. We used $k \in \{1, 2, 5, 10\}$. For an example of a SQUARE-01 instance see Figure 1.19.

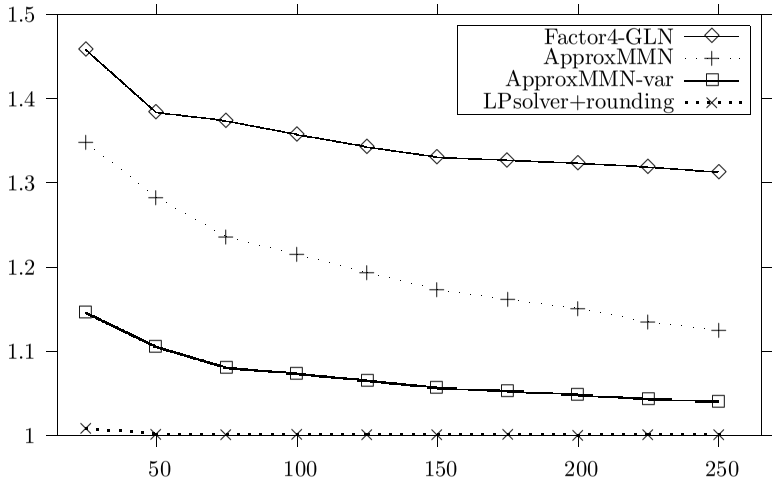
HALFCIRCLE-k instances consist of a point p_1 at the origin o and $n-1$ points on the upper half of the unit circle. The points are distributed as follows. The angular range $I = [0, \pi/4]$ is split into k subranges $I_1, \ldots, I_k$ of equal length. We used $k \in \{1, 2, 5, 10, 99\}$. Then $n-1$ random numbers $r_2, \ldots, r_n$ are drawn from I. If the number r_i falls into a subinterval of even index, it is mapped to the point $p_i = (\sin r_i, \cos r_i)$, otherwise to $p_i = (-\sin r_i, \cos r_i)$. The resulting points p_i (except for the topmost point in each quadrant and the "bottommost" point in each subinterval) form pairs (p_i, p_1) that lie in Z_{quad}. This makes HALFCIRCLE instances very different from SQUARE instances where usually only very few point pairs belong to Z_{quad}. For an example of a HALFCIRCLE-02 instance, see Figure 1.20.

We generated instances of the above types and solved them with APPROXMMN and with the Xpress-Optimizer using the MIP formulation. The results of our experiments can be found in Figures 1.21–1.23. In all graphs the sample size, i.e. the number of points per instance, is shown on the x-axis. For each sample size we generated 50 instances and averaged the results over those. In Figure 1.21a the y-axis shows the performance ratio of APPROXMMN, i.e. $|N|/|N_{\text{opt}}|$. In Figures 1.21b and 1.22 we compared the performance ratios of APPROXMMN, a slightly modified variant of APPROXMMN, and the $O(n^3)$-time factor-4 approximation algorithm of Gudmundsson et al. [GLN01]. In the graphs we skipped the factor-8 approximation algorithm [GLN01] because its results were only slightly worse than those of the factor-4 approximation: the difference was below 5%.

We also tested the performance of the following simple method to which we will refer as LPsolver+rounding. Recall that in the MIP formulation described in Section 1.7, a grid segment $\overline{gg'}$ is part of the solution if $f(p, q, g, g') = 1$ for some (p, q) in Z. Now we relax all these 0–1 variables of type $f(p, q, g, g')$ and solve the resulting linear program (LP). Our method LPsolver+rounding puts the grid segment $\overline{gg'}$ into the solution if there is some (p, q) in Z with $f(p, q, g, g') > 0$. By the construction of the MIP it is clear that this network is a Manhattan network.

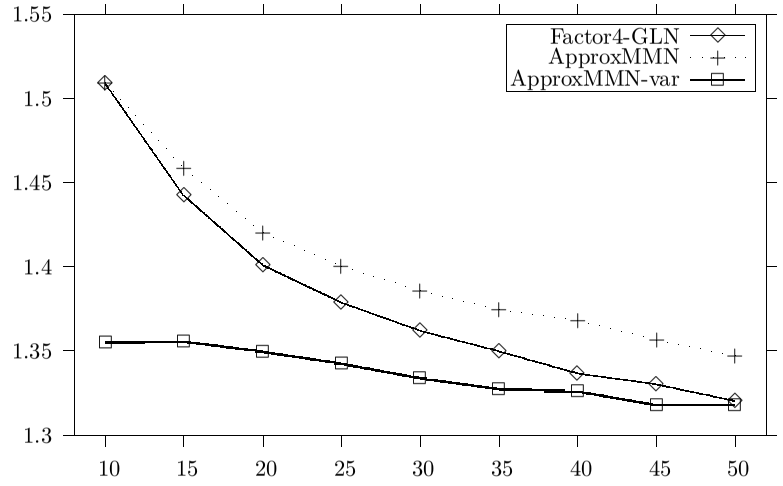


(a) ApproxMMN on various instance classes

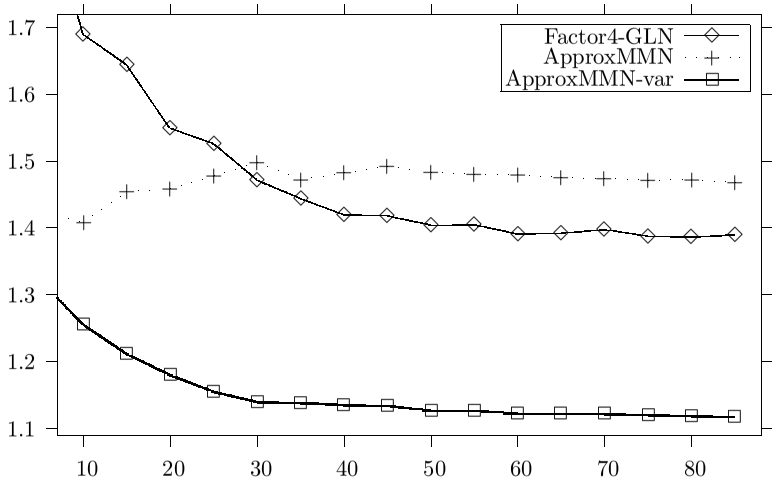


(b) Various algorithms on Square-10 instances

Figure 1.21: Performance of various algorithms.



(a) HalfCircle-01 instances



(b) HalfCircle-10 instances

Figure 1.22: Performance of various algorithms on HalfCircle instances.

In Figure 1.23 the y-axis measures the ratio between the running times of the corresponding algorithms over the running time of ApproxMMN. The asymptotic runtime of our implementation is $\Theta(n^2)$, the CPU time consumption was measured on an Intel Xeon machine with 2.6 GHz and 2 GB RAM under the operating system Linux-2.4.20. The Xpress-Optimizer was run on the same machine.

1.8.2 Results

The MIP solver required an unacceptable amount of time (i.e. at least several hours) on HalfCircle-01 instances of more than 50 points and on Square-01 instances of more than 250 points. The performance ratio of ApproxMMN seems to approach 1.1–1.2 on Square instances of increasing size, and 1.3–1.5 on HalfCircle instances, see Figure 1.21a. On HalfCircle instances we observed that with an increasing number k of subranges the performance of ApproxMMN degrades. The reason for this is that each subrange induces a connected component of type $\delta(o, 1)$ or $\delta(o, 2)$. Thus, the length of the network N_2 increases with an increasing number of subranges. Indeed, the length of N_2 seems to be the bottleneck of our algorithm.

To reduce this effect we implemented a slightly modified variant of ApproxMMN, to which we will refer as ApproxMMN-var. This variant changes only the networks N_2 and N_3. We explain the approach exemplarily for a connected component A of type $\delta(q, 1)$. Let again $p_1, \ldots, p_m$ be the input points on ∂A ordered according to x-distance from the input point q in the lower left corner of A. Let R_A be the rectangle in $\mathcal{R}_{\text{hor}}$ that touches the bottom edge of A, see Figure 1.14. Let v_A and w_A be the input points that span R_A.

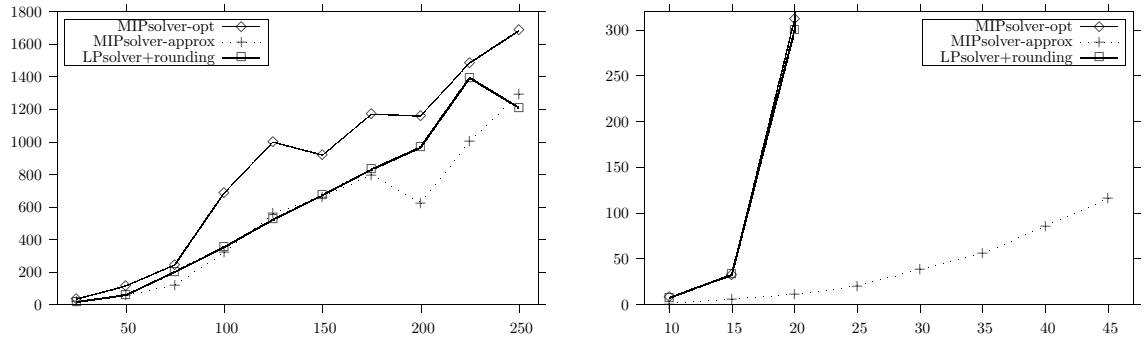
In phase II ApproxMMN-var adds only the segments $\text{Seg}[p_1, (x_{p_1}, y_{w_A})]$ and $\text{Seg}[p_m, (x_{p_1}, y_{p_m})]$ instead of the whole boundary of A to N_2. Accordingly, the connecting segment is now $\text{Seg}[(x_{p_1}, y_{w_a}), (x_{p_1}, y_{v_A})]$. As before, the connecting segment is inserted only if necessary. In phase III, a similar algorithm to algorithm Bridge is used to establish connections from $p_2, \ldots, p_{m-1}$ to (x_{p_1}, y_{w_a}). Here we use the thickest-first algorithm introduced in [GLN01]. Now the parts of ∂A that represent the staircase between p_1 and p_m are only inserted if the thickest-first algorithm requires this. However, the segments that lie on ∂A are now inserted in N_3, and there is the rub. We were not able to prove $|N_3 \cap \mathcal{A}_3| \leqslant 2|N_{\text{opt}} \cap \mathcal{A}_3|$ for ApproxMMN-var.

However, as we had hoped, the performance of ApproxMMN-var was better than that of ApproxMMN. Figure 1.21b shows the performance of ApproxMMN, ApproxMMN-var, the factor-4 approximation algorithm by Gudmundsson et al. [GLN01] and LPsolver+rounding. Exemplarily for the Square instances, we included the graphs for the Square-10 instances. The behavior of the algorithms was similar on the other Square instances, with slightly better results. On Square instances ApproxMMN performed only slightly worse than ApproxMMN-var. This is different on HalfCircle instances as Figure 1.22 shows. Especially with an increasing number of subranges the influence of N_2 on the total length of the network increases. The performance of LPsolver+rounding was amazingly good. The worst performance ratio of this method was 1.078. It occurred on a Square-10 instance with 25 points. Moreover, LPsolver+rounding solved all Circle instances optimally.

The CPU time of ApproxMMN depends neither on the value of k nor on the instance type. Solving instances with 3000 points took only about 5–6 seconds. In contrast to that, the runtime of the exact solver heavily depended on the value of k and even more on the instance type. Square instances were solved the faster the smaller k, because then the

probability for two points having the same x- or y-coordinate is higher, which predetermines a larger number of segments to be in the network. The average CPU time of the exact solver on SQUARE-10 instances with 250 points was about 170 seconds, compared to 0.1–0.2 seconds for APPROXMMN. HALFCIRCLE instances were solved slower the smaller k, because then more grid points and grid segments lie in more rectangles of $\mathcal{R}_{\text{quad}}$, which means that the MIP formulation has more constraints and variables. Generally SQUARE instances were solved much faster than HALFCIRCLE instances. This is due to the number of Z_{quad} pairs, which is significant higher in HALFCIRCLE instances. The MIP formulation requires $O(n^2)$ variables and constraints for a point pair in Z_{quad}, while it requires only $O(n)$ variables and constraints for point pairs in $Z_{\text{ver}} \cup Z_{\text{hor}}$. (There are Z_{quad} pairs that require $\Theta(n^2)$ variables and constraints.)

We wanted to see how fast the MIP solver becomes if it only has to compute a solution as good as the one computed by APPROXMMN. The Xpress-Optimizer allows to specify a bound that stops the branch-and-bound process as soon as the target function is at least as good as the bound. We refer to this version of the MIP solver as MIPsolver-approx and to the original exact version as MIPsolver-opt. The results are shown in Figure 1.23, where the average ratio between the running times of MIPsolver-approx, MIPsolver-opt and LPsolver+rounding over the running time of APPROXMMN is shown. For SQUARE-10 instances, MIPsolver-approx is not much faster than MIPsolver-opt. This changes with decreasing k: for SQUARE-01 instances MIPsolver-approx takes only about half the time of MIPsolver-opt. This is due to the fact that the smaller k the more segments in a Manhattan network are predetermined. The method LPsolver+rounding turned out to run only slightly faster than MIPsolver-opt. HALFCIRCLE instances were solved relatively fast by MIPsolver-approx. Solving HALFCIRCLE-01 instances with 45 points took MIPsolver-opt on average 2200 seconds CPU time as compared to 1.2 seconds for MIPsolver-approx (and 0.01 seconds for APPROXMMN).



(a) SQUARE-10 instances **(b)** HALFCIRCLE-01 instances

Figure 1.23: Ratios of the running times of MIPsolver-opt, MIPsolver-approx, and LPsolver+rounding over the running time of APPROXMMN.

Finally we compared the values of the objective function of the MIP and its LP relaxation. We found out that there are only few instances where the relaxation yields a smaller value of the objective function than the MIP. For an example of an instance with a gap between the two values, see Figure 1.18a. We found gaps in only three SQUARE-10 instances. Moreover, in all of these cases the gap was very small, namely less than 0.011% of the value of the objective function in the MIP formulation. In the example in Figure 1.18, which was constructed by

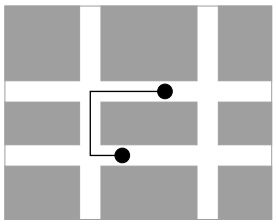


Figure 1.24: The *Real* Manhattan problem: a shortest path connecting two sites.

hand, the gap is 2.5%.

Note that the existence of a gap means that the face of the solution polyhedron with maximum objective function value has *only* fractional corners, while the existence of a fractional corner does not imply a gap. If, however, the LP solver finds such a fractional corner and there is an integral corner, then our rounding scheme returns a non-optimal network. We conjecture that there are only few point configurations that cause a gap and that these configurations cannot occur in HalfCircle instances.

1.8.3 Conclusion

For time-critical applications and large instances clearly ApproxMMN or ApproxMMN-var are the methods of choice. They solve instances with 3000 points within 5–6 seconds CPU time. On average point sets the networks they compute are usually not more than 50% longer than an MMN. Within a threshold of 100 seconds CPU time we were only able to compute optimal networks of the following sizes: HalfCircle instances of at most 25 points and Square instances of at most 175 points. The (polynomial-time!) method LPsolver+rounding returns amazingly good results, but its running time is only slightly faster than the exact solver based on our MIP formulation and thus we were only able to solve instances of the above size with it.

1.9 Open problems

The main open question is the complexity status of the MMN problem. Until now there are not even hints whether it is polynomially solvable, it is NP-hard, or has intermediate status. In the latter cases it would be of interest to find out whether a polynomial-time approximation scheme exists or whether the MMN problem cannot be approximated arbitrarily well. As mentioned in the introduction Chepoi et al. [CNV05] recently gave a factor-2 approximation algorithm that rounds the solution of a linear program with $O(n^3)$ variables and constraints. It would be interesting to see whether it is possible to establish a fast factor-2 approximation algorithm, say one that runs in $O(n \log n)$ time. In order to solve larger instances optimally, it would be of interest to design a fixed-parameter algorithm. However, it is unclear to us what to choose as parameter.

We conclude with two variants of the problem. The first variant is the *real MMN problem* where apart from the point set an underlying rectilinear grid G is given, e.g. the streets of Manhattan, on which the network has to lie. Again each pair of points must be connected by a shortest possible rectilinear path and the length of the network is to be minimized. However, the shortest rectilinear path connecting two points can now be longer than a usual

Manhattan path, see Figure 1.24. The real MMN problem is at least as hard as the MMN problem since G can be set to the grid induced by the input points.

Chepoi et al. [CNV05] suggest another variant of the MMN problem, the *F-restricted MMN problem*. Given a point set P and a set F of pairs of points in P, find a network of minimum length that connects the point pairs in F with Manhattan paths. This variant also generalizes the MMN problem, which is an F-restricted MMN problem where F is a generating set. The F-restricted MMN problem is NP-hard, since it also generalizes the rectilinear Steiner arborescence problem, which is NP-hard [SS00]. In the rectilinear Steiner arborescence problem only point sets P in the first quadrant are considered. The aim is to find a rectilinear network of minimum length that connects all points in P to the origin o. This is equivalent to solving the F-restricted MMN problem for $P' = P \cup \{o\}$ and $F = \{o\} \times P$.

The Manhattan network problem and all its variants seem to be hard because it is not possible to consider the problem locally. Two points that lie very far from each other affect all the network segments that lie within the bounding box induced by the two points. This is also the reason why it seems to be hopeless to apply dynamic programming. On the other hand it also seems to be hard to come up with a hardness proof.

Acknowledgments

We would like to thank Hui Ma for implementing large parts of ApproxMMN, and Anita Schöbel, Marc van Kreveld, Leon Peeters, Karin Höthker, Gautam Appa, Dorit Hochbaum, Bettina Speckmann, and Raghavan Dhandapani for interesting discussions.

Chapter 2

Interference-minimal networks

Constructing intereference-minimal networks yields the second chapter of the *constructing* part. As for the Manhattan networks the geometric aspect is already given by the input, the location of n points in the plane. The restriction on the resulting networks here concerns the type of network: we look for a spanning tree, a t-spanner for some additionally given $t > 1$ and a d-hop network for some additionally given $d \in \mathbb{N}$. The optimization task is to find networks of the above types that minimize the occuring interference.

The chapter is based on conference publication [3]: Marc Benkert, Joachim Gudmundsson, Herman Haverkort and Alexander Wolff: "Constructing Interference-Minimal Networks". A full version will soon be published in the *Internation Journal of Computational Geometry and Application.*

2.1 Introduction

Wireless ad-hoc networks consist of a number of wireless devices spread across a geographical area. Each device has wireless communication capability, some level of intelligence for signal processing and networking of the data, and a typically limited power source such as a small battery.

We study networks that do not depend on dedicated base stations: in theory, all nodes may communicate directly with each other. In practice however, this is often a bad idea: if nodes that are far from each other would exchange signals directly, their signals may interfere with the communication between other nodes within reach. This may cause errors, so that messages have to be sent again. Communicating directly over large distances would also require sending very strong signals, since the necessary strength depends at least quadratically on the distance (in practice the dependency tends to be cubic or worse). Both issues could lead to rapid depletion of the devices' limited power sources. Therefore it is advisable to organize communication between nodes such that direct communication is restricted to pairs of nodes that can reach each other with relatively weak signals that will only disturb a small number of other nodes. We model such a network as a graph $G = (V, E)$, in which the vertices V represent the positions of the mobile devices in the plane, and the links (or edges) E represent the pairs of nodes that exchange signals directly. Communication between nodes that do not exchange signals directly should be routed over other nodes on a path through that network. According to Prakash [Pra99], the basic communication between direct neighbors becomes

unacceptably problematic if the acknowledgement of a message is not sent on the same link in opposite direction. Therefore we will assume that the links are undirected.

Our problem is therefore to find an undirected graph on a given set of nodes in the plane, such that all nodes are connected with each other through the network (preferably over a short path), interference problems are minimized, and direct neighbors in the network can reach each other with signals of bounded transmission radius. We will focus on guaranteeing connectivity and minimizing interference; bounding the transmission radius is an easy extension, which we discuss in Section 2.4. Since wireless devices tend to move frequently, we need to be able to construct networks with the desired properties fast.

The optimal network structure depends ultimately on the actual communication that takes place. This is generally not known a priori. Therefore we aim to optimize a network property that we believe to be a good indicator of the probability that interference problems will occur. Assuming that each node can adjust the strength of each signal so that it can just reach the intended receiver, such indicators may be:

sending-link-based interference of a link $\{u, v\}$: the number of nodes that are within reach of the signals from the communication over a particular link $\{u, v\}$ in the network (proposed by Burkhart et al. [BvRWZ04], also studied by Moaveni-Nejad and Li [ML05]) – in other words, the number of nodes that are hindered when the link $\{u, v\}$ is active. This is the definition of interference we focus on in this chapter.

sending-node-based interference of a node u: the number of nodes that receive signals transmitted by u (proposed by Moaveni-Nejad and Li [ML05]) – in other words, the number of nodes that are hindered when u is active.

receiving-node-based interference of a node u: the number of nodes transmitting signals that reach u (proposed by Rickenbach et al. [vRSWZ05]) – that is, the number of nodes that may prevent u from communicating effectively.

Previous results. To construct a network that connects all nodes and minimizes the maximum and total sending-link-based interference, we could run Prim's minimum-spanning-tree algorithm [Pri57] with a Fibonacci heap. Assuming that the interference for each feasible link is given in advance, this takes $O(m + n \log n)$ time, where n is the number of nodes and m is the number of eligible links. If all possible links are considered, we can compute their interference values in $O(n^{9/4} \operatorname{polylog} n)$ time (see the proof of Lemma 2.3). This will then dominate the total running time.

To make sure that nodes are connected by a relatively short path in the network, one could construct a t-spanner on the given set of nodes. Burkhart et al. [BvRWZ04] presented a first algorithm to construct a t-spanner for given $t > 1$ such that the maximum interference of the edges in the spanner is minimized. It was later improved by Moaveni-Nejad and Li in [ML05]. Assuming that the interference for each possible link is again given in advance, the running time of their algorithm is $O(n \log n(m + n \log n))$. If all possible links are considered, the running time is $O(n^3 \log n)$.

The approach for sending-linked-based interference can be modified to optimize sending-node-based interference by defining the interference of a link $\{u, v\}$ to be the maximum of the sending-node-based interferences of u and v. The maximum occurring interference of a link will then be the maximum occurring node interference in the original sending-node-based setting. Unfortunately, we cannot apply this modification to fit the receiving-node-

based interference. With sending-link-based interference we can decide whether a link causes too much interference independently of the other links that may be active. With receiving-node-based interference this is not possible, so that completely different algorithms would be needed. Rickenbach et al. [vRSWZ05] only give an approximation algorithm for the case where all nodes are on a single line (the *highway model*).

Our results. We improve and extend the results of Burkhart et al. and Moaveni-Nejad and Li in two ways.

First, apart from considering networks that are simply connected (spanning trees) and networks with bounded dilation (t-spanners), we also consider networks with bounded link diameter, that is, networks such that for every pair of nodes $\{u, v\}$ there is a path from u to v that consists of at most d links (or 'hops'), where d is a parameter given as input to the algorithm. Such d-hop networks are useful since much of the delay while sending signals through a network is typically time spent by signals being processed in the nodes rather than time spent by signals actually traveling.

Second, we remove the assumption that the interference of each possible link is given in advance. For each of the three properties (connectivity, bounded dilation or bounded link diameter), we present algorithms that decide whether the graph G_k with all links of interference at most k has the desired property. The main idea is that we significantly restrict the set of possible links for which we have to determine the interference, in such a way that we can still decide correctly whether G_k has the desired property. To find the smallest k such that there is a network with interference k and the desired property, we do a combined exponential and binary search that calls the decision algorithm $O(\log k)$ times. The resulting algorithms are output-sensitive: their running times depend on the interference of the resulting network.

Our algorithms work for sending-link-based and sending-node-based interference. We present two models: the *exact* model and the *estimation* model. In the exact model, we assume that the signal sent by a node u to a node v reaches exactly the nodes that are not farther from u than v is. Our algorithms for this model are faster than the algorithm by Moaveni-Nejad and Li [ML05] for $k \in o(n)$. In the estimation model, we assume that it is not realistic that the boundary of a signal's reach is known precisely: for points w that are slightly farther from u than v is, counting w as being disturbed by the signal sent from u to v is as good a guess as not counting w as disturbed. It turns out that with this model, the number of links for which we actually have to compute the interference can be reduced much further, so that we get faster algorithms, especially for spanning trees with larger values of k. Our results are listed in Table 2.1.

We proceed as following: in Section 2.2 we propose our output-sensitive algorithms for the case of exact interference values, and examine their running times. In Section 2.3 we introduce our model for estimations of link interference and describe one algorithm for each of the network properties (connectivity, bounded dilation, and bounded link diameter). In Section 2.4 we briefly discuss some generalizations of our algorithms: nodes with bounded transmission radius and weighted nodes.

	exact model (expected)	estimation model (determ.)
spanning tree	$\min\{n^{9/4}\operatorname{polylog} n, nk^2 + nk\log n\}$	$\frac{n}{\varepsilon^2}(\log n + \frac{1}{\varepsilon})$
t-spanner	$n^2 \log k(k + \log n)$	$n^2 \log k(\frac{1}{\varepsilon^2} + \log n)$
d-hop network	$\min\{n^{9/4}\operatorname{polylog} n, nk^2\} + n^2 \log n \log k$	$\frac{n}{\varepsilon^2}(n \log k + \frac{1}{\varepsilon})$

Table 2.1: Running times of our algorithms to find a minimum-interference network with the required property. The running times are given in O-notation, n is the number of nodes, k is the maximum interference of any link in the resulting network, and ε specifies the relative inaccuracy with which a signal's reach and the dilation of a spanner are known. Worst-case running times for deterministic algorithms for the exact model are slightly worse than the expected running times of the randomized algorithms listed in the table. The listed running times for the estimation model are worst-case.

2.2 Computing exact-interference graphs

We are given a set V of n points in the plane in general position, that is, we assume that no three input points lie on a straight line or on a circle[1]. Our aim is to establish a wireless network that minimizes interference. First, we define interference. Let u, v be any two points in V. If the edge (link) $\{u, v\}$ is contained in a communication network, the range of u must be at least $|uv|$. Hence, if u sends a signal to v this causes interferences within the closed disk $D(u, |uv|)$ that has center u and radius $|uv|$. The same holds for v. This leads to the following definition that was first given by Burkhart et al. [BvRWZ04]. See also Figure 2.1.

Definition 2.1 ([BvRWZ04]) *Let $\binom{V}{2}$ denote the set of all pairs $\{u, v\} \in V$ with $u \neq v$. The* sphere *of an edge $e = \{u, v\}$ is defined as $S(e) := D(u, |uv|) \cup D(v, |uv|)$. For any edge $e = \{u, v\} \in \binom{V}{2}$ we define the* interference *of e by $\mathrm{Int}(e) := \big|V \cap S(e) \setminus \{u, v\}\big|$. The* interference $\mathrm{Int}(G)$ *of a graph $G = (V, E)$ is defined by $\mathrm{Int}(G) := \max_{e \in E} \mathrm{Int}(e)$.*

In this section, we will give algorithms to compute minimum-interference networks of three types. The first type is a spanning tree $\mathcal{T}$, the second type of network is, for an additionally given $t \geqslant 1$, a t-spanner, and the third type is a d-hop network, for a given integer $d > 1$.

The main idea of the algorithms is the same. For given $j \geqslant 0$ let, in the following, $G_j = (V, E_j)$ denote the graph[2] where E_j includes all edges e with $\mathrm{Int}(e) \leqslant j$. Assume that E_j can be computed by calling a subroutine *ComputeEdgeSet* with arguments V and j. Exponential and binary search are used to determine the minimum value of k for which G_k has the desired property $\mathcal{P}$, see Algorithm 2.2. We first try $upper = 0$, and compute all edges of G_0. If G_0 does not have the desired property, we continue with $upper = 1$ and then keep doubling $upper$ until G_{upper} has the desired property. We compute the interference values for each of its edges, and continue with a binary search between $lower = upper/2$ and $upper$. In each step we construct G_{middle}, the graph to be tested, by selecting the edges with interference at most $middle = \frac{1}{2}(lower + upper)$ from G_{upper}, which has already been computed. Note

[1] None of the algorithms presented in this chapter explicitly requires the point set to be in general position, however, some of the tools used assume general position, for example the concept of higher-order Delaunay edges and the range searching data structures used in Section 2.2.1.

[2] If the sphere of an edge is defined as $D(u, |uv|) \cap D(v, |uv|)$ then G_j is the j-relative neighbourhood graph, and if the sphere is the disk $D((u+v)/2, |uv|/2)$ (shaded in Figure 2.1) then G_j is the j-Gabriel graph [JT92].

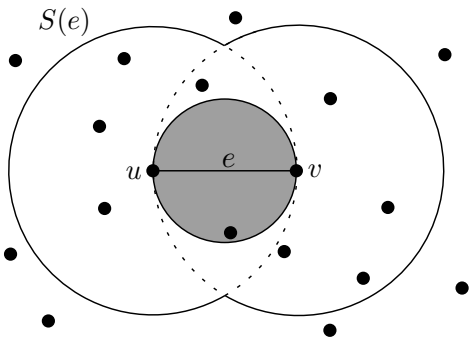


Figure 2.1: The sphere $S(e)$ of e. Here $\mathrm{Int}(e) = 9$.

MinInterferenceNetwork$(V, \mathcal{P})$

// exponential search
$upper \leftarrow 1/4$
repeat
  $upper \leftarrow 2 \cdot upper$
  $E_{upper} \leftarrow ComputeEdgeSet(V, \lfloor upper \rfloor)$
  $G_{upper} \leftarrow (V, E_{upper})$
until $FulfillsProperty(G_{upper}, \mathcal{P})$
$lower \leftarrow \lfloor upper/2 \rfloor$
// binary search
while $upper > lower + 1$ **do**
  $middle \leftarrow \frac{1}{2}(lower + upper)$
  $G_{middle} \leftarrow (V,$ all edges in E_{upper} with interference at most $middle)$
  if $FulfillsProperty(G_{middle}, \mathcal{P})$ **then**
    $upper \leftarrow middle$
  else $lower \leftarrow middle$
return G_{upper}

Figure 2.2: The basic algorithm.

that in the binary search phase, the search interval boundaries *lower* and *upper* are always even.

To get a spanning tree, we test with the property $\mathcal{P}$ = connectivity. After running Algorithm MinInterferenceNetwork to find the minimum k for which G_k is connected, we run a standard minimum spanning tree algorithm on G_k. The result is a tree $\mathcal{T}$ that minimizes both $\max_{e \in \mathcal{T}} \mathrm{Int}(e)$ and $\sum_{e \in \mathcal{T}} \mathrm{Int}(e)$ among all spanning trees. For the t-spanner and the d-hop network, the test consists of determining the dilation or the link diameter of the network. We do this with an all-pairs-shortest-paths computation.

Note that the only non-trivial steps in algorithm MinInterferenceNetwork are the subroutines *FulfillsProperty* and *ComputeEdgeSet*. We first give details on how to implement *ComputeEdgeSet*, that is, how to compute E_j and the interference values of the edges in E_j efficiently for any j.

2.2.1 Computing edge interferences

An edge $\{u, v\}$ is an *order-j Delaunay edge* if there exists a circle through u and v that has at most j points of V inside [GHvK02].

Lemma 2.1 *All edges in E_j are order-j Delaunay edges.*

Proof. Let $e = \{u, v\}$ be an edge with $\mathrm{Int}(e) \leqslant j$. Then by Definition 2.1, the sphere $S(e)$ contains at most j points other than u and v. The disk that has e as diameter, see Figure 2.1, contains at most j points in its interior as it is contained in $S(e)$ and does not contain u and v. Thus, e is an order-j Delaunay edge. ♣

There is a close connection between order-j Delaunay edges and higher-order Voronoi diagrams that we will use.

Lemma 2.2 *(Lemma 2 in [GHvK02])*
Let V be a set of n points in the plane, let $j \leqslant n/2 - 2$, and let $u, v \in V$. The edge $\{u, v\}$ is an order-j Delaunay edge if and only if there are two incident faces, F_1 and F_2, in the order-$(j+1)$ Voronoi diagram such that u is among the $j+1$ points closest to F_1 but not among the $j+1$ points closest to F_2, while v is among the $j+1$ points closest to F_2 but not among those closest to F_1.

Since the worst-case complexity of the order-$(j+1)$ Voronoi diagram is $O((j+1)(n-j-1))$ [Lee82], it follows that $O(nj)$ pairs of points give rise to all order-j Delaunay edges. This is because any two incident faces induce exactly one point pair that corresponds to a Delaunay edge. These pairs can be computed in $O(nj2^{c\log^* j} + n\log n)$ expected time [Ram99] (the best-known deterministic algorithm has a worst-case running time that is only worse by a polylogarithmic factor [Cha00]). Note that this also implies that the number of edges in E_j is bounded by $O(nj)$.

Lemma 2.3 *Given n points in the plane, (i) any edge set E_j with $j \leqslant n/2-2$ can be computed in $O(nj^2 + nj\log n)$ expected time; (ii) after $O(n^{9/4}\,\mathrm{polylog}\,n)$ preprocessing time, any edge set E_j can be computed in $O(nj)$ worst-case time.*

Proof. (i) Computing the order-$(j+1)$ Voronoi diagram and thus all $O(nj)$ order-j Delaunay edges takes $O(nj\log j + n\log n)$ expected time. A data structure of the order-$(j+1)$ Voronoi diagram can be built in $O(nj\log n)$ time that supports point location queries in $O(\log n)$ time. Thus, the time for the construction of the diagram (and the point-location data structure) dominates the time we need to determine for each node $v \in V$ the cell in the order-$(j+1)$ Voronoi diagram in which it lies. This cell corresponds to a list L_v of the $j+1$ nearest nodes to v in V.

According to Lemma 2.1, E_j is contained in the set of all order-j Delaunay edges. We simply test for each of the $O(nj)$ candidate edges in E_j whether its interference is at most j. For a candidate edge $e = \{u, v\}$ we look at the lists L_u and L_v that we have precomputed. If v is not included in L_u we conclude $\mathrm{Int}(e) \geqslant j+1$ and reject e. Analogously, we reject e if u is not included in L_v. Otherwise we traverse L_u until we find v and count the number of points found. After that we traverse L_v until we find u and count the number of points that are not in $D(u, |uv|) \cap D(v, |uv|)$ as these points have already been counted. We accept e as

an edge of E_j if and only if we count at most j points.[3] Since the lists L_u and L_v are given in advance this test can be done in $O(j)$ time per edge. Thus, testing all $O(nj)$ candidate edges requires $O(nj^2)$ time and hence computing one edge set E_j takes $O(nj^2 + nj \log n)$ expected time in total.

(ii) Project all input points (x, y) parallel to the z-axis on a three-dimensional parabola $z = x^2 + y^2$, and build a data structure on them so that we can report the number of points lying inside the intersection of two halfspaces efficiently. We use the half-space range query data structure by Matoušek [Mat93], who showed that one can build such a data structure of size $O(M)$ in time $O(n^{1+\delta} + M \operatorname{polylog} n)$, such that queries can be answered in $O((n/M^{1/3}) \operatorname{polylog} n)$ time, where δ is an arbitrarily small positive constant. We choose $M = n^{9/4}$, and get preprocessing time $O(n^{9/4} \operatorname{polylog} n)$ and query time $O(n^{1/4} \operatorname{polylog} n)$. Observe that in the plane, an input point lies inside a circle if and only if in three dimensions, its projection on the parabola lies below the plane that contains the projection of the circle on the parabola. The number of points in $D(u, |uv|)$ and $D(v, |uv|)$ can thus be determined by halfspace queries in the data structure mentioned above, and the number of points lying in $D(u, |uv|) \cap D(v, |uv|)$ is determined by querying the data structure with the intersection of two halfspaces.

We use this to compute $\mathrm{Int}(e)$ for all $O(n^2)$ possible edges e in $O(n^{9/4} \operatorname{polylog} n)$ total query time, sort the results for all edges by increasing interference value in $O(n^2 \log n)$ time, and store the results. After that, any edge set E_j can be found in $O(nj)$ time by simply selecting the edges e with $\mathrm{Int}(e) \leqslant j$ from the head of the sorted list. ♣

2.2.2 The total running time

Theorem 2.1 *Algorithm* MinInterferenceNetwork *runs in* $O(\min\{nk^2 + nk \log n, n^{9/4} \operatorname{polylog} n\} + P(n, nk) \log k)$ *expected time, where n is the number of nodes, k is the interference of the network output, and $P(n, m)$ is the runtime of FulfillsProperty on a graph with n nodes and m edges.*

Proof. During the exponential-search phase, *ComputeEdgeSet* is called $\lceil \log k \rceil$ times to compute an edge set E_j in time $O(nj^2 + nj \log n)$. As the $j's$ in the running time are geometrically increasing values, the terms in the running time that depend on j are dominated by the computation of E_k. The total expected time spent by *ComputeEdgeSet* is thus $O(nk^2 + nk \log n)$ (by Lemma 2.3 (i)). Once the total time spent by *ComputeEdgeSet* has accumulated to $\Omega(n^{9/4} \operatorname{polylog} n)$, we compute the interference values for all possible edges at once and sort them in $O(n^{9/4} \operatorname{polylog} n)$ time. After this, we can identify all sets $E_{\lfloor upper \rfloor}$ for geometrically increasing values of *upper* up to at most $2k$ easily in $O(nk)$ time (by Lemma 2.3 (ii)). In the binary-search phase, selecting edges of E_{middle} from E_{upper} takes $O(nk)$ time, which is done $O(\log k)$ times for a total of $O(nk \log k)$. A total of $O(\log k)$ tests for the property $\mathcal{P}$ on graphs with $O(nk)$ edges takes $O(P(n, nk) \cdot \log k)$ time. ♣

[3] Note that the computation of the order-j Voronoi diagram instead of the order-$(j + 1)$ Voronoi diagram would not be sufficient to decide whether $\mathrm{Int}(e) \leqslant j$. For example, the nearest neighbor of u could be v while u is not one of the j nearest neighbors of v. Then we would have $\mathrm{Int}(e) \geqslant j$, but we cannot decide whether $\mathrm{Int}(e) = j$.

2.2.3 Minimum-interference spanning trees

In this section we consider the basic problem of computing a connected graph with minimum interference. For a graph with n nodes and m edges, we can test by breadth-first search in $O(n+m)$ worst-case time whether it is connected. We have $P(n, nk) = O(nk)$ and by applying Theorem 2.1 we get:

Corollary 2.1 *We can compute a connected graph with minimum-interference in expected time* $O(\min\{nk^2 + nk\log n, n^{9/4}\operatorname{polylog} n\})$*, where* k *is the interference of the network output.*

We can compute the minimum spanning tree of the resulting graph in $O(nk+n\log n)$ time (for example with Prim's algorithm), with the weights of the edges set to their interference values. Note that any minimum spanning tree $\mathcal{T}$ on the given set of vertices not only minimizes $\sum_{e\in\mathcal{T}} \mathrm{Int}(e)$ but also $\mathrm{Int}(\mathcal{T}) = \max_{e\in\mathcal{T}} \mathrm{Int}(e)$ for the set of all connected trees. Therefore such a tree can be found in the minimum-interference graph G_k obtained by Corollary 2.1. By running Prim's algorithm on G_k, we can compute $\mathcal{T}$ from G_k in $O(nk + n\log n)$ time, i.e., $P(n, nk) = O(nk + n\log n)$. We get the following corollary:

Corollary 2.2 *We can compute a spanning tree* $\mathcal{T}$ *of* V *with minimum-interference that minimizes* $\sum_{e\in\mathcal{T}} \mathrm{Int}(e)$ *and* $\max_{e\in T} \mathrm{Int}(e)$ *in expected time* $O(\min\{nk^2+nk\log n, n^{9/4}\operatorname{polylog} n\})$.

2.2.4 Minimum-interference t-spanners

It is often desired that a network is not only a connected network, but also a t-spanner for some given constant t [BvRWZ04, LW04]. The dilation of a graph with n nodes and m edges, and thus the check whether it is a t-spanner, can be computed in $O(nm + n^2\log n)$ time by computing all pairs' shortest paths with Dijkstra's algorithm [Dij59]. We have $P(n, nk) = O(n^2k + n^2\log n)$ and by applying Theorem 2.1 we get:

Corollary 2.3 *For any* $t > 1$ *we can compute a* t*-spanner of* V *with minimum-interference in* $O(n^2\log k(\log n + k))$ *expected time.*

The weight of a graph G, denoted $wt(G)$, is defined as the sum of its edge lengths. By adding a postprocessing step to the computation of a minimum-interference t-spanner, we obtain the following:

Lemma 2.4 *For any constants* $t > 1$ *and* $\varepsilon > 0$ *we can compute a* $(1+\varepsilon)t$*-spanner of* V *with weight* $O(wt(MST(V)))$ *and interference* k *in* $O(n^2\log k(\log n + k) + nk\log n/\varepsilon^7)$ *expected time, where* $MST(V)$ *is a minimum weight spanning tree of* V *and* k *is the interference of the minimum-estimated interference of any* t*-spanner of* V*.*

Proof. Compute a t-spanner G of V with minimum-interference using Corollary 2.3, and let k be the interference of G. Note that G has $O(nk)$ edges. Now we argue that we can compute a $(1+\varepsilon)$-spanner G' of G in $O(nk\log n/\varepsilon^7)$ time such that G' has interference k and weight $O(wt(MST(V)))$.

Let G be an arbitrary t-spanner for a point set V having m edges, where $t > 1$ is a constant. Let ε be an arbitrary positive real constant. The greedy algorithm in [GLN02] computes a subgraph G' of G that is a $(1+\varepsilon)$-spanner of G and that satisfies the so-called leapfrog property. The graph G' can be computed in $O(m\log n/\varepsilon^7)$ time, for details see the

book by Narasimhan and Smid [NS07]. Das and Narasimhan [DN97] have shown that any set of edges that satisfies the leapfrog property has weight $O(wt(MST(V)))$. This completes the proof of the lemma. ♣

2.2.5 Minimum-interference d-hop networks

We can test whether a given graph is a d-hop network by computing its link diameter and checking if it is at most d. For a graph with n nodes and m edges, the link diameter can be computed in $O(nm)$ worst-case time by doing a breadth-first search from every vertex. Alternatively, since all edge weights for the distance computation are '1' we can compute the link diameter in $O(n^2 \log n)$ expected time with the all-pairs-shortest-paths algorithm by Moffat and Takaoka [MT87]. Thus, $P(n, nk) = O(n^2 \log n)$ and by applying Theorem 2.1 we get:

Corollary 2.4 *For any integer $d > 1$ we can compute a d-hop network with minimum-interference in $O(\min\{nk^2, n^{9/4} \operatorname{polylog} n\} + n^2 \log n \log k)$ expected time.*

2.3 Computing estimation-interference graphs

In this section we show how to compute minimum-interference networks in the estimated model. We define estimated interference as follows (see Fig. 2.3a):

Definition 2.2 *Let $D(u, r)$ be the closed disk centered at u with radius r. The $(1+\varepsilon)$-*sphere *$S_{1+\varepsilon}(e)$ of an edge $e = \{u, v\}$ is defined as $S_{1+\varepsilon}(e) := D(u, (1+\varepsilon) \cdot |uv|) \cup D(v, (1+\varepsilon) \cdot |uv|)$. For $0 \leqslant \varepsilon_1 \leqslant \varepsilon_2$ we say that an integer I is an $(\varepsilon_1, \varepsilon_2)$-valid estimation of the interference of e if and only if $\big|V \cap S_{1+\varepsilon_1}(e) \setminus \{u, v\}\big| \leqslant I \leqslant \big|V \cap S_{1+\varepsilon_2}(e) \setminus \{u, v\}\big|$.*

We will use *ε-valid estimation* as a shorthand for *$(0, \varepsilon)$-valid estimation*. Our aim is to compute minimum-interference networks based on ε-valid estimations of interference. To do so we will need to fix a particular assignment $\mathrm{Int}_\varepsilon : \binom{V}{2} \to \mathbb{N}$ of ε-valid estimations for all edges, which will be explained in more detail below.

2.3.1 The well-separated pair decomposition

Our construction uses the well-separated pair decomposition by Callahan and Kosaraju [CK95], which we briefly present here.

Definition 2.3 ([CK95]) *Let $s > 0$ be a real number, and let A and B be two finite sets of points in $\mathbb{R}^2$. We say that A and B are* well-separated *with respect to s, if there are two disjoint disks D_A and D_B of same radius r, such that*

(i) D_A contains A,
(ii) D_B contains B, and
(iii) the minimum distance between D_A and D_B is at least $s \cdot r$.

The parameter s will be referred to as the *separation constant*. The next lemma follows easily from Definition 2.3.

Lemma 2.5 ([CK95]) *Let A and B be two finite sets of points that are well-separated with respect to s, let u and x be points of A, and let v and y be points of B. Then*

(i) $|uv| \leqslant (1 + 2/s) \cdot |uy|$,

(ii) $(1 - 4/s) \cdot |xy| \leqslant |uv| \leqslant (1 + 4/s) \cdot |xy|$, *and*

(iii) $|ux| \leqslant (2/s) \cdot |xy|$.

Definition 2.4 ([CK95]) *Let V be a set of n points in $\mathbb{R}^2$, and let $s > 0$ be a real number.* A well-separated pair decomposition *(WSPD) for V with respect to s is a sequence of pairs of non-empty subsets of V, $\{A_1, B_1\}, \{A_2, B_2\}, \ldots, \{A_m, B_m\}$, such that (i) A_i and B_i are well-separated with respect to s, for $i = 1, \ldots, m$, and (ii) for any two distinct points u and v of V, there is exactly one pair $\{A_i, B_i\}$ in the sequence, such that (a) $u \in A_i$ and $v \in B_i$, or (b) $v \in A_i$ and $u \in B_i$. The integer m is called the* size *of the WSPD.*

Callahan and Kosaraju [CK95] showed that a WSPD of size $O(s^2 n)$ can be computed in $O(s^2 n + n \log n)$ time.

2.3.2 A sparse interference-estimation graph

Consider the complete graph $G^\varepsilon = (V, E^\varepsilon)$, where the weights of the edges are computed as follows.

Let $\{A_i, B_i\}_{1 \leqslant i \leqslant m}$ be a well-separated pair decomposition of V with separation constant $s = 8 + 18/\varepsilon$. For each well-separated pair $\{A_i, B_i\}$ select two arbitrary points $u \in A_i$ and $v \in B_i$, and compute an $(\frac{1}{3}\varepsilon, \frac{2}{3}\varepsilon)$-valid interference estimation of $\{u, v\}$. This value is assigned to all edges $\{x, y\}$ in E^ε with $x \in A_i$ and $y \in B_i$. This assignment with interference estimations for all edges $e \in E^\varepsilon$ is denoted by $\mathrm{Int}_\varepsilon(e)$.

We will denote by $G^\varepsilon_j = (V, E^\varepsilon_j)$ the subgraph of G^ε containing all the edges of weight at most j. The following lemma shows that Int_ε correctly represents ε-valid estimations of interference.

Lemma 2.6 *Let V be a set of points and let $\{A_i, B_i\}_{1 \leqslant i \leqslant m}$ be a WSPD of V with separation constant $s = 8 + 18/\varepsilon$. Let $\widetilde{e} = \{u, v\}$ and $e = \{x, y\}$ be two edges such that $u, x \in A_i$ and $v, y \in B_i$. It holds that every $(\frac{1}{3}\varepsilon, \frac{2}{3}\varepsilon)$-valid interference estimation for $\widetilde{e}$, denoted I, is an ε-valid interference estimation for e.*

Proof. We will prove the lemma in two steps. First we show (i) that $D(x, |xy|) \subseteq D(u, (1 + \frac{1}{3}\varepsilon)|uv|)$ (and analogously $D(y, |xy|) \subseteq D(v, (1+\frac{1}{3}\varepsilon)|uv|)$) which implies that $S_1(e) \subseteq S_{1+\varepsilon/3}(\widetilde{e})$, and thus $|V \cap S_1(e)| \leqslant |V \cap S_{1+\varepsilon/3}(\widetilde{e})| \leqslant I$. Then, in the second step, we show (ii) $D(u, (1 + \frac{2}{3}\varepsilon)|uv|) \subseteq D(x, (1 + \varepsilon)|xy|)$ (and analogously $D(v, (1 + \frac{2}{3}\varepsilon)|uv|) \subseteq D(y, (1 + \varepsilon)|xy|)$) which implies that $S_{1+2\varepsilon/3}(\widetilde{e}) \subseteq S_{1+\varepsilon}(e)$, and thus $I \leqslant |V \cap S_{1+2\varepsilon/3}(\widetilde{e})| \leqslant |V \cap S_{1+\varepsilon}(e)|$. As a consequence of the two bounds we have $|V \cap S_1(e)| \leqslant I \leqslant |V \cap S_{1+\varepsilon}(e)|$ and thus I is an ε-valid interference estimation for e.

(i): Let p_x be the point on the perimeter of $D(u, (1 + \frac{1}{3}\varepsilon)|uv|)$ closest to x. It suffices to

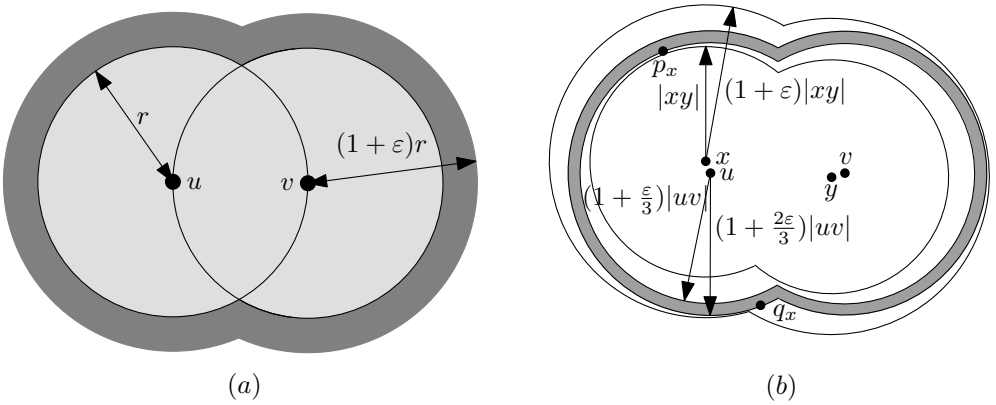


Figure 2.3: (a) The light gray area represents the (exact) interference region, that is, the sphere $S(u, v)$, while the larger region represents the estimated interference region, that is, the sphere $S_{1+\varepsilon}(u, v)$. (b) Illustration for the proof of Lemma 2.6.

show that $|xp_x| \geqslant |xy|$. We use the triangle inequality and Lemma 2.5(ii) and (iii).

$$\begin{aligned}
|xp_x| &\geqslant |p_x u| - |xu| \\
&= (1 + \frac{\varepsilon}{3}) \cdot |uv| - |xu| \\
&\geqslant (1 + \frac{\varepsilon}{3})(1 - \frac{4}{s}) \cdot |xy| - \frac{2}{s} \cdot |xy| \\
&= (1 + \frac{\varepsilon}{3} - \frac{4\varepsilon}{3s} - \frac{6}{s}) \cdot |xy| \\
&\geqslant |xy|.
\end{aligned}$$

The last inequality follows from $s = 8 + 18/\varepsilon \geqslant 4 + 18/\varepsilon$.

(ii): Let q_x be the point on the perimeter of $D(u, (1 + \frac{2}{3}\varepsilon)|uv|)$ furthest from x. It suffices to show that $|xq_x| \leqslant (1 + \varepsilon) \cdot |xy|$. We use the triangle inequality and Lemma 2.5(ii) and (iii).

$$\begin{aligned}
|xq_x| &\leqslant |q_x u| + |xu| \\
&= (1 + \frac{2\varepsilon}{3}) \cdot |uv| + |xu| \\
&\leqslant (1 + \frac{2\varepsilon}{3})(1 + \frac{4}{s}) \cdot |xy| + \frac{2}{s} \cdot |xy| \\
&= (1 + \frac{2\varepsilon}{3} + \frac{8\varepsilon}{3s} + \frac{6}{s}) \cdot |xy| \\
&\leqslant (1 + \varepsilon) \cdot |xy|.
\end{aligned}$$

The last inequality follows from the assumption that $s \geqslant 8 + 18/\varepsilon$. ♣

Next we define the edge set $\widetilde{E}^{\varepsilon}$ as follows. For each well-separated pair $\{A_i, B_i\}$ select exactly one pair of points u, v such that $u \in A_i$ and $v \in V_i$. The edge $\{u, v\}$ is added to $\widetilde{E}^{\varepsilon}$ setting its weight to be the ε-valid estimation $\mathrm{Int}_{\varepsilon}(\{u, v\})$. Note that the number of edges in E^{ε} is $n(n-1)/2$ while the number of edges in $\widetilde{E}^{\varepsilon}$ is bounded by $O(n/\varepsilon^2)$, that is, the number of well-separated pairs.

MinEstimatedInterferenceNetwork$(V, \mathcal{P})$

$\widetilde{G}^\varepsilon \leftarrow (V, \widetilde{E}^\varepsilon)$
$upper \leftarrow 1/4$
repeat
  $upper \leftarrow 2 \cdot upper$
  $\widetilde{E}^\varepsilon_{upper} \leftarrow$ all edges in $\widetilde{E}^\varepsilon$ with weight at most $upper$
  $\widetilde{G}^\varepsilon_{upper} \leftarrow (V, \widetilde{E}^\varepsilon_{upper})$
until *FulfillsProperty*$(\widetilde{G}^\varepsilon_{upper}, \mathcal{P})$
$lower \leftarrow \lfloor upper/2 \rfloor$
// binary search
while $upper > lower + 1$ **do**
  $middle \leftarrow \frac{1}{2}(lower + upper)$
  $\widetilde{E}^\varepsilon_{middle} \leftarrow$all edges in $\widetilde{E}^\varepsilon$ with weight at most $middle$
  $\widetilde{G}^\varepsilon_{middle} \leftarrow (V, \widetilde{E}^\varepsilon_{middle})$
  if *FulfillsProperty*$(\widetilde{G}^\varepsilon_{middle}, \mathcal{P})$ **then**
    $upper \leftarrow middle$
  else $lower \leftarrow middle$
return G^ε_{upper}

Figure 2.4: The basic algorithm for the estimated model.

2.3.3 Computing minimum-estimated-interference networks

We will use the same approach as we used in Section 2.2, with one difference, instead of using the graphs G_j we will use the graphs G^ε_j. Actually we will go one step further, instead of using G^ε_j we will use $\widetilde{G}^\varepsilon_j = (V, \widetilde{E}^\varepsilon_j)$.

The general algorithm for finding a minimum-interference network based on estimated interferences is given in Algorithm 2.4. Next we analyze the running time of the general algorithm and then prove the correctness for each of the three properties that we consider.

Lemma 2.7 *The graph $\widetilde{G}^\varepsilon = (V, \widetilde{E}^\varepsilon)$ can be constructed in $O(n/\varepsilon^2(\log n + 1/\varepsilon))$ time.*

Proof. Initially set $\widetilde{E}^\varepsilon$ to be empty. Construct a well-separated pair decomposition with respect to $s = 8 + 18/\varepsilon$ in $O(n/\varepsilon^2 + n \log n)$ time. For each well-separated pair $\{A_i, B_i\}$ select an arbitrary pair of points $u \in A_i$ and $v \in B_i$. Perform an ε'-approximate range counting query [AM00] with $S_{1+\varepsilon/2}(u, v)$ as the query range, where $\varepsilon' = \frac{\varepsilon}{12+6\varepsilon}$. The number of reported points (minus u and v) determines an estimated interference between $|V \cap S_{1+\varepsilon/3}(u, v)|$ and $|V \cap S_{1+2\varepsilon/3}(u, v)|$ and hence an ε-estimated interference for every edge in E^ε, according to Lemma 2.6. Since the query range has constant complexity, each such query can be answered in $O(\log n + 1/\varepsilon)$ time with the BBD-tree by Arya and Mount [AM00] (following the analysis by Haverkort et al. [HdBG04]). In total this requires $O(n/\varepsilon^2(\log n + 1/\varepsilon))$ time since $O(n/\varepsilon^2)$ queries are performed. ♣

Theorem 2.2 *Algorithm* MinEstimatedInterferenceNetwork *runs in $O(n/\varepsilon^2(\log n + 1/\varepsilon) + P(n, n/\varepsilon^2) \log k)$ time, where n is the number of nodes, k is the maximum*

ε-valid estimated interference of any edge in the network output, and $P(n, m)$ is the running time of FulfillsProperty on a graph with n nodes and $O(m)$ edges.

Proof. The graph $\widetilde{G}^\varepsilon = (V, \widetilde{E}^\varepsilon)$ is constructed in $O(n/\varepsilon^2(\log n + 1/\varepsilon))$ time. The exponential and binary search is iterated at most $\lceil \log k \rceil$ times. Each time the edge set $\widetilde{E}^\varepsilon_j$ is computed in $O(n/\varepsilon^2)$ time by looking at all edges in $\widetilde{E}^\varepsilon$. The graph is tested for the desired property in $O(P(n, n/\varepsilon^2))$ time. Summing up the time bounds gives a total of $O(n/\varepsilon^2(\log n + 1/\varepsilon) + P(n, n/\varepsilon^2) \log k)$ time. ♣

Before we study the three desirable properties (connectivity, bounded dilation, and bounded link diameter) we need two basic properties of the graph $\widetilde{G}^\varepsilon$.

2.3.4 Two properties of the sparse graph

For technical reasons we have to ensure that the interference estimations of all shortest edges in G^ε are zero. This can be accomplished during the processing of the well-separated pair decomposition without requiring any extra time.

In Theorem 2.3 below we show that $\widetilde{G}^\varepsilon_j$ closely approximates G^ε_j. For the proof we require the following technical lemma.

Lemma 2.8 *For a well-separated pair $\{A_i, B_i\}$ let $\{u, v\}$ be an arbitrary edge, where $u \in A_i$ and $v \in B_i$, with ε-valid interference estimation j. Then for every edge $\{x, y\}$ with $x, y \in A_i$ or $x, y \in B_i$ it holds that every ε-valid interference estimation of $\{x, y\}$ is at most j.*

Proof. The distance between x and y is bounded by $2/s \cdot |uv|$ according to Lemma 2.5. By the choice of s this is less than $\varepsilon/9 \cdot |uv|$ and thus, for reasonably small values of ε, it holds that $S_{1+\varepsilon}(x, y) \subseteq S(u, v)$, see Figure 2.5a. Hence, even if $\{u, v\}$ attains its minimum possible ε-valid interference estimation j, each ε-valid interference estimation of (x, y) is at most j. ♣

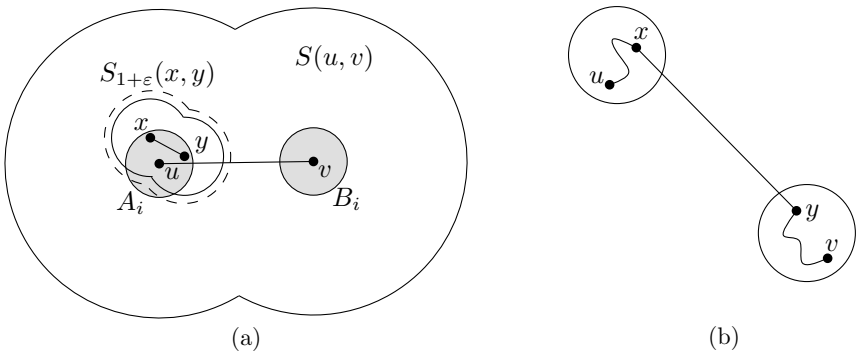


Figure 2.5: (a) Illustration for the proof of Lemma 2.8. (b) Illustration for the proof of Theorem 2.3.

Let $d_j(u, v)$ and $\widetilde{d}_j(u, v)$ denote the Euclidean length of the shortest path between u and v in G^ε_j and $\widetilde{G}^\varepsilon_j$, respectively.

Theorem 2.3 *For any $u, v \in V$ and any non-negative integer j we have $d_j(u, v) \leqslant \widetilde{d}_j(u, v) \leqslant (1 + \varepsilon) \cdot d_j(u, v)$.*

Proof. Since $\widetilde{E}_j^\varepsilon$ is a subset of E_j^ε the left inequality follows. It remains to prove the right inequality. For this it is sufficient to show that for every edge $e = \{u, v\}$ in E_j^ε there is a path P in $\widetilde{E}_j^\varepsilon$ such that $|uv| \leqslant |P| \leqslant (1+\varepsilon)|uv|$.

Sort the edges of E_j^ε with respect to their length in increasing order. We denote the edges in the sorted sequence by $e_1, \ldots, e_m$. The theorem is now proven by induction on the length of the edges.

Base cases: Consider a shortest edge $e_1 = \{u, v\}$ in E_j^ε. Our claim is that e_1 is also in $\widetilde{E}_j^\varepsilon$. From Definition 2.4 it follows that there is a well-separated pair $\{A_i, B_i\}$ with $A_i = \{u\}$ and $B_i = \{v\}$ (or $A_i = \{v\}$ and $B_i = \{u\}$). As $\{u, v\}$ is a closest pair in V, its interference estimation is zero by construction. Hence, e_1 is in $\widetilde{E}_j^\varepsilon$.

Induction hypothesis: Assume that the theorem holds for all edges of E_j^ε shorter than e_i.

Induction step: Consider the edge $e_i = \{u, v\}$, and let $\{A_i, B_i\}$ be the well-separated pair such that $u \in A_i$ and $v \in B_i$. According to the construction of $\widetilde{E}_j^\varepsilon$ there exists an edge $\{x, y\}$ in $\widetilde{E}_j^\varepsilon$ such that $x \in A_i$ and $y \in B_i$, see Figure 2.5b. For ease of explanation we assume that $\{x, y\} \cap \{u, v\} = \emptyset$ (the case of $\{x, y\} \cap \{u, v\} \neq \emptyset$ is only easier, as the rest of the proof will show).

The length of $\{x, y\}$ is at most $(1 + 4/s) \cdot |uv|$, according to Lemma 2.5. According to Lemma 2.8, the edges $\{u, x\}$ and $\{y, v\}$ are in E_j^ε. Hence, as $\{u, x\}$ and $\{y, v\}$ are shorter than e_i, there is path P_{ux} between u and x of length at most $(1+\varepsilon) \cdot |ux|$, and a path P_{yv} between y and v of length at most $(1+\varepsilon) \cdot |yv|$ in $\widetilde{G}_j^\varepsilon$ by induction hypothesis. For the u-v path $P_{ux}, \{x, y\}, P_{yv}$ in $\widetilde{G}_j^\varepsilon$ we have:

$$\begin{aligned}
\widetilde{d}(u,v) &\leqslant \widetilde{d}(u,x) + \widetilde{d}(x,y) + \widetilde{d}(y,v) \\
&\leqslant (1+\varepsilon) \cdot |ux| + \left(1 + \frac{4}{s}\right) \cdot |uv| + (1+\varepsilon) \cdot |yv| \\
&\leqslant (1+\varepsilon) \cdot \frac{2}{s}|uv| + \left(1 + \frac{4}{s}\right) \cdot |uv| + (1+\varepsilon) \cdot \frac{2}{s}|uv| \\
&= \left(1 + \frac{8}{s} + \frac{4\varepsilon}{s}\right) \cdot |uv| \\
&\leqslant (1+\varepsilon) \cdot |uv|.
\end{aligned}$$

In the third inequality we used Lemma 2.5 and in the final inequality we used that s was chosen to be $8 + 18/\varepsilon \geqslant 4 + 8/\varepsilon$. ♣

2.3.5 Minimum-estimated interference spanning trees

We start with the most basic property, namely connectivity. We will prove the corresponding estimated versions of Corollaries 2.1 and 2.2 in Section 2.2.3.

From Theorem 2.3 it immediately follows that G_j^ε is connected if and only if $\widetilde{G}_j^\varepsilon$ is connected. Thus a simple approach to test the connectivity is to use a breadth-first search in $\widetilde{G}_j^\varepsilon$ which takes linear time with respect to the size of $\widetilde{E}_j^\varepsilon$. By filling in $P(n, kn) = O(n/\varepsilon^2)$ in Theorem 2.2 we get:

Corollary 2.5 *We can compute a minimum-estimated-interference connected graph in* $O(n/\varepsilon^2(\log n + 1/\varepsilon))$ *time.*

And by running Kruskal's algorithm on the graph $\widetilde{G}_k^\varepsilon$ we get:

Theorem 2.4 *We can compute a minimum-estimated-interference spanning tree $\mathcal{T}$ of V with $\sum_{e\in\mathcal{T}} \mathrm{Int}_\varepsilon(e) = \sum_{e\in\mathcal{T}_{\min}} \mathrm{Int}_\varepsilon(e)$ in $O(n/\varepsilon^2(\log n + 1/\varepsilon))$ time, where $\mathcal{T}_{\min}$ is a minimum spanning tree of G_k^ε with respect to Int_ε.*

Proof. Running Kruskal's algorithm [Kru56] on $\widetilde{G}_k^\varepsilon$ takes $O(n/\varepsilon^2 + n\log n)$ time and thus the running time is dominated by the computation of $\widetilde{G}_k^\varepsilon$ according to Corollary 2.5.

It remains to show that $\widetilde{G}_k^\varepsilon$ contains a spanning tree $\mathcal{T}$ of cost equal to the cost of $\mathcal{T}_{\min}$. Consider the edges in E_k^ε and order them with respect to their interference estimations. If two edges have the same interference estimation then we order them with respect to the Euclidean distance between their endpoints. For simplicity, let $\mathcal{T}_{\min}$ denote the minimum spanning tree obtained by running Kruskal's algorithm using this ordering. Let $\{u_1, v_1\}, \{u_2, v_2\}, \ldots, \{u_{n-1}, v_{n-1}\}$ be the edges of $\mathcal{T}_{\min}$ in the order in which they were added to $\mathcal{T}_{\min}$.

Consider the subgraph $\widetilde{F}$ of $\widetilde{G}_k^\varepsilon$ constructed as follows: for each edge $\{u_j, v_j\}$ in $\mathcal{T}_{\min}$, find the well-separated pair (A_i, B_i) such that $u_j \in A_i$ and $v_j \in B_i$, and add the edge $\{x, y\} \in \widetilde{G}_k^\varepsilon$ with $x \in A_i$ and $y \in B_i$ to $\widetilde{F}$. Clearly the cost of $\widetilde{F}$ is equal to the cost of $\mathcal{T}_{\min}$, since for every edge $\{u, v\}$ in $\mathcal{T}_{\min}$ an edge $\{x, y\}$ is added to $\widetilde{F}$ whose interference estimation equals the estimation of $\{u, v\}$. It remains to prove that $\widetilde{F}$ is a spanning tree.

We will prove the following statement:

For each edge $\{u_j, v_j\}$ in $\mathcal{T}_{\min}$ there is a path in $\widetilde{F}$ between u_j and v_j. (*)

Since $\widetilde{F}$ contains $n-1$ edges, it follows that $\widetilde{F}$ must be a spanning tree. Our proof of (*) is by induction over the index j of the edges in $\mathcal{T}_{\min}$.

Base case: The first edge added to $\mathcal{T}_{\min}$ is $\{u_1, v_1\}$. By construction, $\{u_1, v_1\}$ is the shortest edge in $\binom{V}{2}$ and its interference estimation is zero. Then the well-separated pair that contains it must be a pair (A_i, B_i) with $A_i = \{u_1\}$ and $B_i = \{v_1\}$ and hence, $\{u_1, v_1\}$ is also in $\widetilde{F}$.

Induction hypothesis: Assume that condition (*) holds for all edges $\{u_1, v_1\}, \ldots, \{u_{i-1}, v_{i-1}\}$ in $\mathcal{T}_{\min}$.

Induction step: Let $\{x_i, y_i\}$ be the edge in $\widetilde{F}$ corresponding to $\{u_i, v_i\}$, and let $\{A_i, B_i\}$ be the well-separated pair such that $u_i, x_i \in A_i$ and $v_i, y_i \in B_i$. Our claim is that there is a path between u_i and x_i, and a path between v_i and y_i, only containing edges from the set $\{\{u_1, v_1\}, \ldots, \{u_{i-1}, v_{i-1}\}\}$. According to Lemma 2.8 an interference estimation of the edge $\{u_i, x_i\}$ does not exceed the interference estimation of the edge $\{u_i, v_i\}$, and obviously $\{u_i, x_i\}$ is shorter than $\{u_i, v_i\}$. Thus $\{u_i, x_i\}$ was considered before $\{u_i, v_i\}$ and either it was added to $\mathcal{T}_{\min}$, or u_i and x_i were already connected in $\mathcal{T}_{\min}$ and adding $\{u_i, x_i\}$ would have created a cycle in $\mathcal{T}_{\min}$ (at the time when $\{u_i, x_i\}$ would have been inserted according to the order). In both cases it follows that u_i and x_i are connected in $\mathcal{T}_{\min}$ by a path that only contains edges from the set $\{\{u_1, v_1\}, \ldots, \{u_{i-1}, v_{i-1}\}\}$, and thus, by the induction hypothesis, they are also connected in $\widetilde{F}$. The same argument yields a path in $\widetilde{F}$ between v_i and y_i. This completes the proof of (*). ♣

2.3.6 Minimum-estimated-interference t-spanners

We use the same approach as in Section 2.2.4 and apply Dijkstra's algorithm to compute the dilation in $\widetilde{G}_j^\varepsilon$. This takes $O(n^2/\varepsilon^2 + n^2 \log n)$ time. Note that according to Theorem 2.3, $\widetilde{G}_j^\varepsilon$ is at least an $(1+\varepsilon)t$-spanner if G_j^ε is a t-spanner. Thus, we implement the answer of the routine *FulfillsProperty* to be 'yes' if and only if $\widetilde{G}_j^\varepsilon$ is an $(1+\varepsilon)t$-spanner. This only implies that G_j^ε is an $(1+\varepsilon)t$-spanner, but the interference of G_j^ε is at most the minimum-estimated interference of a t-spanner with respect to Int_ε. Thus, $P(n, n/\varepsilon^2) = O(n^2/\varepsilon^2 + n^2 \log n)$ and by applying Theorem 2.2 we obtain:

Corollary 2.6 *In $O(n^2 \log k(1/\varepsilon^2 + \log n))$ time we can compute a $(1+\varepsilon)t$-spanner of V with estimated interference at most the minimum-estimated interference of any t-spanner of V.*

2.3.7 Minimum-estimated-interference d-hop networks

When one wants to construct a d-hop network with minimum estimated interference value it seems hard to avoid studying all the edges in E_j^ε. A simple way to check if a graph is a d-hop network is to perform a breadth-first search (BFS) in G_j^ε from each of the nodes. The problem is that the running time is linear in the number of edges, thus running the n searches in G_j^ε would give a cubic running time. Also, we cannot perform the BFS in $\widetilde{G}_j^\varepsilon$ since $\widetilde{G}_j^\varepsilon$ does not approximate G_j^ε when it comes to the number of hops. We will show, however, that for every G_j^ε we can perform an implicit breadth-first search in a graph $\mathcal{T}$ that has only $O(n/\varepsilon^2)$ edges.

The graph $\mathcal{T}$ is the split tree [CK95] $\mathcal{S}$ that is built in the first step of Algorithm 2 in order to compute the WSPD, augmented with a number of non-tree edges. In particular, $\mathcal{S}$ is a binary tree in which each vertex corresponds to a subset of V. We identify each vertex with its corresponding point set. The root of $\mathcal{S}$ is V. For any non-leaf vertex $\mathcal{A} \in \mathcal{S}$ and its two children $\mathcal{A}_1, \mathcal{A}_2$ it holds that $\mathcal{A}_1 \cap \mathcal{A}_2 = \emptyset$ and $\mathcal{A}_1 \cup \mathcal{A}_2 = \mathcal{A}$. All leaves correspond to single points and for every well-separated pair $\{A_i, B_i\}_{1 \leqslant i \leqslant m}$ there is exactly one vertex $\mathcal{A} \in \mathcal{S}$ with $\mathcal{A} = A_i$ and one vertex $\mathcal{B} \in \mathcal{S}$ with $\mathcal{B} = B_i$. (However, not all vertices in $\mathcal{S}$ correspond to a set A_i or B_i.) To avoid confusion we will always refer to the nodes of $\mathcal{T}$ as vertices. For a given vertex $\mathcal{A}$, our algorithm needs to access all vertices $\mathcal{B}$ such that $\{\mathcal{A}, \mathcal{B}\}$ corresponds to a pair $\{A_i, B_i\}$ of the WSPD. Therefore we extend $\mathcal{S}$ with edges between all pairs of vertices that correspond to pairs of the WSPD. This extended version of $\mathcal{S}$ constitutes the graph $\mathcal{T}$ that we will use for our breadth-first search. The non-tree edges are distinguishable from the tree edges and can be inserted without additional cost when computing the WSPD. Observe that the number of tree edges is at most $2(n-1)$ while the number of edges that join well-separated sets is $O(n/\varepsilon^2)$. Therefore $\mathcal{T}$ has $O(n/\varepsilon^2)$ edges.

The idea is now to perform n implicit BFS's, one for each point $x \in V$. The value $\mathrm{MaxHops}(x)$ obtained by the implicit BFS for x is the depth of a BFS-tree of G_j^ε with root x. After performing all implicit BFS's, we have access to the link diameter of G_j^ε which is $\max_{x \in V} \mathrm{MaxHops}(x)$.

We will first give the algorithm to compute $\mathrm{MaxHops}(x)$ and then prove its correctness. The algorithm is essentially the same as the one by Gudmunsson et al. [GNS03] but slightly modified to fit our setting. It computes a breadth-first forest consisting of breadth-first trees rooted at all vertices of $\mathcal{T}$ which contain x, these are exactly x itself and all its ancestors in the split tree. As usually the breadth-first search features a queue Q of vertices that still have

to be processed and it terminates as soon as Q is empty. For each vertex $\mathcal{A}$ of the split tree, the algorithm maintains two variables:

- $color(\mathcal{A})$, whose value is either *white*, *gray*, or *black*, and
- $dist(\mathcal{A})$, whose value is the minimum distance to get from x to at least one point of $\mathcal{A}$ in G_j^ε (as computed by the algorithm).

The algorithm will maintain the following invariants: All vertices that are white have not been visited yet. The vertices that are grey have been visited and are stored in the queue Q. All vertices $\mathcal{A}$ that are black have been processed already and $dist(\mathcal{A})$ contains the correct distance from x.

Step 1: For each vertex $\mathcal{A} \in \mathcal{T}$ set $color(\mathcal{A}) := white$ and $dist(\mathcal{A}) := \infty$.

Step 2: Initialize an empty queue Q. Starting at the leaf of $\mathcal{T}$ storing x, walk up the tree to the root. For each vertex $\mathcal{A}$ encountered, set $color(\mathcal{A}) := gray$ and, if $\mathcal{A}$ corresponds to a set in a well-separated pair, set $dist(\mathcal{A}) := 0$ and append $\mathcal{A}$ to the end of Q.

Step 3: If Q is empty then go to Step 4. Otherwise, let $\mathcal{A}$ be the first element of Q. Delete $\mathcal{A}$ from Q and set $color(\mathcal{A}) := black$. For each well-separated pair $\{A_i, B_i\}$ for which $A_i = \mathcal{A}$ (or $B_i = \mathcal{A}$) and whose edges have interference estimations of at most j, let $\mathcal{B}$ be the vertex in $\mathcal{T}$ with $\mathcal{B} = B_i$ ($\mathcal{B} = A_i$). If $\mathcal{B}$ is white do the following:

Step 3.1: Starting at vertex $\mathcal{B}$, walk up the split tree until the first non-white vertex is reached. For each white vertex $\mathcal{A}'$ encountered, set $color(\mathcal{A}') := gray$ and, if $\mathcal{A}'$ corresponds to a set in a well-separated pair, set $dist(\mathcal{A}') := dist(\mathcal{A}) + 1$ and add $\mathcal{A}'$ to the end of Q.

Step 3.2: Visit all vertices in the subtree of $\mathcal{B}$. For each vertex $\mathcal{A}'$ in this subtree, set $color(\mathcal{A}') := gray$ and, if $\mathcal{A}'$ corresponds to a well-separated set, set $dist(\mathcal{A}') := dist(\mathcal{A}) + 1$ and add $\mathcal{A}'$ to the end of Q.

After all such vertices $\mathcal{B}$ have been processed, go to Step 3.

Step 4: Return $\text{MaxHops}(x) = \max\{\text{dist}(\mathcal{A}) \mid \mathcal{A} \in \mathcal{T} \text{ is a set in a well-separated pair}\}$.

Observe that, if $\mathcal{A}'$ is the first non-white vertex reached in Step 3.1, all vertices on the path from $\mathcal{A}'$ to the root of the split tree are non-white. Also, if $color(\mathcal{B}) = white$, then all vertices in the subtree of $\mathcal{B}$ (these are visited in Step 3.2) are white.

To estimate the running time of the algorithm, we first note that Step 1 as well as Step 2 takes $O(n)$ time. The total time for Step 3 is proportional to the number of edges in $\mathcal{T}$ and the total time for walking through $\mathcal{T}$ in Steps 3.1 and 3.2. It follows from the algorithm that each edge of $\mathcal{T}$ is traversed at most once. Therefore, Step 3 takes $O(n/\varepsilon^2)$ time, which is also the total running time of the BFS from x. We now prove its correctness.

Theorem 2.5 *The value returned by the above algorithm is* $\text{MaxHops}(x)$.

Proof. We show (*): for each $\mathcal{A} \in \mathcal{T}$ that corresponds to a set in a well-separated pair, $dist(\mathcal{A})$ is the minimum distance to get from x to at least one point of $\mathcal{A}$ in G_j^ε. This proves the claim as all nodes V are stored in a leaf of $\mathcal{T}$ and correspond to a well-separated set, thus the set in Step 4 over which the maximum is taken contains all the distances from x to every other node in G_j^ε and no bigger values than that.

Note that, besides initializing with ∞, $dist(\mathcal{A})$ is set only once for each $\mathcal{A}$, namely when $\mathcal{A}$ is encountered for the first time. We show the following two invariants: (i) when $dist(\mathcal{A})$ is set, there is at least one node in $\mathcal{A}$ that is at most $dist(\mathcal{A})$ steps away from x in G_j^ε and (ii) no vertex in $\mathcal{A}$ can be reached from x by less than $dist(\mathcal{A})$ steps. This proves (*). Step 2 sets $dist(\mathcal{A})$ to zero for all $\mathcal{A} \in \mathcal{T}$ that contain x and thus fulfills the two invariants for these vertices.

Next, we show that invariant (i) holds for Step 3.1 and Step 3.2. Let $\{A_i, B_i\}$ be the well-separated pair that caused the calls of Step 3.1 and Step 3.2 and let $\mathcal{B} = B_i$. Since $dist(A_i) = dist(B_i) - 1$, we can prove by induction that the invariant holds for A_i and thus there is at least one point $a \in A_i$ that can be reached from x in $dist(\mathcal{B}) - 1$ steps. However, as the interference estimations of all edges between A_i and B_i is at most j this means that in G_j^ε all points in $\mathcal{B}$ can be reached from x via a in $dist(\mathcal{B})$ steps. This shows invariant (i) for Step 3.1 as all vertices on the path from $\mathcal{B}$ to the first non-white ancestor contain supersets of the point set that corresponds to $\mathcal{B}$. It also shows invariant (i) for Step 3.2 as the vertices in the subtree of $\mathcal{B}$ contain subsets of the point set that corresponds to $\mathcal{B}$.

We prove that invariant (ii) holds throughout the algorithm by contradiction. Let $\mathcal{B} \in \mathcal{T}$ be the first vertex encountered for which (ii) is violated. This means that there exists a path $e_1, \ldots, e_\ell \in G_j^\varepsilon$ connecting x and a node $v \in \mathcal{B}$ with $\ell < dist(\mathcal{B})$ edges. Let $e_\ell = \{u, v\}$ and $\{A_i, B_i\}$ be the unique well-separated pair with $u \in A_i$ and $v \in B_i$. The path $e_1, \ldots, e_{\ell-1}$ connects x with $u \in A_i$ by $\ell - 1$ edges. Since $\mathcal{B}$ is a minimal counterexample, invariant (ii) holds for A_i which means that $dist(A_i) \leqslant \ell - 1$. This is a contradiction because then $dist(\mathcal{B}) \leqslant dist(A_i) + 1 = \ell$ as again all edges between A_i and B_i are in G_j^ε. ♣

Recall that we can implement the function *FulfillsProperty(G_j^ε,"has link-diameter at most d")* as follows. We augment the split tree $\mathcal{S}$ of the WSPD with the $O(n/\varepsilon^2)$ edges that correspond to well-separated sets whose representative edges have interference estimation of at most j. Then we run the above algorithm once for each node in V. We have $P(n, n/\varepsilon^2) = O(n^2/\varepsilon^2)$ and by applying Theorem 2.2 we get:

Corollary 2.7 *Given a set V of n points in the plane and an integer d, one can compute a d-hop network with minimum estimated interference value in $O(n/\varepsilon^2(n \log k + 1/\varepsilon))$ time.*

2.4 Generalizations and extensions

In this section we consider different models and different interference functions, and we discuss the possibility to modify the presented algorithms to handle these cases.

Bounded transmission radius: Burkhart et al. [BvRWZ04] and Moaveni-Nejad and Li [ML05] considered the case where each transmitter has a bounded transmission radius. The algorithms in Section 2.2 can be modified to handle this case. For the estimated interference model our algorithms only work if for every well-separated set $\{A, B\}$ there is indeed a point pair u, v with $u \in A$, $v \in B$ which are allowed to communicate. This ensures that the sparse graph is connected. In practice the above restriction seems to make sense as otherwise communication between stations u and v out of a well-separated set for which any communication is forbidden has to be routed on graph paths that make large detours concerning the bee-line distance of u and v.

Weighted points: In the scenario in which the points are weighted, the weights can be thought of as a measure of importance assigned to the nodes. Points with a very high weight are very important and thus interference with those nodes should be avoided. The changes to the algorithms in Section 2.2 are trivial, since range counting queries can easily be modified to handle weighted points. In the estimated interference model the same bounds also hold for the weighted case. The only difference is in computing the interference value of an edge – fortunately the BBD-tree [AM00] can also be used for weighted range queries without additional preprocessing or query time.

Chapter 3

Boundary Labeling with One-Bend Leaders

As in the two previous chapters the geometric aspect is given by the input: n points contained in a rectangle R. Here, a network will not be involved in the strong sense. However, the points and labels together with the determined assignment from labels to points can be interpreted as a bipartite graph. The optimization is done on the number of bends that are contained in the edge set of the graph and on the total edge length. Both are minimized in order to get a high-quality labeling.

I have done main parts of this research together with Herman Haverkort, Mira Lee and Martin Nöllenburg on the Fifth Korean Workshop on Computational Geometry. An abstract entitled "Improved algorithms for length-minimal one-sided boundary labeling" has been accepted on the *23rd European Workshop on Computational Geometry* [4]. Furthermore, an extended version entitled "Algorithms for multi-criteria one-sided boundary labeling" which is based on ideas of this chapter has been accepted at the *15th International Symposium on Graph Drawing* and will be published soon [5].

3.1 Introduction

Presentation of visual information often makes use of textual labels for features of interest within the visualization. Well-known examples stem from diverse areas such as cartography, anatomy, engineering, sociology etc. Visualizations in these applications all have in common that a graphic may have very dense regions that need to be enriched by textual information in order to convey its full meaning. A lot of research on automatic label placement has concentrated on placing labels as close to the features of interest as possible, see the extensive bibliography on map labeling by Wolff and Strijk [WS96]. However, there are situations in which this is impossible or undesirable. Such situations can occur if the labels are relatively large (for example a block of text rather than a single word), if the labeled features lie so close to each other that there is not enough space to place the labels, or if the full figure should remain visible, without parts of it being occluded or cluttered by labels. Geographic maps that depict metropolitan areas and medical atlases are examples were these situations occur. A reasonable alternative for static displays is to place the labels next to the actual illustration and connect each label to its feature by a curve. This is called a *call-out*, and the curves are called *leaders*. An example is shown in Figure 3.1. To produce a call-out, we

have to decide where exactly to place each point's label and how to draw the curves such that the connections between points and labels are clear and the curves do not clutter the figure. Clearly, leaders should not intersect each other to avoid confusion.

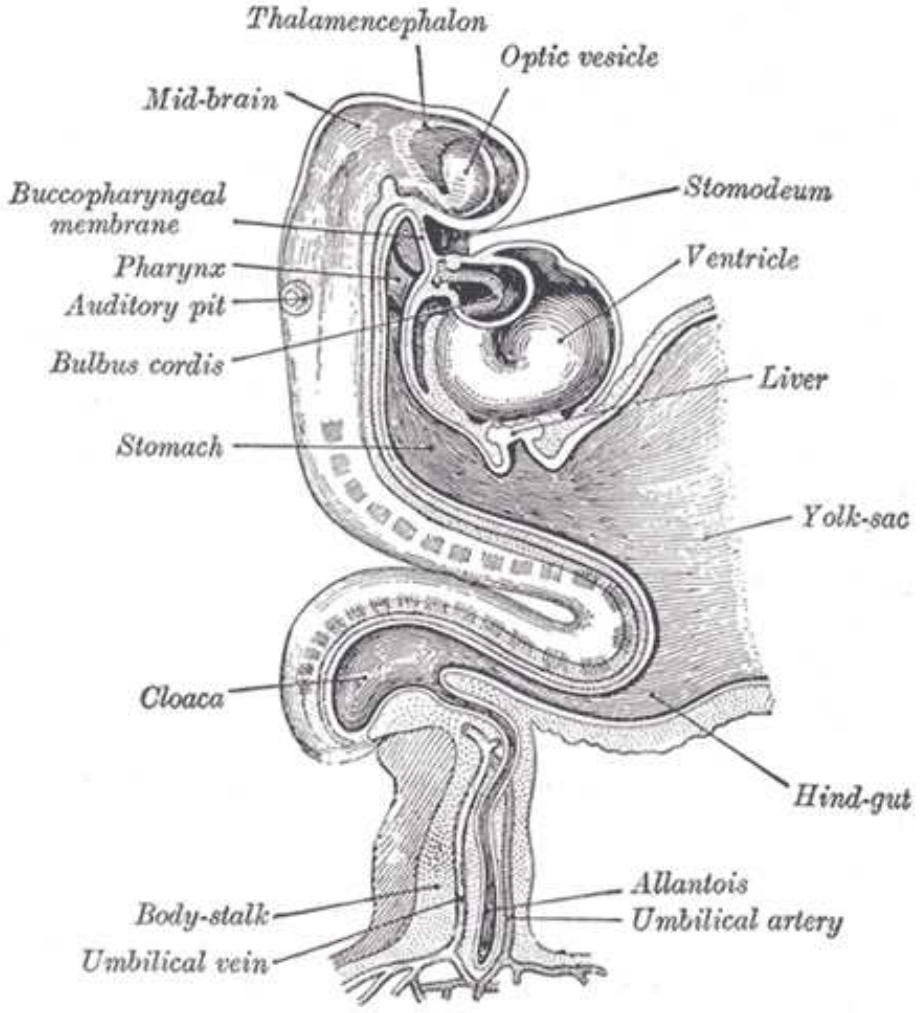


Figure 3.1: Anatomy of a human embryo [Gra18].

Recently, interest in algorithms for this problem has grown. Fekete and Plaisant [FP99] described an interactive labeling technique that uses leaders. A set of points in a focus area around the cursor position is labeled to the left and to the right of the focus using non-intersecting polygonal leaders with up to two bends. Points not in the focus remain unlabeled. Ali et al. [AHS05] describe several heuristics for assigning label positions in a call-out to the points being labeled, using straight-line and rectilinear leaders. They first compute an initial labeling and subsequentially eliminate intersections between leaders. Bekos et al. [BKSW07] insist that it is desirable to keep the leaders short and avoid unnecessary bends. Therefore they define the problem as an optimization problem where either the total number of bends or the total length of the leaders is to be minimized under the condition that leaders are not allowed to intersect. They consider a set of points contained in an axis-aligned rectangle R. The labels are placed around R and connected to the corresponding points by non-intersecting rectilinear

leaders. Variants with labels on one, two, or four sides of R are considered. The focus of their work is on efficient algorithms for minimizing the total leader length; the problem of finding efficient algorithms to minimize the number of bends is mostly left open. In subsequent works Bekos et al. study variations where labels are arranged in multiple columns on one side of the rectangle [BKPS06a] or where a set of polygons is labeled instead of a set of fixed points [BKPS06b]. The latter is motivated by the fact that features in an illustration often extend over an area and are not just single points.

One of the most important aspects determining the readability of a boundary labeling is the type of the leaders that is used. Surprisingly, it is not always the best choice to rely exclusively on straight-line leaders as in Figure 3.1. The reason is that the variety of different slopes among the leaders may unnecessarily clutter the visualization, especially if the number of labels is large. Leaders look less disturbing if their shape is more uniform and a small number of slopes is used. At the same time leaders appear easier to follow if their bends are smooth. In all work cited above, leaders are either straight lines or polylines whose segments may have arbitrarily many different slopes, or they are rectilinear, which means that they have rather sharp bends.

In this work we follow the problem definition of Bekos et al. [BKSW07] and aim to reduce the number of sharp bends in two ways. First, we introduce leaders that can also consist of diagonal segments of a fixed angle. This leads to smoother bends and leaders that are visually easier to follow. Secondly, in addition to algorithms that minmize the total length of the leaders, we provide algorithms that minimize the number of bends, both for rectilinear leaders and for leaders with diagonal segments. To the best of our knowledge there is no literature that deals with diagonal leaders algorithmically.

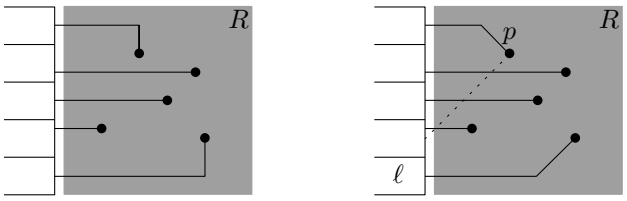
In the following sections we describe the problem more precisely and give algorithms for labels placed on one side of the figure and for labels placed on two sides of the figure. In the last section we compare the effects of the leader type and the optimization criterion on the basis of an example. For practical issues, we come to the conclusion that a hybrid method that simultaneously minimizes the leader length and the number of bends may be the method of choice.

3.2 Problem definition

We are given a set P of n points, a rectangle R that contains P, and n disjoint rectangles (labels) of equal width that border R on the left side or on the right side. The points are to be matched with the labels and the matching shall be indicated by a polygonal line–the leader—connecting the point with its corresponding label.

We consider two types of leaders. A *po-leader* for a point q consists of two segments: a vertical segment ('p'arallel to the labeled sides of R) that starts at q, followed by a horizontal segment ('o'rthogonal to the labeled sides of R) that ends at the corresponding label (the terminology is from Bekos et al. [BKSW07]). Furthermore we introduce *do-leaders*, that start with a 'd'iagonal line segment and end in a horizontal segment. The diagonal segments are always directed towards the side of R where the label is, and have a fixed angle with the x-axis—for simplicity we assume this angle to be fixed at 45°. However, our algorithms are applicable for any angle between 0° and 90° degree. Both leader types are illustrated in Figure 3.2.

A point is labeled *directly* if its leader is a single line segment, that is, if only the horizontal



(a) *po*-leaders. **(b)** *do*-leaders.

Figure 3.2: Valid labelings for labels on the left side.

segment has non-zero length, otherwise it is labeled by a leader that contains a bend, from now on called *bend leader*.

A labeling is a matching of points to labels and a selection of leaders that connects each point to its label. A labeling is *valid* if every point has a unique label and every label is connected to a unique point, and no two leaders intersect.

Our goal is to find a valid labeling with the minimum number of bends (that is, with the maximum number of directly labeled points), or with the minimum total leader length. The problem type of minimizing the number of bends is, for all variants, accessible to dynamic programming. Minimizing the total leader length can obviously be done by the same technique if we change the way in which the costs of the solutions of subproblems are computed. However, for minimizing the leader length in most variants sweepline-based algorithms can be used to improve upon the corresponding dynamic-programming solution.

In Section 3.3 we present algorithms for the variants in which labels are located only on one side of the point-containing rectangle R and in Section 3.4 we present algorithms where labels are located on two opposite sides of R.

For simplicity we assume throughout this chapter that the label rectangles are uniform (our techniques can easily be extended to non-uniform label rectangles at fixed positions along the labeled sides of R). For the one-sided case we assume that the labels are located on the left side of R. For simplicity we further assume that no two input points lie on a horizontal line induced by a label and that for the *po*-leaders no two input points lie on a vertical line and for the *do*-leaders no two input points lie on a diagonal line.

3.3 Algorithms for labels on one side

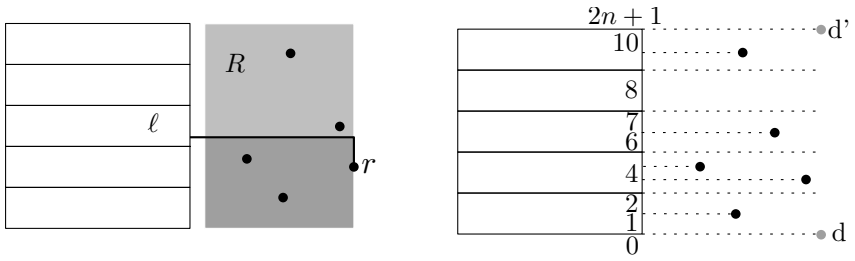
We first explain in Section 3.3.1 how to find a labeling with the minimum number of bends. For the *po*-leaders this can be done in $O(n^3)$ time. Obviously there always exists a valid *po*-leader labeling since any point can connect to any label by a *po*-leader. Then, we explain how to extend our approach to an algorithm for *do*-leaders that runs in $O(n^5)$ time. Apparently, *do*-leaders are harder to handle since for them, there are points that cannot connect to any label. See Figure 3.2b where no *do*-leader can connect p and ℓ, indicated by the dashed line.

In Section 3.3.2 we give sweepline-based algorithms that generate length-minimum labelings using *po*- and *do*-leaders. For *po*-leaders our algorithm requires $O(n \log n)$ time improving the algorithm of Bekos et al. [BKSW07] that had quadratic running time. For *do*-leaders our algorithm requires $O(n^2)$ time.

3.3.1 Bend minimization

po-leaders

The dynamic program is based on the following idea. Choose a label ℓ for the rightmost point r. Then, the leader from r to ℓ splits the problem into two independent subproblems that can be solved recursively, see Figure 3.3a.



(a) Split into two independent subproblems.

(b) Numbering of the strips and the points r, d and d'.

Figure 3.3: The idea for the dynamic programming.

Combinatorially, there are $2n$ different positions in which the leader of r can connect to a label. These $2n$ positions are induced by the horizontal lines through each point and through each horizontal side of the label rectangles, see Figure 3.3b. We identify these positions with the strips between two consecutive horizontal lines and number them from 1 to $2n$ from bottom to top. For the boundary cases of the dynamic program we give the halfplane below R number 0 and the halfplane above R number $2n + 1$, and introduce two dummy points d and d' that lie on the right corners of R, see Figure 3.3b.

A subproblem is now specifiecd by two parameters *bot* and *top* that define strips from the set $\{0, 1, \ldots, 2n, 2n+1\}$. These parameters uniquely define a subproblem as follows. Let n_{sub} be the number of labels between, but not including, the labels incident to *bot* and *top*. These n_{sub} labels are the *open labels* for the specified subproblem, that is, the labels that need to be assigned to a point. Let R_{sub} be the closed strip between the upper horizontal line incident to *bot* and the lower horizontal line incident to *top*, see Figure 3.4. The leftmost n_{sub} points in R_{sub} (and on its boundary) are the points that need to be assigned to a label and belong to the subproblem.

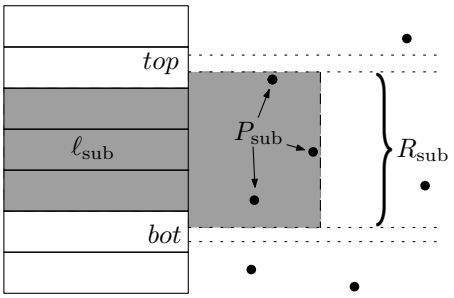


Figure 3.4: The subproblem defined by *bot* and *top*.

We will store the number of bends of the optimal labelings in a table T. A table entry

$T[bot, top]$ gives the minimum number of bends for an optimal labeling of the corresponding subproblem. The entry $T[0, 2n+1]$ characterizes the whole input and hence gives the minimum number of bends in a valid labeling of the complete input. The labeling itself is obtained by keeping track of the subproblems whose solutions were used in the optimal solution.

To compute a solution to the complete problem, we fill the table T recursively in a top-down fashion.

If no open label remains between *bot* and *top* we set $T[bot, top] = 0$.

To solve a specific subproblem $[bot, top]$ that has at least one open label, we try all possible choices for the leader of the rightmost point r in the subproblem. Note that not every choice of a position for the leader of r results in a *feasible* split in the sense that both resulting subproblems actually have a valid labeling. In the example of Figure 3.3b only positions $1, 3, 4, 6, 8$ and 10 are feasible for the leader of r. The remaining positions $2, 5, 7$ and 9 split the problem into two subproblems in which the number of open labels does not match the number of points that still have to be labeled. Such subproblems do not contribute any information to the global optimum labeling and thus we do not compute their values. We restrict our choices of leaders for r to those that end in a strip $\sigma \in \{bot+1, \ldots, top-1\}$ such that a leader from r to the label at σ results in a feasible split.

We define $\delta_{r,\sigma}$ to be 0 if σ is one of the two strips incident to r (then r can be labeled directly, for example when $\sigma = 3$ or 4 in Figure 3.3b) and 1 otherwise (then r is labeled by a bend leader). The required entries are now computed by the following recursion:

$$T[bot, top] = \min_{\sigma \in \{bot+1,\ldots,top-1\} \text{ feasible}} T[bot, \sigma] + T[\sigma, top] + \delta_{r,\sigma}.$$

Theorem 3.1 *A valid one-sided bend-minimum labeling using po-leaders can be computed in $O(n^3)$ time requiring $O(n^2)$ space.*

Proof. We first sort all points once by x-coordinate and once by y-coordinate in $O(n \log n)$ time.

The table T has $O(n^2)$ entries. For determining the solution of a subproblem we first compute the points belonging to the subproblem, its rightmost point r, and all feasible splits. This can be done in $O(n)$ time by traversing the sorted lists. To compute the optimal solution for a subproblem, we have to examine at most $4n$ entries of the table, which takes $O(n)$ time.

The overall worst-case running time is thus $O(n^2)$ entries times $O(n)$ time each, which yields a total running time of $O(n^3)$.

Eventually, the dynamic program finds the optimal solution as it iterates over all valid labelings. ♣

Note that when filling T in a top-down fashion as mentioned above, many of the entries do not need to be computed since they do not belong to a feasible split.

do-leaders

Leaders of the *do*-type may provide a better readability of the figure as their shape is smoother. However, they are harder to handle as in general not every point can connect to every label by a *do*-leader as pointed out earlier. Nevertheless, we can use a dynamic programming approach very similar to the solution for *po*-leaders.

For simplicity we assume that no two points lie on a horizontal or a diagonal line and no point lies on a horizontal line induced by a label boundary. We use the previous nota-

tion for the $2n$ strips that define the combinatorially distinct leader ending positions, recall Figure 3.3b. The main difference between the solution for *do*-leaders and the solution for *po*-leaders is that we now need four parameters of linear choice to sufficiently describe a subproblem, namely two parameters that indicate the lower boundary and two that indicate the upper boundary of the subproblem.

Let $bot < top$ be positions from the strip set $\{0, 1, \ldots, 2n, 2n+1\}$ and let p and q be points from the set $P \cup \{r_{\text{bot}}, r_{\text{top}}\}$. Then, the lower boundary of the subproblem specified by p, bot, q, and top consists of the *do*-leader from point p to the label incident to position bot. The upper boundary consists of the *do*-leader from point q to the label at position top. The open labels of the subproblem are the labels between bot and top, and the points that belong to the subproblem are the points in the interior of the region bounded by the leaders mentioned above and the vertical line through $\min\{x_p, x_q\}$, see Figure 3.5. The points p and q are required for the definition of a subproblem since otherwise, together with the region boundary, the set P_{sub} would not be well-defined.

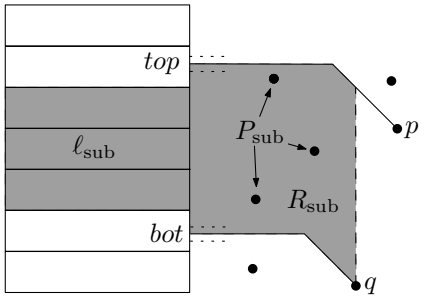


Figure 3.5: The subproblem defined by p, bot, q, and top.

As before, the table entry $T[p, bot, q, top]$ gives the minimum number of bends for leaders of points in P_{sub} in a valid labeling of the corresponding subproblem. The entry $T[r_{\text{bot}}, 0, r_{\text{top}}, 2n+1]$ includes the whole input and thus gives the minimum number of bends in a valid labeling for the complete instance.

The computation of the table entries is similar to the computation for *po*-leaders. Let r be the rightmost point in P_{sub} and let σ be a strip in $\{bot+1, \ldots, top-1\}$. This set contains all strips that are incident to an open label that can be connected to r by a *do*-leader that has its horizontal segment in σ. We say that σ is *feasible* for a subproblem if σ is incident to an open label and if the *do*-leader as described above splits the problem into two subproblems in which the number of points and open labels match. For a feasible strip σ we define $\delta_{r,\sigma}$ as for *po*-leaders.

The table is filled as for *po*-leaders, with the exception that it now can happen that a subproblem has no feasible strip. An example where this happens is depicted in Figure 3.6: for the depicted subproblem the number of open labels and points matches, but leaders from r can only reach the label below ℓ creating subproblems in which the number of open labels and points do not match. If the problem defined by p, bot, q, top has no feasible strip, we set $T[p, bot, q, top] = \infty$.

In Section 3.3.2 we will give a further characterization to detect infeasibility as early as possible but for now we are contented with the above cardinality test of labels and points. If the number of open labels in an subproblem is zero, we set $T[p, bot, q, top] = 0$.

Otherwise, we compute $T[p, bot, q, top]$ by the following recursion:

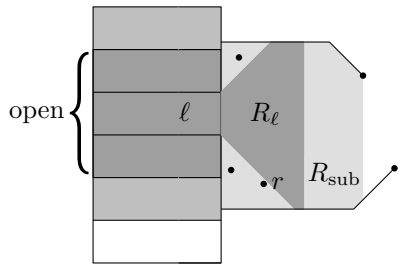


Figure 3.6: Infeasible subproblem for which the number of points and labels match.

$$T[p, bot, q, top] = \min_{\sigma \in \{bot+1,\ldots,top-1\} \text{ feasible}} T[p, bot, r, \sigma] + T[r, \sigma, q, top] + \delta_{r,\sigma}.$$

With the same analysis as before, but now with a table of size $O(n^4)$, we get the following result:

Theorem 3.2 *A valid one-sided bend-minimum labeling using do-leaders can be computed in $O(n^5)$ time requiring $O(n^4)$ space, if there is any. If not we can report infeasibility within the same time and space bounds.*

3.3.2 Length minimization

po-leaders

For the special case of minimizing the total leader length using *po*-leaders one can do better than in $O(n^3)$ time. Bekos et al. [BKSW07] have given a quadratic-time algorithm. We will give a sweepline algorithm that runs in $O(n \log n)$ time and show that this bound is tight.

Lemma 3.1 comprises the key for showing that the lablings produced by our algorithms (for *po*- and *do*-leaders) are crossing-free. We need the following notation. Let $l(p, \ell)$ denote the leader from point p to label ℓ. We say that $l(p, \ell)$ is *downwards oriented* if ℓ is below p. Conversely, $l(p, \ell)$ is a *upwards oriented* if ℓ lies above p.

Lemma 3.1 *For any labeling L^* with po- or do-leaders that may contain crossings and has minimum total leader length, there is a crossing-free labeling L whose total leader length does not exceed the total leader length of L^*. This labeling L can be constructed from L^* in $O(n^2)$ time.*

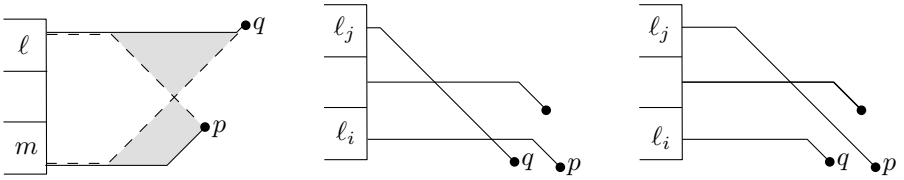
Proof.

We first observe that L^* does not contain any crossings between a downwards-oriented leader $l(p, \ell)$ and an upwards-oriented leader $l(q, m)$: it is easy to see that if such a crossing exists, we could reduce the total leader length by replacing the leaders $l(p, \ell)$ and $l(q, m)$ by $l(p, m)$ and $l(q, \ell)$. Hence, L^* would not be optimal, see Figure 3.7(a). In a similar way we can verify that in L^* no direct leader can have a crossing with a downwards-oriented leader and an upwards-oriented leader at the same time.

We call a crossing in L^* in which at least one downwards-oriented leader is involved a *downward crossing*. An *upward crossing* is defined analogously. From the above it follows that the set of leaders involved in downward crossings contains downwards-oriented leaders

and possibly direct leaders, the set of leaders involved in upward crossings contains upwards-oriented leaders and possibly direct leaders, and no leader appears in both sets.

Below we explain how L^* can be transformed into a labeling without upward crossings without increasing the total leader length and without introducing more downward crossings. The transformation can be carried out in $O(n^2)$ time. By repeating the transformation upside-down, we can subsequently eliminate all downward crossings without re-introducing upward crossings, thus obtaining a crossing-free labeling with the same total leader length as L^*.



(a) In a labeling of minimum total leader length, differently oriented leaders do not intersect.

(b) Purging upward crossings: before processing ℓ_i

(c) Purging upward crossings: after processing ℓ_i

Figure 3.7: Illustrations for the proof of Lemma 3.1.

To eliminate the upward crossings we proceed as follows. The approach follows the idea of Bekos et al. [BKSW07], generalized to leaders with any bend angle α including *do*-leaders. Let $\ell_1, \ldots, \ell_n$ be the sequence of all labels ordered from bottom to top. We process the labels in this order and make sure that all leaders ending at already processed labels are not involved in upward crossings any more. Now assume that we are about to process label ℓ_i and its leader $l(p, \ell_i)$. In $O(n)$ time we determine the leader $l(q, \ell_j)$ that is involved in the leftmost upward crossing with $l(p, \ell_i)$—if there is any. Since all labels below ℓ_i have already been processed and are not involved in any upward crossings any more, ℓ_j must lie above ℓ_i. This implies that the crossing is located on the horizontal segment of $l(p, \ell_i)$, see Figure 3.7(b). We now swap the label assignment and replace $l(p, \ell_i)$ and $l(q, \ell_j)$ by $l(p, \ell_j)$ and $l(q, \ell_i)$, see Figure 3.7(c). Obviously this does not change the total leader length. Both new leaders $l(p, \ell_j)$ and $l(q, \ell_i)$ are upwards-oriented leaders and hence they cannot be involved in any new downward crossings—otherwise the new labeling, and therefore also the original labeling L^*, would be suboptimal. Regarding upward crossings, the horizontal segment of $l(q, \ell_i)$ is crossing-free since all leaders to labels below ℓ_i are crossing-free, and the horizontal segment of $l(q, \ell_i)$ is crossing-free by construction. Hence, $l(q, \ell_i)$, the leader to ℓ_i, is not involved in any upward crossing and we can proceed with ℓ_{i+1}.

Since each of the n labels is processed exactly once, using $O(n)$ time per label, the total running time is $O(n^2)$. ♣

We now describe our $O(n \log n)$-time algorithm to compute a crossing-free labeling with *po*-leaders of minimum total length. The algorithm first scans the input to divide it into parts that can be handled independently; then it uses a sweepline algorithm for each of these parts. For a higher generalization we now allow space between two labels. The more restricted version in which the labels connects directly to each other works absolutely analogously.

The initial scan works as follows. Consider the horizontal strips defined in the previous subsection. We traverse these strips in order from bottom to top, counting for each strip σ:

- pa_σ: number of points above σ (incl. any point on the top edge of σ);

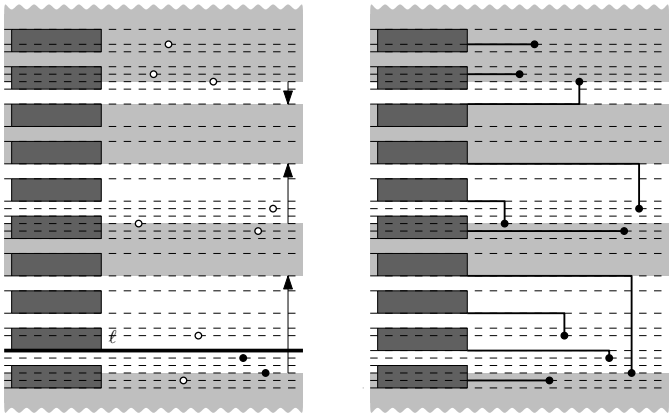


Figure 3.8: Left: Classification of strips in the plane-sweep algorithm: neutral strips are shaded, downward and upward strips are marked by arrows. When the sweep line reaches the label ℓ, the two black points are in W. Right: The completed minimum-length labeling.

- ℓa_σ: number of labels above σ (incl. any label intersecting σ);
- pb_σ: number of points below σ (incl. any point on the bottom edge of σ);
- ℓb_σ: number of labels below σ (incl. any label intersecting σ).

Note that for every strip, $pa_\sigma + pb_\sigma = n$, and $\ell a_\sigma + \ell b_\sigma$ is either n or $n+1$. We classify the strips in three categories and then divide the input into maximal sets of consecutive strips of the same category (see Figure 3.8):

- *downward*: strips s such that $pa_\sigma > \ell a_\sigma$ (and therefore $pb_\sigma < \ell b_\sigma$);
- *upward*: strips s such that $pb_\sigma > \ell b_\sigma$ (and therefore $pa_\sigma < \ell a_\sigma$);
- *neutral*: the remaining strips; these have $pa_\sigma = \ell a_\sigma$ and/or $pb_\sigma = \ell b_\sigma$.

Neutral sets are handled as follows: any point p that lies in the interior of a neutral set is labeled with a direct leader.

Points in an upward set S (including any points on its boundary) are labeled as follows. We use a plane-sweep algorithm, maintaining a waiting list W of points to be labeled, sorted by increasing x-coordinate. Initially W is empty. We sweep S with a horizontal line from bottom to top. During the sweep two types of events are encountered: *point events* (the line hits a point p) and *label events* (the line hits the bottom edge of a label ℓ). When a point event happens, we insert the point in W. When a label event happens, we remove the leftmost point from W and connect it to ℓ with the shortest possible leader. Using the leftmost point for labeling ℓ prevents producing crossings in the further run of our algorithm.

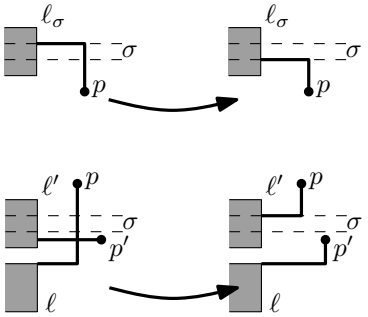
Points in downward sets are labeled by a symmetric plane sweep algorithm, going from top to bottom.

Theorem 3.3 *The valid one-sided length-minimum labeling using po-leaders can be computed in $O(n \log n)$ time requiring $O(n)$ space.*

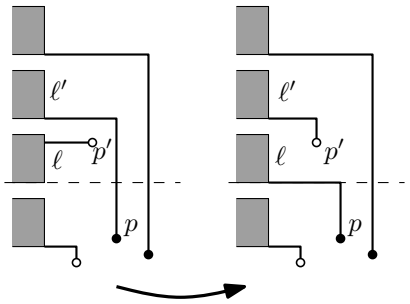
Proof. The algorithm described above can easily be implemented to run in $O(n \log n)$ time and $O(n)$ space. We will now prove that the algorithm produces a crossing-free labeling of minimum total leader length.

We observe that in any optimal labeling L, no leader crosses a neutral strip. To see this, consider any neutral strip σ. Assume $pa_\sigma = \ell a_\sigma$ (otherwise we have $pb_\sigma = \ell b_\sigma$, and that case is symmetric). Let ℓ_σ be the label that intersects σ, if it exists. Suppose L contains a leader l that crosses σ. We consider three cases:

- If l connects a point p to ℓ_σ, then l can be shortened by moving its horizontal segment on σ closest to p, eliminating the intersection with σ (Figure 3.9(a), top).
- If l leads from a point p above σ to a label ℓ below σ, then, because $pa_\sigma = \ell a_\sigma$, there must also be a leader l' leading from a point p' below σ to a label ℓ' above or intersecting σ. Now the total leader length can be reduced by connecting p to ℓ' and p' to ℓ (Figure 3.9(a), bottom).
- The case that l leads from a point p below σ to a label above σ is symmetric to the previous case.



(a) Eliminating intersections with neutral strips



(b) The black points are in W when the sweep line reaches ℓ.

Figure 3.9: Illustrations for the proof of Theorem 3.3

Swapping the labels of p and p' may cause leaders to intersect each other. Therefore the above argument only shows that if L contains a leader that crosses a neutral strip, then L is not length optimal among all, not necessarily crossing-free labelings. However, Lemma 3.1 shows that for any optimal labeling that has intersections, there exists a crossing-free labeling that is equally good. Thus the above argument also shows that if L contains a leader that crosses a neutral strip, then L is not optimal among all crossing-free labelings.

It follows that in any optimal labeling, points in the interior of neutral sets are labeled by direct leaders—as done by our algorithm. Observe that between any downward strip and any upward strip, both $pa_\sigma - \ell a_\sigma$ and $pb_\sigma - \ell b_\sigma$ differ by at least two. When going from a strip to an adjacent strip, the value of each of these expressions changes by at most one. Hence downward strips and upward strips are always separated by neutral strips. It follows that in any optimal labeling, the points in any upward (or downward) set S must be labeled by leaders that lie entirely within S. We will now argue that our plane-sweep algorithm for such a set S produces an optimal labeling for the points in S.

Consider an upward set S. Note that the strip β directly below S is a neutral strip with $pb_\beta \leqslant \ell b_\beta$ while the bottommost strip σ in S has $pb_\sigma > \ell b_\sigma$; hence we have $pb_\beta = \ell b_\beta$ and the first event must be a point event for a point p on the bottom edge of S. Observe that the strips β and σ may intersect a label ℓ (as in Figure 3.8), but it cannot be used to label p: since $pb_\beta = \ell b_\beta$ and no leaders can cross β, all labels up to and including ℓ are needed to label points below β. So we must label all points in (and on the boundary of) S with labels that lie entirely above σ. It remains to prove that our algorithm produces such a labeling.

First note that as soon as the number of label events processed catches up with the number of point events processed, we enter a neutral strip and the plane sweep of S ends; thus W always contains at least one point when a label event happens. Now consider a labeling L that deviates from the one produced by our algorithm. Let ℓ be the lowest label that, according to L, is *not* connected to the leftmost point p that is in W when the sweep line reaches ℓ. Note that it follows that p is connected to a label ℓ' above ℓ with a leader that crosses all strips that intersect ℓ; see Figure 3.9(b). Now ℓ cannot be connected to any other point in W, because that would create a crossing with the leader between p and ℓ'. So ℓ must be connected to a point p' above the sweep line—but then swapping the labels of p and p' and subsequently resolving any resulting crossings would give a labeling with smaller total leader length. Hence any labeling that deviates from the one produced by our algorithm must be suboptimal. ♣

Lemma 3.2 *The valid one-sided length-minimum labeling using po-leaders cannot be computed faster than in $O(n \log n)$ time.*

Proof. To see that solving the labeling problem takes $\Omega(n \log n)$ time in the worst case, consider a sequence $x_1, x_2, \ldots, x_n$ of distinct positive numbers, not necessarily in sorted order. Suppose we compute a crossing-free labeling with *po*-leaders for a set of points $(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)$, where $0 < y_i < 1$ for $1 \leqslant i \leqslant n$ and $y_i \neq y_j$ for $1 \leqslant i < j \leqslant n$, and a set of labels with lower right corners $(0, 1), (0, 2), ..., (0, n)$, and look up, for the labels $(0, 1), (0, 2), ..., (0, n)$ in order, the points attached to it. We would find the points sorted in order of increasing x-coordinate, otherwise leaders would intersect. Hence computing a crossing-free labeling is at least as difficult as sorting. ♣

do-leaders

We cannot use the same approach as for *po*-leaders since not every point can connect to every label by a *do*-leader. Roughly speaking we will use a generalization of the algorithm by Bekos et al. [BKSW07] for computing the valid length-minimum labeling using *po*-leaders that takes these restrictions into account.

We start by introducing necessary conditions for the existence of a *do*-labeling and show how to algorithmically make use of them. In the end, we will constructively see that the established conditions are already sufficient for the existence of a valid *do*-labeling.

Each label ℓ induces a funnel-shaped subregion R_ℓ in which all points that could be assigned to this label are located, refer to Figure 3.6 for an example. The arrangement of all these regions defines $O(n^2)$ cells, see Figure 3.10. All points in the same cell of this arrangement can connect to the same set of labels and these sets are distinct for any two cells.

A cell itself is the intersection of an ascending and a descending diagonal strip and after numbering these strips we can index each cell, e.g. the white cell in Figure 3.10 has index

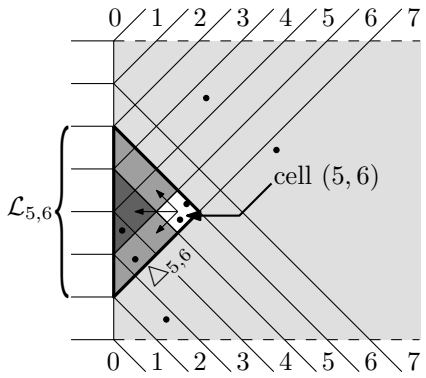


Figure 3.10: The cell arrangement.

$(5, 6)$, when we take the index of the descending strip as first coordinate. For a cell (i, j) we denote the label set that can be reached by $\mathcal{L}_{i,j}$ and the smallest triangle bordering to $\mathcal{L}_{i,j}$ and containing (i, j) by $\triangle_{i,j}$, see Figure 3.10. Now, a necessary condition for the existence of a *do*-labeling is obviously that the number $n_{i,j}$ of points in $\triangle_{i,j}$ shall not exceed the number of labels in $\mathcal{L}_{i,j}$ which is $i + j - n$. Otherwise there will be unassigned points left over in $\triangle_{i,j}$ that cannot connect to any other labels beside $\mathcal{L}_{i,j}$. We say that $i + j - n$ is the *level* of cell (i, j) and summarize the necessary conditions in the following lemma.

Lemma 3.3 *There can only be a valid do-labeling if for each k-level cell* (i, j) *it holds that* $n_{i,j} \leq k$.

We can check these necessary conditions in $O(n^2)$ time. For this we have to compute all numbers $n_{i,j}$: initially we set each $n_{i,j}$ to zero. For each input point we determine its containing cell (i, j) and increment $n_{i,j}$ by one. Then, each $n_{i,j}$ gives the number of points in the cell (i, j) but we aim for the number of points in $\triangle_{i,j}$. We traverse the cells in increasing order of their levels. Apparently, all 1-level cells already contain the desired values, for all other cells $n_{i,j}$ is updated based on three predecessor values (see Figure 3.10):

$$n_{i,j} \leftarrow n_{i,j} + n_{i,j-1} + n_{i-1,j} - n_{i-1,j-1}.$$

This counts each point in $\triangle_{i,j}$ exactly once. The time complexity per cell is obviously constant.

Now, we ready to present our algorithm that computes the length-minimum labeling. We assume that we computed the numbers $n_{i,j}$ in a preprocessing and neither of the necessary conditions has been violated. For a k-level cell (i, j) for which $n_{i,j}$ is k we say that $\triangle_{i,j}$ is *full*, meaning that in any valid *do*-labeling each of the $n_{i,j}$ points in $\triangle_{i,j}$ connects to a label in $\mathcal{L}_{i,j}$. For the algorithm we generalize the above definition: a triangle $\triangle_{i,j}$ is full if the numbers of points in $\triangle_{i,j}$ and labels in $\mathcal{L}_{i,j}$ that have not been assigned yet match. From now on we call such points and labels *open*.

The algorithm traverses the cells in increasing order of their levels and for each level from bottom to top. Whenever we find a k-level cell (i, j) for which $\triangle_{i,j}$ is full, we call the subroutine *complete*$(\triangle_{i,j})$ which computes a length-minimum valid labeling for the remaining open items in $\triangle_{i,j}$. Then, $\triangle_{i,j}$ is marked as completed. Eventually the traversal will examine the n-level cell (n, n) and if not all points have been assigned yet, an assignment for the remaining open points and labels will be found.

In the procedure *complete*($\triangle_{i,j}$) we process the open labels from bottom to top. Basically, for each open label ℓ the point that we assign to ℓ is the first open point that we find when we sweep $R_\ell \cap \triangle_{i,j}$ by a horizontal line from bottom to top (from now on we will omit the obvious intersection with $\triangle_{i,j}$. If the placement of the leader inserts any crossing with earlier drawn-in leaders we purge the crossings by flipping assigned labels without changing the total leader length similarly to Bekos et al. algorithm for the *po*-leaders.

However, we have to pay attention during the completion of a triangle $\triangle_{i,j}$: each time we assign a point that does not lie in a 1-level cell, we artificially shift this point into the 1-level cell adjacent to the assigned label, see Figure 3.13. This decreases the number of open labels for incompleted subtriangles of $\triangle_{i,j}$ while the number of open points in them stays the same, thus, these triangles can become full. If this happens we have to bring the completion of these recently filled subtriangles forward to the usual completion of $\triangle_{i,j}$.

For describing the full operation mode of *complete*($\triangle_{i,j}$) we distinguish the two cases that either a full triangle has newly been found (*complete*($\triangle_{i,j}$)) or a subtriangle (*subtriangle-complete*($\triangle_{i',j'}$)) has become full during the completion of a supertriangle.

complete($\triangle_{i,j}$)**:** First, we traverse the incompleted cells of $\triangle_{i,j}$ by a breadth-first search starting from (i, j). This yields lists of the remaining open points and labels in $\triangle_{i,j}$. If the lists of points and labels are empty we mark (i, j) as completed and are done, otherwise we sort both lists according to increasing y-coordinate.

Together with the BFS we purge redundant cells: due to already completed subtriangles of $\triangle_{i,j}$ cells can have become equivalent in the sense that they now can reach the same set of open labels. We merge these equivalent cells and assign the number and level of the original topmost-level cell to the newly emerged cell. Obviously, this maintains the number of points and labels in the triangle associated with the new cell, see Figure 3.11. This step is indispensable for the maintenance of a quadratic running time as the update of the cell entries that we have to do when we make an assignment later on would cause the runtime to get super-quadratic if the number of cells was quadratic.

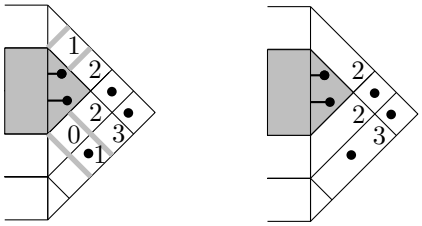


Figure 3.11: Merging redundant cells.

After finishing these initilizations we start with assigning points to the open labels. For this, we sweep the labels from bottom to top. For an open label ℓ we do the following: we traverse the list of open points and assign the first point p that we find and that is in R_ℓ to ℓ. We remove ℓ and p from the lists of open labels and points. If the leader from p to ℓ intersects earlier drawn-in leaders we take the leader of the topmost label among them and flip the assigned points with ℓ, we repeat this step until there are no crossings any more, see Figure 3.12.

Furthermore, after making an assignment, we update the cell structure and data of $\triangle_{i,j}$. For cells that have become redundant by the assigment this works analogously as for the initilization. For the numbers $n_{i,j}$ we have shifted the assigned point from its original cell

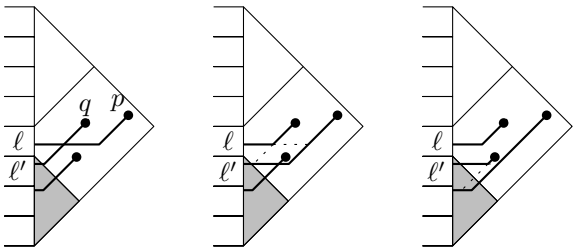


Figure 3.12: Purging crossings after assigning $p \to \ell$.

to the cell c_ℓ adjacent to ℓ. We trace the leader from ℓ back to p and update the affected cells accordingly, see Figure 3.13. This update can cause subtriangles (i', j') to become full. If this happens we have to bring their completion forward. For this, we prepare the lists of open labels and points in $\triangle_{i',j'}$ and start the subroutine *subtriangle-complete*$(\triangle_{i',j'})$. Then, we mark (i', j') including the according points and labels as completed and proceed with the completion of $\triangle_{i,j}$.

Finally, after the last open label in $\triangle_{i,j}$ is assigned we mark $\triangle_{i,j}$ as completed.

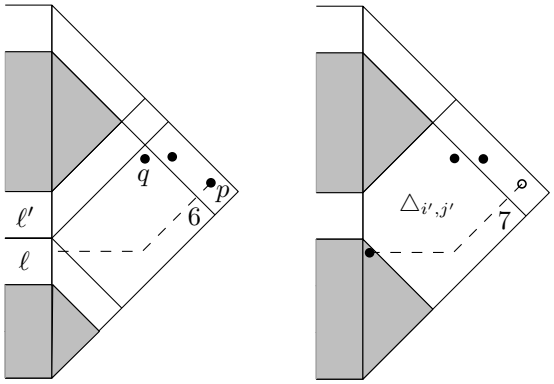


Figure 3.13: $\triangle_{i',j'}$ becomes full by assigning $p \to \ell$. The open point q in $\triangle_{i',j'}$ now has to be assigned to ℓ' in order to find a valid labeling.

subtriangle-complete$(\triangle_{i',j'})$**:** As before, only the lists of open points and labels are handed over by the overall procedure and are not computed by BFS.

Now we show that this algorithm indeed computes a valid length-minimum labeling in quadratic time.

Theorem 3.4 *The valid one-sided length-minimum labeling using do-leaders can be computed in $O(n^2)$ time requiring $O(n^2)$ space, if there is any. If not we can report infeasibility within the same time and space bounds.*

Proof. We assume that the necessary conditions from Lemma 3.3 hold, otherwise we report infeasibility after the $O(n^2)$-time preprocessing.

For the correctness of the described algorithm it is obviously sufficient to show that the procedure *complete*$(\triangle_{i,j})$, not called within the completion of a supertriangle, computes a valid length-minimum labeling within $\triangle_{i,j}$. We do this in the following stages: after *complete*$(\triangle_{i,j})$ has finished it holds that

(i) each of the labels $\mathcal{L}_{i,j}$ is assigned to a distinct point in $\triangle_{i,j}$.
(ii) the computed labeling is crossing free and length-minimum.
(iii) the computed labeling is valid, i.e. crossing free.

Finally for the time and space requirements we show that

(iv) the algorithm can be implemented in quadratic time and space.

For simplicity we prove (i)–(iii) only for cells (i, j) for which $\triangle_{i,j}$ does not contain any of the outermost 1-level cells, i.e. neither $(1, n)$ nor $(n, 1)$. Then, $\triangle_{i,j}$ is a right-angled triangle with its hypotenuse on the left. This makes the understanding easier. It is not hard to see that the following considerations generalize to the omitted cells. For sake of completeness note that *complete*$(\triangle_{i,j})$ has to sweep $\triangle_{i,j}$ from top to bottom if the cell $(1, n)$ is contained in $\triangle_{i,j}$.

(i):

We show that in the end every label that had been open when *complete*$(\triangle_{i,j})$ was called, gets a point assigned. Marking assigned points ensures that they are not assigned twice and thus the assignment is distinct.

Let $\ell_1, \ell_2, \ldots, \ell_k$ be the numbering of the labels $\mathcal{L}_{i,j}$ from bottom to top. We show that each time we take the next label $\ell = \ell_l$ from the open label list we still find an open point in R_ℓ. Let $\triangle_{\text{below}}$ be $\bigcup_{\kappa=1}^{l-1} R_\kappa \setminus R_\ell$ and $\triangle_{\text{above}}$ be $\bigcup_{\kappa=l+1}^{k} R_\kappa \setminus R_\ell$, see Figure 3.14. Inductively the labels $\ell_1, \ldots, \ell_{l-1}$ are already assigned. As the necessary conditions for the existence of a valid labeling hold we know that there are at most $k - l$ points in $\triangle_{\text{above}}$. This yields that $R_\ell \cup \triangle_{\text{below}}$ contains at least l points. Furthermore, by the proceeding of the algorithm it is clear that no point in R_ℓ is assigned to a label in $\ell_{l+1}, \ldots, \ell_k$ yet; some of the labels bordering to $\triangle_{\text{above}}$ can already be assigned but then their assigned points can only lie in $\triangle_{\text{above}}$ or a supertriangle of $\triangle_{\text{above}}$ that does not contain R_ℓ otherwise ℓ would also be assigned yet as it then lies in an earlier processed full triangle, a contradiction.

By the necessary conditions we also know that $\triangle_{\text{below}}$ contains at most $l - 1$ points all of which are assigned because we have only shifted points from R_ℓ to $\triangle_{\text{below}}$ as long as $\triangle_{\text{below}}$ was not full. Thus, the number of open points in R_ℓ is at least $k - (k - l) - (l - 1) = 1$, which means we still find an open point in R_ℓ for assigning it to ℓ_l.

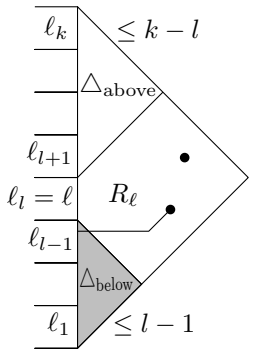


Figure 3.14: Illustrating part (i) of the proof of Theorem 3.4.

(ii):

We compare the labeling L^* that we would get by *complete*$(\triangle_{i,j})$ without performing any crossing purges with a length-minimum labeling $L^{\min}$ for $\triangle_{i,j}$ in which each point is labeled

by a distinct label but that is allowed to have crossings. Then, proving that L^* and $L^{\min}$ are equal in length is sufficient for (ii) because Lemma 3.1 shows that final labeling produced by *complete*($\triangle_{i,j}$) is crossing free and has no longer total leader length than L^*.

The key observation is that minimizing the total leader length comes down to minimizing the total length l^d of diagonal leader segments: for a point p let l_p^x be the x-distance to the labeling side of R. Then, a leader from p to any label has length $l_p^x + (1 - \frac{1}{\sqrt{2}})l_p^d$, where l_p^d is the length of the diagonal segment of p's leader. Thus, the total length of a fixed labeling is $\sum_p l_p^x + (1 - \frac{1}{\sqrt{2}})l_p^d = \sum_p l_p^x + (1 - \frac{1}{\sqrt{2}}) \sum_p l_p^d = \sum_p l_p^x + (1 - \frac{1}{\sqrt{2}})l^d$ which shows that minimizing l^d also minimizes the total length.

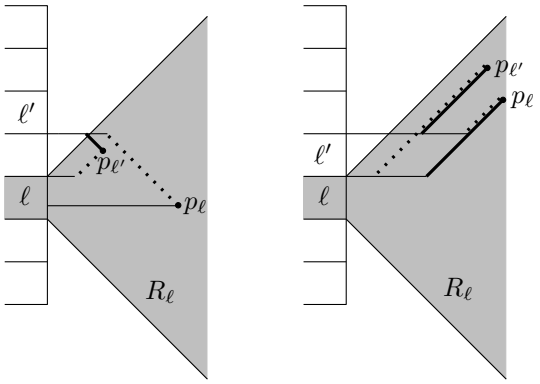


Figure 3.15: Illustrating part (iii) of the proof of Theorem 3.4.

We show that *complete*($\triangle_{i,j}$) computes an assignment which minimizes the total length of diagonal leader segments in $\triangle_{i,j}$. We will show this by a nested induction. The outer induction is on the level k of the cell (i, j) and the inner induction on k^{open}, the number of open points and labels when we find $\triangle_{i,j}$ to be full.

Outer induction base $k = 1$:
As the necessary conditions are satisfied, $\triangle_{i,j}$ contains exactly one point that is not assigned yet. Thus, k^{open} must be 1 and the open point is assigned to the label incident to $\triangle_{i,j}$ which is the case for every valid labeling.

Outer induction step $k > 1$:
Inner induction base $k^{\text{open}} = 0$: There are no remaining open items.
Inner induction step $k^{\text{open}} > 0$: Let ℓ be the bottommost open label adjacent to $\triangle_{i,j}$ and let p_ℓ be the point that the algorithm assigns to ℓ. We prove that there is an assignment for the remaining open items that minimizes the total length of diagonal leader segments that indeed labels p_ℓ with ℓ. The rest is then done by the induction hypotheses: Subtriangles that became full by the assignment are solved optimally by the outer induction hypothesis and the triangle $\triangle_{i,j}$ in which ℓ and p_ℓ are not open any more is solved optimally by the inner induction hypothesis.

It remains to prove that there is an minimum assignment that assigns p_ℓ to ℓ, assume the contrary. We show that there is a minimum assignment OPT' in which p_ℓ is assigned to a label ℓ' and the point assigned to label ℓ is p' such that we can switch the labels of p' and p resulting in an assignment OPT whose diagonal leader length is not longer than the one of OPT'. This contradicts the assumption as we then have a minimum assignment for which p is assigned to ℓ.

By outer and inner induction hypothesis, all subtriangles of $\triangle_{i,j}$ that are already completed have been labeled optimally. Since we took ℓ to be the bottommost open label in $\triangle_{i,j}$ this means that there is a minimum assignment OPT' in which $p_{\ell'}$ lies above p_ℓ and ℓ' lies above ℓ. By $p_{\ell'} \in R_\ell$ and by simple geometric observations we first get that $p_{\ell'}$ is in $R_{\ell'}$ and that we could indeed assign $p_{\ell'}$ to ℓ', see Figure 3.15. By a case distinction on the relation between p_ℓ and and the horizontal strip incident to ℓ (below/in-strip/above) and on the relation between p_ℓ and $p_{\ell'}$ (left/right) it is not hard to see that the assignment $p_{\ell'} \to \ell'$, $p_\ell \to \ell$ never exceeds the assignment $p_{\ell'} \to \ell$, $p_\ell \to \ell'$ in terms of diagonal leader length which yields the desired. We omit the examination of each case, two examples are depicted in Figure 3.15 where the significant diagonal segments are drawn bold and dotted for the assignment $p_{\ell'} \to \ell$, $p_\ell \to \ell'$ and solid for the assignment $p_{\ell'} \to \ell'$, $p_\ell \to \ell$.

(iii):

Storing the cells of the arrangement dominates the space consumption and is quadratic. A call to *complete*($\triangle_{i,j}$), where $\triangle_{i,j}$ has κ open points, requires at most $O(\kappa^2)$ time: after the lists of open points in $\triangle_{i,j}$ have been generated and sorted in $O(\kappa \log \kappa)$ time, finding the point for an open label and updating the list of remaining items is in $O(\kappa)$. For the crossing purges we have to deal with at most $O(\kappa^2)$ crossings in total. Since each point appears as an open point for exactly one full triangle this settles the total running time of $O(n^2)$. ♣

Remark. Constructively our algorithm shows that the necessary conditions in Lemma 3.3 are already sufficient for the existence of a valid *do*-labeling. Hence, for any subproblem that we deal with in the dynamic program of the bend-minimization algorithm for *do*-leaders in Section 3.3.1, we can check whether the subproblem is feasible in $O(m^2)$ time where m is the number of points in the subproblem. Performing these checks raises the theoretic running time of the algorithm from $O(n^5)$ to $O(n^6)$. However, in practice we detect infeasible subproblems as soon as possible, meaning that we do not compute needless table entries. It turned out that indeed, performing the checks yields a better running time in practice.

3.4 Algorithms for labels on two sides

In this section we consider the problem of matching points with labels that are placed on the left and on the right side of R. For simplicity we assume that the labels on the left and right side are aligned. Let again n denote the number of points and the number of labels. Note that by the above assumption n is always even and the number of labels on each side is $n/2$.

For the bend-minimization problems we use dynamic programming again. Disappointingly, the running times of our algorithms are rather high, i.e. $O(n^8)$ for the *po*-leaders and even $O(n^{14})$ for the *do*-leaders. Accordingly to the one-sided cases, the algorithms can be adapted for computing the length-minimum labelings as well, within the same running times. However, for length-minimization Bekos et al. [BKSW07] gave a nice quadratic-time algorithm for *po*-leaders that we briefly sketch in Section 3.4.2. For finding the length-minimum labeling using *do*-leaders it is still open whether there exists an algorithm with running time faster than $O(n^{14})$. However, this suggests itself as for all other variants there are faster algorithms for length-minimization than for bend-minimization.

3.4.1 Bend minimization

For both leader types the idea is the same: we partition the region of a subproblem into two regions by a polygonal line. Each such region represents a new subproblem that can be solved

independently. We will show that by the way in which we partition the regions, there is at least one bend-minimum labeling that will be found by our algorithm.

For the *po*-leaders a polygonal split line will require three parameters of linear range while for the *do*-leaders we need five. Thus, the table sizes are $O(n^6)$ and $O(n^{10})$, respectively, Since we have to examine $O(n^2)$ and $O(n^4)$ table entries for the determination of a new entry, we end up with a running time of $O(n^8)$ and $O(n^{14})$ for the *po*- and *do*-leaders, respectively.

We adapt the numbering of horizontal strips from Section 3.3.1, recall Figure 3.3b. Note that we now have $\frac{3}{2}n$ strips that partition R while the half planes below and above R are numbered with 0 and $\frac{3}{2}n + 1$.

Differing from the one-sided case we here impose the following restriction (*): each non-direct leader connects to the label in the middle of the strip that was assigned for its connection. Direct leaders still connect to their labels on the boundary of two strips as they induce this boundary, otherwise there will not be any direct leaders at all. Restriction (*) enables a simpler split which requires less technical details as without the restriction as we then can split the subproblems always with lines located in the middles of the strips. We note that our methods could be extended obeying the general case of sliding ports but this would involve nasty technical details; the theoretic running time would remain the same, however the number of horizontal strips has to doubled.

In general, (*) ensures that the connecting points will not lie too close to a label boundary, which justifies the restriction in practice. Surely, we increase the minimum length for the length minimum labeling by (*) if we use the algorithm for length minimization, but by the above argument the readability of the theoretic length minimal labeling is poor anyway.

However, we have to make sure that we do not lose any significant labelings for the optimization by (*). We show that the number of bends in a bend-minimum labeling $\mathcal{L}$ that does not obey (*) is not less than the number in a bend-minimum labeling obeying (*): let σ be the strip in which a non-direct leader l connects to its label in $\mathcal{L}$. W.l.o.g. we assume that l is upwards directed and connects to a label ℓ on the left side, see Figure 3.16. Let $\mathcal{L}^*_{\text{inv}}$ be the labeling that emerges from $\mathcal{L}$ when (*) gets fulfilled, i.e. we move all non-direct leaders to the midpoints of their corresponding connecting strips. $\mathcal{L}^*_{\text{inv}}$ has the same number of bends than $\mathcal{L}$ and is obviously not valid if and only if l intersects a leader l' that it did not intersect in $\mathcal{L}$. Since there are no points in the interior of σ, l' can only be a non-direct, downards directed leader that connects to the label on the right side in σ, see Figure 3.16. However then, redirecting l to the right and l' to the left yields a labeling $\mathcal{L}^*$ that has not more bends than $\mathcal{L}$ and whose length is even smaller than the length of $\mathcal{L}^*_{\text{inv}}$. Thus, we do not lose significant labelings for the bend minimization.

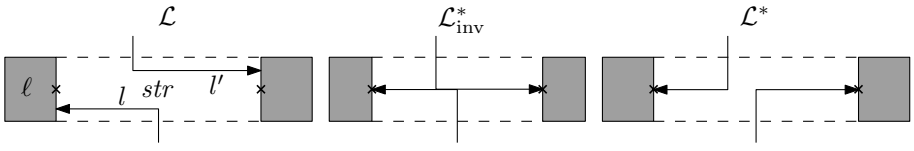


Figure 3.16: We do not lose a significant bend-minimum labeling by restriction (*).

po-leaders

As in Section 3.3.1 we assume that no two input points lie on a horizontal or a vertical line and no input point lies on a horizontal line induced by a label rectangle.

For the *po*-leaders we need a partitioning of the R into vertical strips: R is partitioned into $n+1$ vertical strips by the vertical lines through every input point. We number these strips by $0, 1, \ldots, n$ from left to right.

A subproblem of the dynamic program is enclosed by a lower and an upper boundary line. The lower boundary line consists of a left horizontal segment in strip $\mathrm{hor}_\ell^{\mathrm{bot}}$, a right horizontal segment in strip $\mathrm{hor}_r^{\mathrm{bot}}$, and a vertical segment in strip $\mathrm{ver}^{\mathrm{bot}}$ connecting the two horizontal segments. Note that the vertical segment is unnecessary if both horizontal segments use the same strip. In this case we can simply set the vertical parameter to 0. Similarly, the triple $(\mathrm{hor}_\ell^{\mathrm{top}}, \mathrm{ver}^{\mathrm{top}}, \mathrm{hor}_r^{\mathrm{top}})$ defines the upper boundary line. We will abbreviate the triple $(\mathrm{hor}_\ell^{\mathrm{bot}}, \mathrm{ver}^{\mathrm{bot}}, \mathrm{hor}_r^{\mathrm{bot}})$ by $\overrightarrow{bot}$ and the triple $(\mathrm{hor}_\ell^{\mathrm{top}}, \mathrm{ver}^{\mathrm{top}}, \mathrm{hor}_r^{\mathrm{top}})$ by $\overrightarrow{top}$. Additionally, we need a seventh parameter *type* to complete the description of a subproblem, this parameter indicates whether there are still open labels on both sides of R (*type* $= 2$) only on the left side of R (*type* $= 1_l$) or only the the right side (*type* $= 1_r$). A subproblem is thus specified by six parameters, with $O(n)$ choices for six of them and a constant range for the seventh.

In contrast to the one-sided case, when splitting, we here only partition the plane and do not assign the labels that are incident to the boundary line yet. This is why we additionally endow the horizontal parameters by a sign that indicates whether the label incident to the denoted strip belongs to the subproblem $(+)$ or not $(-)$. Hence, the range of the parameters $\mathrm{hor}_\ell^{\mathrm{bot}}, \mathrm{hor}_r^{\mathrm{bot}}, \mathrm{hor}_\ell^{\mathrm{top}}, \mathrm{hor}_r^{\mathrm{top}}$ is $\{0, \pm 1, \ldots, \pm\frac{3}{2}n, \frac{3}{2}n+1\}$. The range of $\mathrm{ver}^{\mathrm{bot}}$ and $\mathrm{ver}^{\mathrm{top}}$ is $\{0, 1, \ldots, n\}$. See Figure 3.17 for examples.

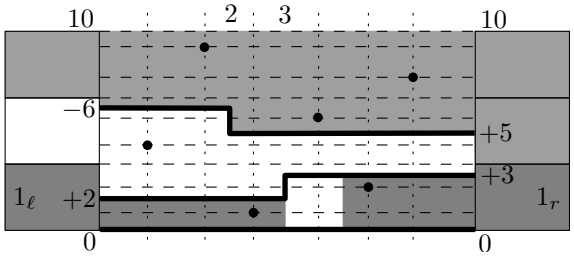


Figure 3.17: The subproblem of *type* $= 2$ with $\overrightarrow{bot} = (-6, 2, +5)$ and $\overrightarrow{top} = (10, 0, 10)$ is highlighted in light grey. The degenerated subproblems of *type* $= 1_\ell$ and *type* $= 1_r$ for $\overrightarrow{bot} = (0, 0, 0)$ and $\overrightarrow{top} = (+2, 3, +3)$ are highlighted in grey.

We call the subproblems of *type* 1_l and 1_r *degenerated*. We have to describe their definition in more detail since we want them to use the same parameters as the *type*-2 instances. Let $(1_\ell[1_r], \overrightarrow{bot}, \overrightarrow{top})$ be a degenerated subproblem and let n_ℓ $[n_r]$ be the number of belonging labels on the left [right] side. Then, the degenerated instance $(1_\ell[1_r], \overrightarrow{bot}, \overrightarrow{top})$ contains the n_ℓ $[n_r]$ leftmost [rightmost] points of the subproblem region, see Figure 3.17 for an example.

The table entry $T[\overrightarrow{bot}, \overrightarrow{top}, \mathit{type}]$ shall give the minimum number of bends for a valid labeling of the induced subproblem. The base cases are: for a subproblem that contains no points we set $T = 0$. For a subproblem that contains one point we check whether there is a *po*-leader that directly connects to the open label, whether a connection by a bend leader is possible or whether there is no *po*-leader at all that connects the point and the incident label. Accordingly we set $T = 0$, $T = 1$, or $T = \infty$, a situation in which the latter happens is depicted in Figure 3.18.

A subproblem S that still contains at least two points will be further split. For this we use a polygonal cut line defined by three parameters $(\mathrm{hor}_\ell^{\mathrm{cut}}, \mathrm{ver}^{\mathrm{cut}}, \mathrm{hor}_r^{\mathrm{cut}}) =: \overrightarrow{cut}$ splitting S

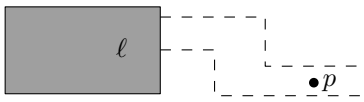


Figure 3.18: There is no *po*-leader from p to label ℓ.

into two subproblem. To make sure that these subproblems are well-defined we only iterate over polygonal lines $\overrightarrow{cut}$ that do not stick out of the region defined by S. For this we consider the regions as closed, meaning that $|\mathrm{hor}_\ell^{\mathrm{cut}}|$ can come from $\{\mathrm{hor}_\ell^{\mathrm{bot}}, \mathrm{hor}_\ell^{\mathrm{bot}}+1, \ldots, \mathrm{hor}_\ell^{\mathrm{top}}\}$ and $|\mathrm{hor}_r^{\mathrm{cut}}|$ can come from $\{\mathrm{hor}_r^{\mathrm{bot}}, \mathrm{hor}_r^{\mathrm{bot}}+1, \ldots, \mathrm{hor}_r^{\mathrm{top}}\}$. By looking at $\overrightarrow{bot}$ and $\overrightarrow{top}$ we can obviously determine the values of $\mathrm{ver}^{\mathrm{cut}}$ that satisfy the above condition in constant time. Moreover, using only one $\mathrm{ver}^{\mathrm{cut}}$ for splitting is enough, namely one of the $\mathrm{ver}^{\mathrm{cut}}$'s that splits into two subproblems in which the number of points and associated labels match. To summarize, we call a triple $\overrightarrow{cut}$ *feasible* if the induced polygonal line does not stick out of S and splits S into two subproblems in which the number of points and associated labels match.

We define $-\overrightarrow{cut}$ to be $(-\mathrm{hor}_\ell^{\mathrm{cut}}, \mathrm{ver}^{\mathrm{cut}}, -\mathrm{hor}_r^{\mathrm{cut}})$. This makes sure that the label ℓ incident to the split line is assigned to only one subproblem. However, we have to be a bit more careful here, if ℓ does not belong to the problem that we are about to split, the horizontal strip parameters must always appear as their negative value as we would otherwise re-assign ℓ that had already been assigned to a different subproblem.

Now, we can give the recursion for the computation of the table entries.

$$T[\overrightarrow{bot}, \overrightarrow{top}, 2] = \min_{\overrightarrow{cut}\text{ feasible}} \left\{T[\overrightarrow{bot}, \overrightarrow{cut}, 2] + T[-\overrightarrow{cut}, \overrightarrow{top}, 2], T[\overrightarrow{bot}, \overrightarrow{top}, 1_\ell] + T[\overrightarrow{bot}, \overrightarrow{top}, 1_r]\right\},$$

$$T[\overrightarrow{bot}, \overrightarrow{top}, 1_\ell] = \min_{\overrightarrow{cut}\text{ feasible}} \left\{T[\overrightarrow{bot}, \overrightarrow{cut}, 1_\ell] + T[\overrightarrow{cut}, \overrightarrow{top}, 1_\ell]\right\} + \delta_{p_\ell, h_\ell},$$

$$T[\overrightarrow{bot}, \overrightarrow{top}, 1_r] = \min_{\overrightarrow{cut}\text{ feasible}} \left\{T[\overrightarrow{bot}, \overrightarrow{cut}, 1_r] + T[\overrightarrow{cut}, \overrightarrow{top}, 1_r]\right\} + \delta_{p_r, h_r}.$$

Note that if there is no feasible split for a subproblem, we define the minimum over an empty set as ∞. In the degenerated cases p_ℓ (p_r) denotes the rightmost (leftmost) point of the instance. In the first case a feasible cut is of the type $(-h_\ell, \mathrm{ver}^{\mathrm{cut}}, \mathrm{hor}_r^{\mathrm{cut}})$, where h_ℓ is the strip that the leader of p_ℓ connects to. The value δ_{p_ℓ, h_ℓ} is defined as for the one-sided case and equals 0 if the leader is direct and 1 otherwise. For a degenerated subproblem of type 1_r the terms are defined analogously.

For a better understanding we have ignored a technical detail by now: there are feasible splits that split a type-2 instance into a degenerated and a non-degenerated subproblem. We can overcome this easily: everytime a type-2 instance is generated we check whether its type is really 2. If not we change the type accordingly. This can be done in constant time by looking at $\overrightarrow{bot}$ and $\overrightarrow{top}$ and its signs.

The entry $T[0, 0, 0, \frac{3}{2}n+1, 0, \frac{3}{2}n+1, type = 2]$ characterizes the whole input instance and thus gives—as Theorem 3.5 will show—the minimum number of bends for a valid labeling. The table size is $O(n^6)$ and each entry can be determined by exploiting $O(n^2)$ other entries which establishs the theoretical worst-case running time of $O(n^8)$.

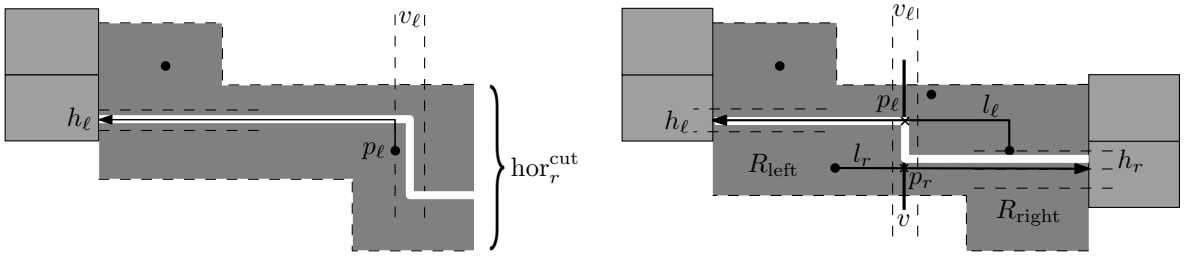
Theorem 3.5 *A valid bend-minimum two-sided labeling using po-leaders can be computed in $O(n^8)$ time requiring $O(n^6)$ space.*

Proof.

Let S be a subproblem defined by $(\overrightarrow{bot}, \overrightarrow{top}, type)$. We show that for every assignment of $(\overrightarrow{bot}, \overrightarrow{top}, type)$ the table entry $T[\overrightarrow{bot}, \overrightarrow{top}, type]$ holds the minimum number of bends for a valid labeling of S or ∞ if there is no valid labeling. Note that the input instance always has a valid labeling.

We start with the case that S is degenerated, w.l.o.g. of type 1_ℓ. The analysis is very similar to the one-sided case, however, we have to take more care as the boundary lines are not necessarily straight lines.

Let n_ℓ be the number of associated labels on the left side, let P_ℓ be the n_ℓ leftmost points in S and let p_ℓ be the rightmost point in P_ℓ, see Figure 3.19a. Finally, let v_ℓ be the vertical strip that has p_ℓ on its left boundary. If no valid labeling for S exists we (possibly later) end up with a subproblem for which there is no valid labeling as we only iterate over feasible splits. This means that eventually T will be evaluated as ∞. So we only consider the case that there is a valid labeling, we fix a bend-minimum one and denote the strip in which the leader of p_ℓ connects to the left side in the bend-minimum labeling by h_ℓ (if the leader is a direct leader and thus incident to two strips, we can take any). We set $\mathrm{hor}_\ell^{\mathrm{cut}} = -h_\ell$, $\mathrm{ver}^{\mathrm{cut}} = v_\ell$ and $\mathrm{hor}_r^{\mathrm{cut}}$ appropriately in $\{\mathrm{hor}_r^{\mathrm{bot}}, \mathrm{hor}_r^{\mathrm{bot}}+1, \ldots, \mathrm{hor}_r^{\mathrm{top}}\}$ such that the induced polygonal line $\overrightarrow{cut}$ does not stick out of the region of S. By the structure of the two boundary lines it is not hard to see that such a value for $\mathrm{hor}_r^{\mathrm{cut}}$ always exists. As in the one-sided case, there is only a linear number of feasible splits for such a degenerated instance.



(a) Type 1_ℓ. **(b)** Type 2.

Figure 3.19: Illustrations for the proof of Theorem 3.5. The polygonal lines that split the instance such that a fixed bend-minimum labeling is found are depicted in white.

By construction $\overrightarrow{cut}$ is not pierced by any leader of the fixed bend-minimum labeling. By setting $\mathrm{hor}_\ell^{\mathrm{cut}} = -h_\ell$ and counting δ_{p_ℓ,h_ℓ} we reserve this strip for the optimal leader of p_ℓ and define two subproblems that both do not contain p_ℓ and its label any more. These subproblems are then solved optimally by induction.

It remains to show that $T[\overrightarrow{bot}, \overrightarrow{top}, type]$ also holds the correct value for S being of type 2. Let n_ℓ and n_r be the number of associated labels on the left and on the right side, respectively. There is at least one vertical strip v_ℓ such that the mid-vertical line v in v_ℓ partitions the region of S into two regions R_{left} and R_{right} such that there are n_ℓ points in R_{left} and n_r points in R_{right}, see Figure 3.19b. Again, we fix a bend-minimum labeling. We show that there is always a feasible split using v_ℓ such that its split line is not pierced by any leader of the fixed labeling. Then we are done since the emerging subproblems are solved optimally by induction.

We look at the intersections of v and leaders in the fixed bend-minimum labeling. If no

leader intersects v, the dynamic program finds the value of the bend-minimum labeling when it looks at $T[\overrightarrow{bot}, \overrightarrow{top}, 1_\ell]+T[\overrightarrow{bot}, \overrightarrow{top}, 1_r]$ and the split into two degenerated subproblems. Next, consider the case that there are leaders intersecting v. By the choice of v there must be a point in R_{right} that receives its label on the left side for every point in R_{left} that receives its label on the right side. Obviously, we can find a pair of leaders (l_ℓ, l_r), l_ℓ emanating in R_{right} and ending on the left side, l_r emanating in R_{left} and ending on the right side, such that no other leader intersects v between the intersection points p_ℓ and p_r of l_ℓ and l_r with v, see Figure 3.19b. It follows that $p_\ell \neq p_r$ otherwise l_ℓ and l_r would intersect, w.l.o.g. we assume that p_ℓ is above p_r. First, consider the case that l_ℓ and l_r both have bends in the fixed bend-minimum labeling. Let h_ℓ be the strip in which l_ℓ connects to its label and accordingly let h_r be the strip in which l_r connects to its label. Then, setting $|\text{hor}_\ell^{\text{cut}}| = h_\ell$, $|\text{hor}_r^{\text{cut}}| = h_r$ and $\text{ver}^{\text{cut}} = v_\ell$ indicates a split line that is not pierced by any leader of the fixed labeling: its complete left (right) horizontal segment coincides with a part of l_ℓ (l_r) and the vertical segment is not pierced by any leader by construction. The case in which any of l_ℓ and l_r are direct leaders in the fixed labeling is slightly more involved: we have to take care since the split line, which is always in the middle of a strip, does not coincide with the leader any more. However, for p_ℓ being above p_r it is not hard to see that choosing $|\text{hor}_\ell^{\text{cut}}|$ to be the lower incident strip to l_ℓ in case of l_ℓ being a direct leader and choosing $|\text{hor}_r^{\text{cut}}|$ to be the upper incident strip to l_r, together with $\text{ver}^{\text{cut}} = v_\ell$ again indicates a split line that is not pierced by any leader of the fixed labeling.

Hence, when iterating over all feasible splits for which the absolute values of $\text{hor}_\ell^{\text{cut}}$ and $\text{hor}_r^{\text{cut}}$ match the above choices, we also look at a split whose subproblems can be solved as in the optimal labeling (the combination of assigned labels has to match).

Because the vertical parameter v_ℓ is fixed, for a split into two subproblems we examine at most $O(n^2)$ table entries by iterating all possible horizontal segments of the split. This gives a running time of $O(n^8)$. ♣

do-leaders

The basic idea for computing the bend-minimum labeling using *do*-leaders is absolutely analogous as for the *po*-leaders: the plane is partitioned into regions such that the boundaries are not pierced by any leader of a fixed bend-minimum labeling. Differing to the case of *po*-leaders, the lines that split into subproblems here involve diagonal segments instead of vertical segments.

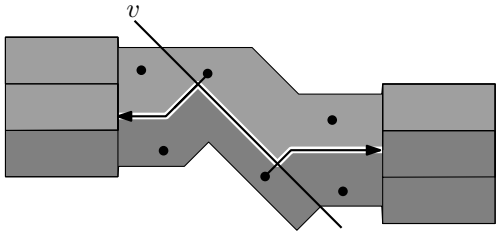


Figure 3.20: The white line indicates the split into two subproblems.

The vertical line v that splits the points evenly into a left and right region for which the number of points and associated label matches must here also be choosen as a diagonal line. Since v can be crossed by two *do*-leaders in the fixed bend-minimum labeling which have

different diagonal direction than v, we need five parameters of linear range to define one region boundary, see Figure 3.20. This immediately suggests a table size of $O(n^{10})$. Similarly to the *po*-leaders we can spare one of these five parameters for the split line into two subproblems. This is why we end up with a running time of $O(n^{14})$. As the analysis is very similar to the case of *po*-leaders and does not bear any new ideas we omit the details and state only the result.

Theorem 3.6 *A valid bend-minimum two-sided labeling using do-leaders can be computed in $O(n^{14})$ time requiring $O(n^{10})$ space, if there is any. If not we can report infeasibility within the same time and space bounds.*

3.4.2 Length minimization

po-leaders

For finding the length-minimum labeling with labels on two sides Bekos et al. [BKSW07] give a nice quadratic-time algorithm that does not seem to be improvable. We briefly outline the functionality of their algorithm. They sweep the points from bottom to top and fill a quadratic table T, where $T[i, j]$ contains the length of the length-minimum labeling of the $i + j$ lowest points for which i points have received labels on the left side and j points on the right side. The entry $T[i, j]$ can be determined by checking $T[i - 1, j]$ and $T[i, j - 1]$ and deciding whether the $(i + j)$th point receives its label on the left or on the right side. In the end entry $T[n/2, n/2]$ contains the length of the length-minimum labeling of the input instance and the labeling itself can be found by backtracking. Again, a postprocessing step to purge crossings is required. Crossings of leaders that go to the same side can be purged as for the one-sided case. Crossings of leaders that go to different sides cannot occur because they would contradict the length minimality.

Theorem 3.7 (Bekos et al. [BKSW07]) *For labels on two sides, a valid length-minimum labeling using po-leaders can be computed in $O(n^2)$ time requiring $O(n^2)$ space.*

Remark. Bekos et al. technique is not applicable for minimizing bends since a bend leader in the minimum bend labeling can be very long which is not covered by the dynamic-programming "sweep-from-bottom-to-top"-approach. One could try to overcome this by computing a minimum matching in the complete graph in which direct leaders are weighted by 0 and bend leaders are weighted by 1. However, this approach fails too, since the labeling induced by the minimum matching can contain crossings and purging these crossings would not leave the number of bends unchanged.

do-leaders

For finding the length-minimum labeling using *do*-leaders we have not found an algorithm that outperforms the adapted $O(n^{14})$-time algorithm for minimizing the number of bends. Apparently, we cannot adapt previous strategies: the feasibility grid for the one-sided case is completely out of place because we do not even know to which side a point should be assigned; if the point-containing rectangle R is wide enough, a point that can reach only a limited number of labels on one side can reach all labels on the other side which makes an argumentation impossible without fixing a side assignment for all points first. Bekos et al. dynamic-programming "sweep-from-bottom-to-top"-approach cannot be applied either since

not every point can be labeled by every label. Their algorithm strongly relies on this property that trivially holds for the *po*-leaders.

Theorem 3.8 *The valid length-minimum two-sided labeling using do-leaders can be computed in $O(n^{14})$ time requiring $O(n^{10})$ space, if there is any. If not we can report infeasibility within the same time and space bounds.*

3.5 Experiments

We implemented our algorithms[1] to evaluate their practical running time and to judge the quality of the labelings they produce.Let us start with a simple question. Which of the labelings in Figure 3.21 is preferable, i.e. which optimization criterion produces the better labeling, the length or the bend minimization? We get back to our answer for this question at the end of this section and first start with a general discussion on the results.

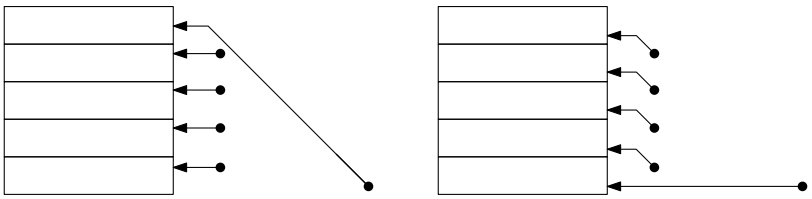


Figure 3.21: An instance where the bend-minimum labeling has 1 bend and the length-minimum labeling has $n - 1$ bends.

One-sided labelings. We tested the algorithms for the one-sided labelings on a map showing the departments of France, see Figure 3.22. The labelings were computed on an AMD Sempron 2200+ with 1GB main memory, which took 1 ms for the *po*-leaders and 15 ms for the *do*-leaders. Running the dynamic programs in a top-down fashion, for *po*-leaders 39% of $O(n^2)$ table entries were computed, while for the *do*-leaders only 0.21% of $O(n^4)$ entries were computed. We also ran the algorithms on artifically generated instances of 100 points. Here, the computation of the *po*-leaders took 234 ms averaged over 30 instances and on average 22% of the table entries were computed. The average running time for the *do*-leaders on the same instances was 3328 ms and on average 0.01% of the table entries were computed.

Two-sided labelings. The results for the two-sided labelings were very disappointing. For the simpler case of the *po*-leaders even computing instances with only 10 points already took several minutes. Hence, one can hardly say that our two-sided algorithms are of any practical relevance. After the results for the *po*-leaders had already been so bad, we refrained from implementing the two-sided *do*-leader version.

To get a two-sided labeling in practice we suggest to either use the *po*-leader length-minimization algorithm of Bekos et al. [BKSW07] (see Section 3.4.2) or to split the instance along the vertical line through the point with median x-coordinate and to solve the resulting one-sided problems. We leave it as an open problem to find efficient heuristics for dividing points between the left and the right side in an appropiate fashion to find good two-sided *po*-

[1] A Java applet is available at `http://i11www.iti.uni-karlsruhe.de/labeling`.

and *do*-labelings. Note that splitting in the middle does in general not yield aesthetically good results. For *do*-leaders a feasible instance can even get infeasible by splitting in the middle.

***po*-leaders vs. *do*-leaders.** Comparing *po*- to *do*-leaders, we find *do*-leaders preferable because their shape is easier to follow, which simplifies finding the correct label for a point and vice versa.

Distance between leaders. A bothering aspect of our figures is the closeness of some leader segments: in Figure 3.22a the vertical segments of the leaders for Franche-Comte and Lorraine are so close that seeing the correct assignment becomes impossible. A way to handle this problem is to only accept a leader to a certain point if there is no other point within a predefined safety margin around the leader. If we do not find a feasible solution that obeys this restriction we reduce the safety margin and start from scratch.

Leader connecting points. Another bothering aspect is that a direct leader may get very close to a label boundary and thus complicates the point-to-label understanding, see e.g. the leader that connects to Alsace in Figure 3.22c. A way to handle this problem is either to introduce safety spaces between the labels or to incorporate penalities for closeness to label boundaries in the dynamic programs.

Optimizing for length vs. bends. Optimizing the minimum total leader length seems to give more comprehensible and visually more pleasing results than optimizing the minimum total number of bends. The reason for this seems to be that optimizing for minimum length favours having each label close to the point being labeled. This results in a label assignment where the vertical order of the labels tends to reflect the location of the points in the figure fairly well. In contrast, when minimizing the number of bends this correspondence is more easily lost, which can be confusing, see e.g. the label position of Alsace in Figures 3.22a and 3.22b. In addition, the longer the non-horizontal leader segments are, the harder leaders are to follow—see Alsace in Figure 3.22b and Franche-Comte in Figures 3.22b and 3.22d.

Nevertheless, locally, bend minimization may produce nicer labelings than length minimization, see e.g. the regions Poitou-Charantes, Auvergne and Limousin. In the minimum-bend labeling these regions are labeled exclusively by direct leaders, while with minimum-length labeling these regions are labeled by non-direct leaders. For this reason we conjecture that a hybrid method may produce the best labelings. The idea is that a penalty has to be paid for every non-direct leader and that this penalty depends on the length of the non-horizontal segment of the leader.

A hybrid method The hybrid method combines the objectives of length and bend minimization in the following way: the value $bad_{\text{hyb}}(l)$ that a leader l contributes to the objective function is

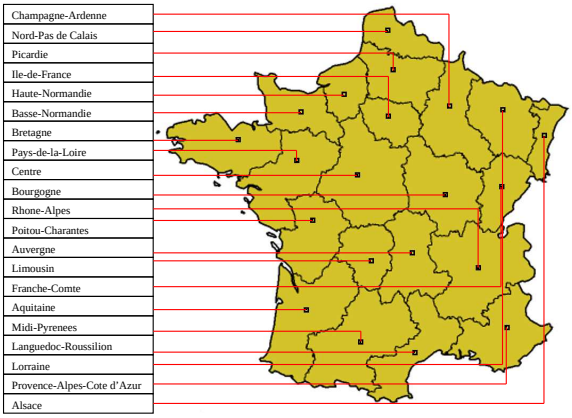
$$bad_{\text{hyb}}(l) = \frac{|\text{non-horizontal segment}(l)|}{|\text{horizontal segment}(l)|} + \lambda_{\text{bend}}\delta_{\text{bend}}(l), \tag{3.1}$$

where $\delta_{\text{bend}}(l)$ is 1 if l has a bend and 0 if l is a direct leader, $|\cdot|$ denotes the Euclidean length. The motivation for taking the length ratio between non-horizontal and horizontal segment of the leader into account is that a long non-horizontal segment on a short horizontal segment

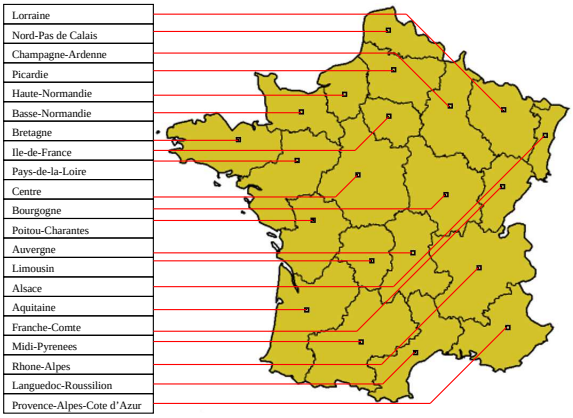
looks worse than on a long horizontal segment. The parameter λ_{bend} is used to adjust the weight of the penalty for the bend. Increasing λ_{bend} favors to have a few number of bends in the resulting labeling.

When testing the hybrid method in practice, our hope that this hybrid method would combine the advantages of the other two methods was fairly well fulfilled, see Figures 3.22e and 3.22f. The comprehension of the point-label assignment is locally optimized. Additionally, the longer a leader is, the more acceptable it becomes that the location of its label does not exactly reflect the position of the point. This seems to be acceptable for a good visualization of the labeling.

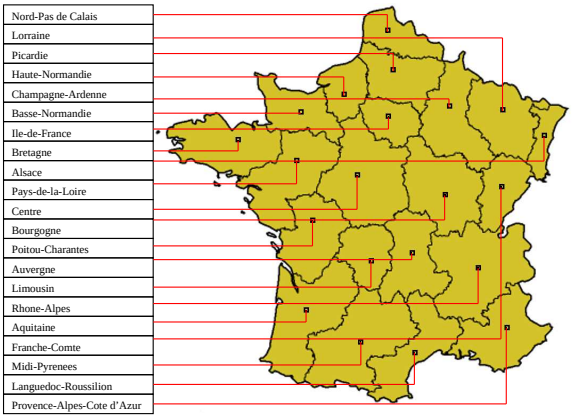
Conclusion. We find that it is better to minimize the total length rather than the number of bends, and that *do*-labelings are superior to *po*-labelings. However, to obtain an aesthetically excellent labeling it is not enough to compute the minimum-length *do*-labeling. Local aspects and the distance between leaders have to be taken into account. For this we suggest to choose a hybrid method that combines both approaches for the production of nice labelings in practice.



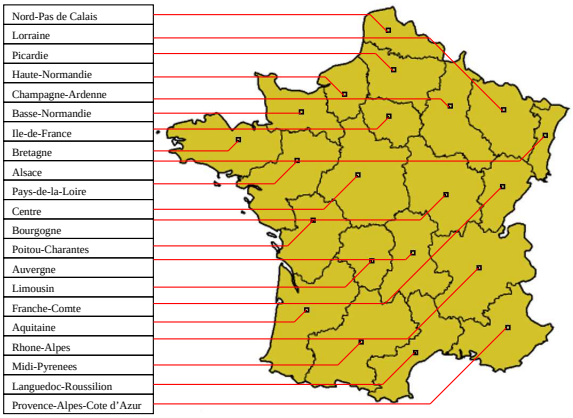
(a) Min. number of bends for *po*-leaders.



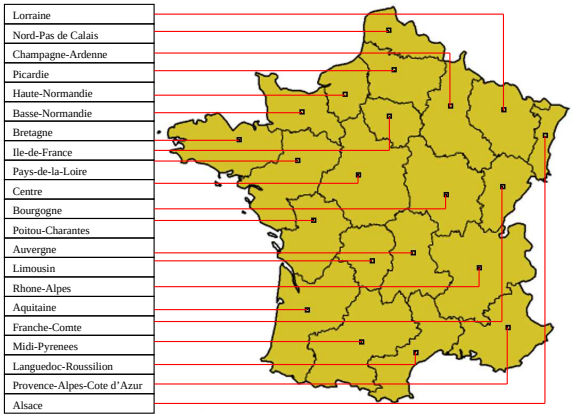
(b) Min. number of bends for *do*-leaders.



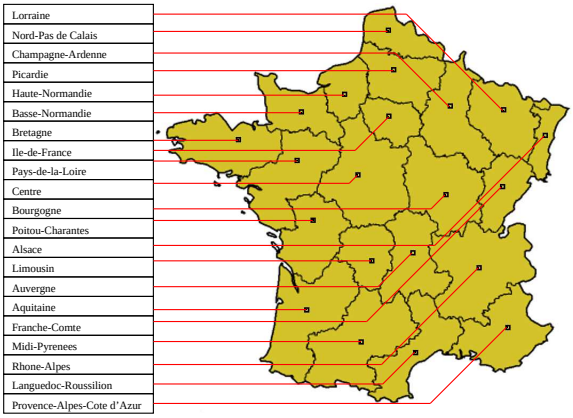
(c) Minimum length for *po*-leaders.



(d) Minimum length for *do*-leaders.



(e) Hybrid method for *po*-leaders.



(f) Hybrid method for *do*-leaders.

Figure 3.22: One-sided labelings for the main-land departments of France.

Part II

Analyzing Geometric Networks

Chapter 4

Detecting and Reporting Flocks

Flock recognition constitutes the first chapter of the *analyzing* part. We want to analyze the given trajectories of entities that move in the plane for the social behavior of the participating individuals. Are all moves independent from each other or are there flocks of individuals gathering and moving along together for a certain amount of time?

The chapter is based on conference publication [6]: Marc Benkert, Joachim Gudmundsson, Florian Hübner and Thomas Wolle: "Reporting Flock Patterns". A full version has been submitted to the *International Journal of Computational Geometry and Application*.

4.1 Introduction

Data related to the movement of objects is becoming increasingly available because of substantial technological advances in position-aware devices such as GPS receivers, navigation systems and mobile phones. The increasing number of such devices will lead to huge spatio-temporal data volumes documenting the movement of animals, vehicles or people. One of the objectives of spatio-temporal data mining [MH01, RHS01] is to analyse such data sets for interesting patterns. For example, a herd of 25 moose in Sweden was equipped with GPS-GSM collars. The GPS collar acquires a position every half hour and then sends the information to a GSM-modem where the positions are extracted and stored. Analysing this data gives insight into entity behaviour, in particular, migration patterns. There are many other examples where spatio-temporal data is collected [wtp06, Por]. The analysis of moving objects also has applications in sports (e.g. soccer players [IS02]), in socio-economic geography [FRC01] and in defence and surveillance areas.

We model a set of moving objects by a set P of n moving point objects $p_1, \ldots, p_n$ whose locations are known at τ consecutive time-steps $t_1, \ldots, t_\tau$ that is, the trajectory of each object is a polygonal line that can self-intersect, see Fig. 4.1a. For brevity, we will call moving point objects *entities* from now on. We assume that the positions are sampled synchonously for all entities, and that an entity moves between two consecutive positions along a straight line with constant speed.

There is some research on data mining of moving objects (e.g. [KSB01, SC03, SB00, VC06]) in particular, on the discovery of similar directions or clusters. Verhein and Chawla [VC06] used associated data mining to detect patterns in spatio-temporal sets.

In 2002 Laube and Imfeld [LI02] proposed a different approach: the REMO (RElative

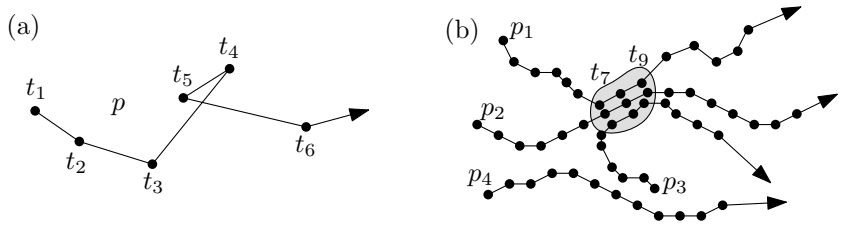


Figure 4.1: (a) A polygonal line describing the movement of an entity p in the time interval $[t_1, t_6]$. (b) p_1, p_2, p_3 form a flock in the time interval $[t_7, t_9]$.

MOtion) framework, which defines similar behaviour in groups of entities. They define a collection of spatio-temporal patterns based on similar direction of motion or change of direction. Laube et al. [LvKI04] extended the framework by not only including direction of motion, but also location itself. They defined several spatio-temporal patterns, including *flock*, *leadership*, *convergence* and *encounter*, and gave algorithms to compute such patterns efficiently.

Laube et al. [LvKI04] developed an algorithm for finding the largest flock pattern (maximum number of entities) using the higher-order Voronoi diagram with running time $\mathcal{O}(\tau(nm^2 + n\log n))$, where m is the minimum number of entities that a flock has to contain. They also showed that the detection problem can be answered in $\mathcal{O}(\tau(nm + n\log n))$ time. Applying the paper by Aronov and Har-Peled [AHP05] to the problem gives a $(1+\varepsilon)$-approximation with expected running time $\mathcal{O}(\tau n/\varepsilon^2 \log^2 n)$. Gudmundsson et al. [GvKS04] showed that if the disk (i.e. the region in which the entities have to be in order to form a flock) is $(1+\varepsilon)$-approximated then the detection problem can be solved in $\mathcal{O}(\tau(n/\varepsilon^2 \log 1/\varepsilon + n\log n))$ time.

However, the above algorithms only consider each time-step separately, that is, given $m \in \mathcal{N}$ and $r > 0$ a flock is defined by at least m entities within a circular region of radius r and moving in the same direction at some point in time. We argue that this is not enough for most practical applications, e.g. a group of animals may need to stay together for days or even weeks before it defines a flock. Therefore we propose the following definition of a flock:

Definition 4.1 *continuos (m, k, r)-flock – Let there be given a set of trajectories, where each trajectory consists of τ line segments. A flock in a time interval $I = [t_i, t_j]$, where $j - i + 1 \geqslant k$, consists of at least m entities such that for every point in time within I there is a disk of radius r that contains all the m entities.*

Given this model, Gudmundsson and van Kreveld [GvK06] recently showed that computing the longest duration flock and the largest subset flock is NP-hard to approximate within a factor of $\tau^{1-\varepsilon}$ and $n^{1-\varepsilon}$, respectively. They also give a 2-radius approximation algorithm for the longest duration flock with running time $\mathcal{O}(n^2\tau \log n)$.

We describe efficient approximation algorithms for reporting and detecting flocks, where we let the size of the region deviate slightly from what is specified. Approximating the size of the circular region with a factor of $\Delta > 1$ means that a disk with radius between r and Δr that contains at least m objects may or may not be reported as a flock while a region with a radius of at most r that contains at least m entities will always be reported.
We present several approximation algorithms for this model, for example, a $(2+\varepsilon)$-approximation with running time $T(n) = \mathcal{O}(\tau n k^2(\log n + 1/\varepsilon^{2k-1}))$ and a $(1+\varepsilon)$-approximation algorithm with running time $\mathcal{O}(1/m\varepsilon^{2k}) \cdot T(n)$.

Our aim is to present algorithms that are efficient not only with respect to the size of the

input (which is τn), but we also keep the dependency on k and m as small as possible. For most of the practical applications we have seen that m was between a couple of entities to a few hundreds or even thousands, and k was between 5 and 30.

In this model a set of entities can form many flocks and even one single entity can be involved in several flocks. For example, a flock involving $m + 1$ entities trivially contains $m + 1$ flocks of cardinality m. We must specify what we want to find and report in a given data set, see [GvKS04] for a discussion. A first possibility is simply to detect whether a flock occurs. If so, we may want to report one example of a flock. Secondly, we may want to find all flocks that occur. Thirdly, we may want to report the largest-size subset of entities that form a flock. In this chapter we mainly deal with the variant of finding all flocks of a given size m, in Section 4.4 we briefly discuss the other variants.

Next we give a brief description of the skip-quadtree structure used in this chapter together with a description of the computational model used. In Section 4.2 we give a discrete version of the definition of a flock and prove that it is equivalent to the original definition provided that the entities move with constant velocity between consecutive time-steps. Furthermore, we describe our general approach to detect flocks. Then, in Section 4.3, we give three approximation algorithms which are all based on the general approach. In Section 4.4 we discuss different ways of pruning the set of flocks reported, and in the final section we discuss our implementations and experimental results.

4.1.1 Computational model

One of the main tools used in this chapter is the skip-quadtree presented by Eppstein, Goodrich and Sun [EGS05] in 2005. As it is standard [BET99, EGS05] in computational geometry, we assume that certain operations on quadtrees or octrees can be done in constant time. The computations needed to perform point location, range queries or nearest neighbour queries in a quadtree, involve finding the most significant binary digit at which two coordinates of two points differ. We assume that this can be done using a constant number of machine instructions by a most-significant-bit instruction, or by using floating-point or extended-precision normalisation.

4.1.2 Skip-quadtree

We will show that approximation algorithms can be obtained by performing a set of range-counting queries in higher-dimensional space. There are several data structures supporting this type of query; quadtrees, skip-quadtrees, octrees, kd-trees, range trees, BBD-trees, BAR-trees and so on. For our requirements we can either use skip-quadtrees or BBD-trees, and since the implementation of the randomised skip-quadtree is very simple we chose to use the skip-quadtree.

The skip-quadtree uses a compressed quadtree as the bottom-level structure. The standard compressed quadtree for d dimensions uses $\mathcal{O}(2^d \cdot n)$ space and the worst-case height is $\mathcal{O}(n)$. We briefly describe the structure and show how to modify the structure so that it uses $\mathcal{O}(dn)$ space while the query time will increase by a factor of $\mathcal{O}(d)$.

First, here is the original description of a compressed quadtree taken from [EGS05]. Consider the standard quadtree T of the input set S. We may assume that the center of the root square (containing the set S) is the origin and half the side length of any square in T is a power of 2.

Define an interesting square of a quadtree to be one that is either the root or that has at least two non-empty quadrants. Any quadtree square p containing at least two points contains a unique largest interesting square q in T. The compressed quadtree explicitly only stores the interesting squares, thus removing all the non-interesting squares and deleting their empty children. So for each interesting square p, the compressed quadtree stores 2^d bi-directed pointers, one for each d-dimensional quadrant. If the quadrant contains at least two points, the pointer goes to the largest interesting square inside the quadrant; if the quadrant contains one point, the pointer goes to that point; and if the quadrant is empty the pointer is NULL.

The above description of a compressed quadtree implies that the size of the tree is $\mathcal{O}(2^d \cdot n)$. We will modify the tree in the following way. Instead of storing information about which children contain points and which children are empty, we use a list that contains only the non-empty children. This improves the space complexity to $\mathcal{O}(dn)$, however this modification will increase the cost of a search in the tree since deciding if a child exists or not requires $\mathcal{O}(d)$ time using binary search in the list of at most 2^d children.

For fat regions Q, the skip-quadtree supports $(1+\delta)$-approximate range (counting) queries, i.e. the query range Q is approximated by an extended query range Q_δ. The extended query range Q_δ consists of Q and all points within a distance $\delta \cdot w$ from Q, where w is the diameter of Q. The approximate query counts all points in Q, it either counts or does not count points in $Q_\delta \setminus Q$ and it does not count any point in $\mathbb{R}^d \setminus Q_\delta$. We use Corollary 11 of [EGS05]:

Corollary 4.1 ([EGS05]) *We can answer a $(1+\delta)$-approximate range query with convex or non-convex fat region in Minkowski space in $O(\log n + 1/\delta^{d-1})$ time.*

Since the dimensionality d is considered to be constant in [EGS05], the factor $d \cdot 2^d$ was omitted by the authors when stating the running time in the above corollary. The above discussion of our modifications is summarized in the following corollary.

Corollary 4.2 *Insertion, deletion and search in the modified d-dimensional skip-quadtree using a total of $\mathcal{O}(dn)$ space can be done in $\mathcal{O}(d \log n)$ time. For any $\delta > 0$, a $(1+\delta)$-approximate range counting query for any fat region of complexity $\mathcal{O}(d)$ can be answered in time $T(n) = \mathcal{O}(d^2(\log n + 1/\delta^{d-1}))$.*

4.2 Approximate flocks

The input for the flock problem is a set P of n trajectories $p_1, \ldots, p_n$, where each trajectory p_i is a sequence of τ coordinates in the plane $(x_1^i, y_1^i), (x_2^i, y_2^i), \ldots, (x_\tau^i, y_\tau^i)$, where (x_j^i, y_j^i) is the position of entity p_i at time t_j. We assume that the movement of an entity from its position at time t_j to its position at time t_{j+1} is described by the straight-line segment between the two coordinates, and that the entity moves along the segment with constant velocity.

4.2.1 An equivalent definition of flock

Next we give an alternative and algorithmically more simple definition of a flock.

Definition 4.2 *discrete (m, k, r) - Let there be given a set of trajectories, where each trajectory consists of τ line segments. Let I be a time interval $[t_i, t_j]$, with $j - i + 1 \geqslant k$ and $i \leqslant j \leqslant \tau$. A* flock *in time interval I consists of at least m entities such that for every discrete time-step $t_\ell \in I$, there is a disk of radius r that contains all the m entities.*

Note that the center of a disk does not have to coincide with one of the positions of the entities, see for example the disk D_5 in Fig. 4.2.

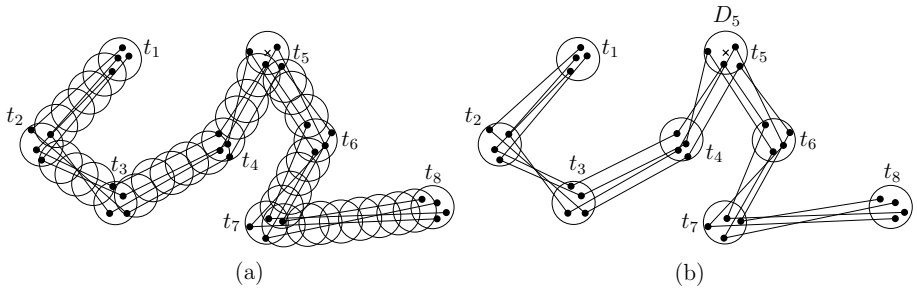


Figure 4.2: A flock of four entities in the time interval $[t_1, t_8]$, according to the definitions of (a) continuos and (b) discrete flocks.

Lemma 4.1 *If the entities move with constant velocity along straight line segments between consecutive positions, then the definitions of continuos (Def. 4.1) and discrete (Def. 4.2) flocks are equivalent.*

Proof. Consider a given time interval $I = [t_1, t_k]$ and assume that F_A and F_B are the set of all flocks in I according to Definition 4.1 and 4.2, respectively. Obviously every flock $f_A \in F_A$ is also a flock in F_B, thus $F_A \subseteq F_B$.

It remains to prove that $F_B \subseteq F_A$. Let f_B be an arbitrary flock in F_B, and let D_ℓ and $D_{\ell+1}$ be disks of radius r that include the entities of f_B at time t_ℓ and $t_{\ell+1}$ respectively, see Fig. 4.3a. It is enough to consider every two discrete time-steps t_ℓ and $t_{\ell+1}$ in I separately.

Next we prove that at every point in time $\gamma \in I' = [t_\ell, t_{\ell+1}]$ there is a disk $\mathcal{D}_\gamma$ that contains all the entities $\{p_1, \ldots, p_m\}$ in f_B. Let c_ℓ and $c_{\ell+1}$ be the centers of D_ℓ and $D_{\ell+1}$, respectively, and let h be the straight-line segment with endpoints at c_ℓ and $c_{\ell+1}$, as illustrated in Fig. 4.3a. An entity q that moves with constant velocity on h has a well-defined position at time $\gamma \in I'$, we denote this position by c_γ. Next we show that the disk $\mathcal{D}_\gamma$ with center at c_γ and radius r contains all the entities of f_B. Let p_i be an arbitrary entity of f_B. Since the movement of q and p_i during I' follows a straight-line and since q and p_i move with constant velocity, the relative trajectory of p_i in relation to q is a straight-line segment as shown in Fig. 4.3b. Since a disk is convex and since $p_i(t_\ell)$ and $p_i(t_{\ell+1})$ are points within D_ℓ and $D_{\ell+1}$, respectively, it holds that $p_i(\gamma)$ must lie within $\mathcal{D}_\gamma$. Consequently, $f_B \in F_A$ and therefore $F_B \subseteq F_A$ which completes the proof of the lemma. ♣

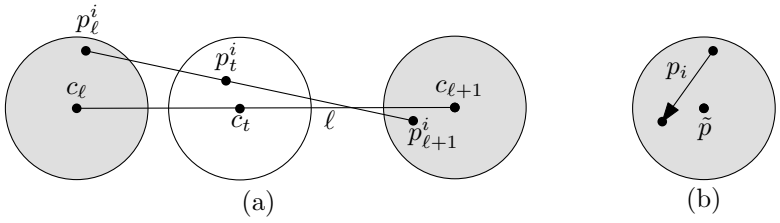


Figure 4.3: Illustration for the proof of Lemma 4.1.

In the remainder of this chapter we refer to Definition 4.2 whenever we talk about flocks. Definition 4.2 immediately suggests a new approach; for each time interval $[t_i, t_{i+k-1}]$ check whether there is a set of m entities $F = \{p_1, \ldots, p_m\}$ that can be covered by a disk of radius r at each discrete time-step in $[t_i, t_{i-1+k}]$. Next we show how this observation allows us to develop an approximation algorithm.

4.2.2 The general approach

When developing an algorithm for flock detection one of the main obstacles that we encountered was to detect flocks without having to keep track of all the objects in a potential flock. That is, when we consider a specific time-step; the number of potential flocks can be very large and the number of objects that one needs to keep track of for each potential flock might be $\Omega(n)$. In general this problem occurs whenever one attempts to develop a method that processes the input time-step by time-step. In this chapter we avoid this problem by transforming the trajectories into higher dimensional space. We then build a tree structure for every possible start time $1, ..., \tau$ and flock length k from scratch, and we then perform counting queries in this tree structure. This might seem like overkill, but both the theoretical results and the experimental bounds support this approach, as long as k is fairly small. Note that the gain is that we only need to count the number of points in a region, instead of keeping track of the actual entities.

The basic idea is to model a 2-dimensional polygonal line with d vertices as a single point in $2d$ dimensions. Formally, the trajectory of an entity p in the time interval $[t_i, t_j]$ is described by the polygonal line

$$p = \langle (x_i, y_i), (x_{i+1}, y_{i+1}), \ldots, (x_j, y_j) \rangle,$$

which we map to the single point $p'(i, j)$ in $2(j - i + 1)$-dimensional space:

$$p' = (x_i, y_i, x_{i+1}, y_{i+1}, \ldots, x_j, y_j).$$

Let p_1 and p_2 be two entities. Now, a key characterization that will help to find flocks is that two entities p_1 and p_2 are close to each other during the time interval $[t_i, t_{i+k-1}]$ if and only if the two points $p'_1(i, i + k - 1)$ and $p'_1(i, i + k - 1)$ are close to each other in $\mathbb{R}^{2k}$. Therefore, the first step when checking whether there is a flock in the time interval $[t_i, t_{i+k-1}]$ is to map the corresponding polygonal lines of all entities to $\mathbb{R}^{2k}$. We now define an (x, y, i, r)-*pipe* which is an unbounded region in $\mathbb{R}^{2k}$. Such a pipe contains all the points that are only restricted in two of the $2k$ dimensions (namely in dimensions i and $i + 1$) and when projected on those two dimensions lie in a circle of radius r around the point (x, y). Formally, a (x, y, i, r)-pipe is the following region:

$$\left\{(x_1, \ldots, x_{2k}) \in \ \mathbb{R}^{2k} \mid \ (x_i - x)^2 + (x_{i+1} - y)^2 \leqslant r^2\right\}$$

Two entities p_1 and p_2 have distance at most r at time-step $j \in \{i, i + k - 1\}$ if and only if there exists a pipe with radius r for the dimensions corresponding to j such that the $2k$-dimensional points p'_1 and p'_2 are contained in that pipe. Hence, two entities p_1 and p_2 have distance at most r at time-steps $i, ..., i + k - 1$ if and only if there exist k pipes with radius r for the dimensions corresponding to $i, ..., i + k - 1$ such that the $2k$-dimensional points p'_1 and p'_2 are contained in all those pipes. This leads to a characterisation of the flock-pattern, which is formalised by the following equivalence.

Equivalence 4.1 *Let $F = \{p_1, \ldots, p_m\}$ be a set of entities and let $I = [t_1, t_k]$ be a time interval. Let $p'_1, \ldots, p'_m$ be the mappings of the trajectories of the entities in F to $\mathbb{R}^{2k}$ w.r.t. I. It holds that:*

$$F \text{ is an } (m,k,r)\text{-flock} \iff$$

$$\exists (x_1, y_1, \ldots, x_k, y_k) \in \mathbb{R}^{2k} : \ \forall p \in F : \ p' \in \bigcap_{i=1}^{k} (x_i, y_i, 2i-1, r)\text{-pipe}$$

4.3 Approximation algorithms

We will now show how approximation algorithms can be obtained by performing a set of range counting queries in higher dimensional space. These algorithms approximate the flock radius r. Here, a Δ-approximation (with $\Delta > 1$) means that every (m,k,r)-flock will be reported, an $(m,k,\Delta r)$-flock may or may not be reported, while no $(m,k,\hat{r})$-flock will be reported, where $\hat{r} > \Delta r$.

4.3.1 Method 'box': A $(\sqrt{8}+\varepsilon)$-approximation algorithm

By Equivalence 4.1 it is fairly straight-forward to develop a $(\sqrt{8}+\varepsilon)$-approximation algorithm, where $\varepsilon > 0$ can be chosen. For each time interval $I = [t_i, t_{i+k-1}]$, where $1 \leqslant i \leqslant \tau - k + 1$, we do the following computations.

For each entity p let p' denote the mapping of the trajectory of p to $\mathbb{R}^{2k}$ with respect to I. We construct a skip-quadtree T for the point set $P' = \{p'_1, \ldots, p'_n\}$. Then, for each point $p' \in P'$ we perform a $(1+\delta)$-approximate range counting query in T with $\delta = \varepsilon/\sqrt{8}$, where the query range $Q(p')$ is a $2k$-dimensional cube of side length $4r$ and center at p'. That is, we approximate the $2k$-dimensional cube which is itself an approximation for the query region. Every such query region containing at least m entities corresponds to an $(m,k,(\sqrt{8}+\varepsilon)r)$-flock as Lemma 4.2 will show. Note that the same flock may be reported several times. We call the method 'box', since the query region is a box.

Lemma 4.2 *The algorithm is a $(\sqrt{8}+\varepsilon)$-approximation algorithm.*

Proof. First we show that every (m,k,r)-flock is reported by the algorithm. Let f be such a flock, e.g. in the time interval $I = [t_i, t_{i+k-1}]$ and let p_f be an arbitrary entity of f. We will prove that the approximation algorithm returns an $(m,k,(\sqrt{8}+\varepsilon)r)$-flock g such that $f \subseteq g$.

According to Definition 4.2 for each discrete time-step t_l in I there exists a disk D_l with radius r that contains the entities in f . The algorithm performs a counting query for each point in P' w.r.t. $[t_i, t_{i+k-1}]$, in particular for p'_f. The query range $Q(p'_f)$ is a $2k$-dimensional cube of side length $4r$ and center at p'_f, where p'_f is the point in $\mathbb{R}^{2k}$ corresponding to p_f. For a discrete point of time t_l, the query range corresponds to a square Q' in two dimensions with center at p and side length $4r$, where the dimensions mark the $x-$ and $y-$positions of the entities at time t_l. As every entity of f has distance at most $2r$ to p_f this implies that every entity in f lies within $Q(p'_f)$. Thus, when p_f is queried, the algorithm reports a flock g such that $f \subseteq g$.

To establish the approximation bound we still have to show that we only report $(m,k,(\sqrt{8}+\varepsilon)r)$-flocks and that no (m,k,r')-flock g where r' exceeds $(\sqrt{8}+\varepsilon)r$ is reported. Let g be a reported flock w.r.t. the time interval $I = [t_i, t_{i+k-1}]$. We have to show that for every time-step t_l in I there exists a disk of radius $(\sqrt{8}+\varepsilon)r$ that contains the entities in g. This follows

trivially by the choice of δ. If we choose δ to be $\varepsilon/\sqrt{8}$, the square of side length $4(1+\delta)r$ is contained in the disk with radius $(\sqrt{8}+\varepsilon)r$ centered at p'_f, as illustrated in Fig. 4.4a. This completes the proof of the lemma. ♣

Lemma 4.3 *The algorithm reports at most τn $(m, k, (\sqrt{8}+\varepsilon)r)$-flocks. It runs in $\mathcal{O}(\tau n k^2(\log n + 1/\varepsilon^{2k-1}))$ time and requires $\mathcal{O}(\tau n)$ space.*

Proof. The number of reported flocks is trivially bounded by n, the number of entities, times τ, the number of time-steps. At each of the $(\tau - k + 1)$ time intervals the algorithm builds a skip-quadtree of the n elements from scratch. In total this requires $\mathcal{O}(\tau k n \log n)$ time according to Lemma 4.2. Then a $(1+\delta)$ approximate-range counting query is performed for each of the n entities; each query requires $\mathcal{O}(k^2(\log n + 1/\varepsilon^{2k-1}))$ time as $\delta = \varepsilon/\sqrt{8}$. Hence, the total time needed to perform all the $n(\tau - k + 1)$ queries is bounded by $\mathcal{O}(\tau k^2 n(\log n + 1/\varepsilon^{2k-1}))$ and thus dominates the running time as stated in the lemma.

The space needed to build the skip-quadtree for each time interval is $\mathcal{O}(\tau n)$, and since we only maintain one tree at a time the bound follows. ♣

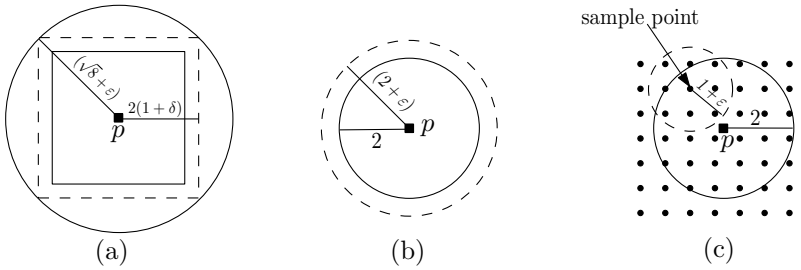


Figure 4.4: Illustration of the query regions of methods box (a), pipe (b) and sample-points (c) for $r = 1$. The approximative query ranges are marked by dashed lines.

4.3.2 Method 'pipe': A $(2+\varepsilon)$-approximation algorithm

The algorithm is similar to the above algorithm. The main difference is that we will use the intersection of k pipes as the query regions instead of the $2k$-dimensional box. For each time interval $I = [t_i, t_{i+k-1}]$, where $1 \leqslant i \leqslant \tau - k + 1$, we do the following computations.

For each entity p let p' denote the mapping of the trajectory of p to $\mathbb{R}^{2k}$ with respect to I. We construct a skip-quadtree T for the point set $P' = \{p'_1, \ldots, p'_n\}$. Then, for each point $p' \in P'$ we perform a $(1+\varepsilon)$-approximate range counting query in T, where the query range $Q(p')$ is the intersection of the k pipes $(x_i, y_i, 2i-1, 2r)$, where (x_i, y_i) is the position of entity p at time-step t_i. We call the method 'pipe', since the query region is the intersection of pipes.

We use the definition of fatness that was introduced by van der Stappen [vdS94]. For convex objects it is basically equivalent to other definitions [AKS95, AFK+92, vK98].

Definition 4.3 ([vdS94]) *Let $\alpha > 1$ be a real value. An object s is α-fat if for any d-dimensional ball D whose center lies in s and whose boundary intersects s, we have* $\text{volume}(D) \leqslant \alpha \cdot \text{volume}(s \cap D)$.

Lemma 4.4 *The intersection of d pipes $(x_i, y_i, 2i-1, 2r)$, $1 \leqslant i \leqslant k$, in $2d$-dimensional space is a bounded convex 4^d-fat region whose boundary consists of $\mathcal{O}(d)$ surfaces that can be described by quadratic functions.*

Proof. W.l.o.g. we assume that the center of the intersection $\mathcal{I}$ of the d pipes is the origin, then $\mathcal{I}$ can be described by the following d inequalities:

$$\begin{aligned} x_1^2 + x_2^2 &\leqslant r^2 \\ x_3^2 + x_4^2 &\leqslant r^2 \\ &\dots \\ x_{d-1}^2 + x_d^2 &\leqslant r^2. \end{aligned}$$

The set of inequalities together with the fact that the inequalities are pairwise independent immediately gives that $\mathcal{I}$ is bounded, convex and its boundary consists of $\mathcal{O}(d)$ surfaces that can be described by quadratic functions. Thus it remains to prove that $\mathcal{I}$ is 4^d-fat. We place an arbitrary ball whose center lies in $\mathcal{I}$ and consider the intersection of the ball and $\mathcal{I}$ projected onto two dimensions, say dimension $2i-1$ and $2i$. The projection of the ball is a disk whose center lies within the projection of $\mathcal{I}$ which is also a disk. Thus, it is obvious that at least $1/4$ of the projected ball lies in the projected region of $\mathcal{I}$. Taking all dimensions into account yields that $\mathcal{I}$ is 4^d-fat. ♣

Recall that since the query range is convex and fat we can use the result stated in Lemma 4.2.

Lemma 4.5 *The algorithm is a $(2+\varepsilon)$-approximation algorithm.*

Proof. The proof follows from the same arguments as used in the proof of Lemma 4.2. When approximately evaluating the query range $Q(p')$ which is the intersection of the k pipes $(x_i, y_i, 2i-1, 2r)$, $1 \leqslant i \leqslant k$ where (x_i, y_i) is the position of entity p at time-step t_i, we test whether there is an $(m, k, (2+\varepsilon)r)$-flock which p is part of. If p is part of an (m, k, r)-flock f in the time interval I, the disk with radius r containing all the entities in f at time-step $t_i \in I$ is contained in the disk with radius $2r$ centered at (x_i, y_i). Thus, when querying p, the algorithm reports an $(m, k, (2+\varepsilon)r)$-flock g with $f \subseteq g$. ♣

Lemma 4.6 *The algorithm reports at most τn $(m, k, (2+\varepsilon)r)$-flocks. It runs in $\mathcal{O}(\tau n k^2 (\log n + 1/\varepsilon^{2k-1}))$ time and requires $\mathcal{O}(\tau n)$ space.*

The proof of Lemma 4.6 is absolutely analogous to the proof of Lemma 4.3.

Remark. A quick comparison between Lemmas 4.3 and 4.6 reveals that even though the approximation factor of the second method is smaller the running time is identical. However, this is a theoretical bound, in practice we chose to implement the $(2+\varepsilon)$-approximation with the help of a compressed quadtree. The reason for this is that the skip-quadtree computes the volume between a d-dimensional cell (orthogonal box) and $Q(p')$, where $Q(p')$ is the intersection of the k pipes, which is possible in theory but hard in practice. The query data structure of a compressed quadtree only checks whether the intersection is non-empty which is much easier to implement. Consequently, the experiments that we performed for the $(8+\varepsilon)$-approximation use the skip-quadtree whild the $(2+\varepsilon)$-approximation use a standard compressed quadtree.

4.3.3 Method 'sample-points': A $(1+\varepsilon)$-approximation algorithm

We use the same approach as for the $(2+\varepsilon)$-approximation but instead of querying only the input points in $\mathbb{R}^{2k}$ we will now query $\mathcal{O}(1/\varepsilon^{2k})$ sample points for each entity point. For each time interval $I = [t_i, t_{i+k'}]$, where $1 \leqslant i \leqslant \tau - k + 1$ and $k' = k - 1$, we do the following computations.

For each entity p let p' denote the mapping of the trajectory of p to $\mathbb{R}^{2k}$ with respect to I. Construct a skip-quadtree T for the point set $P' = \{p'_1, \ldots, p'_n\}$. Let Γ be the intersection points of a regular grid in $\mathbb{R}^{2k}$ of spacing $\varepsilon \cdot r/2$. Each input point p'_i generates the sample set $\Gamma \cap D(p'_i)$, where $D(p'_i)$ is the $2k$-dimensional ball of radius $2r$ centered at p'_i. Clearly, this gives rise to $\mathcal{O}(1/\varepsilon^{2k})$ sample points for each entity p. We call the method 'sample-points', since the query region is based on sample points. A necessary condition for a sample point q to induce an (m, k, r)-flock is that there are at least m entities in the disk D_q of radius $2r$ centered at q. As we generated at most $\mathcal{O}(n/\varepsilon^{2k})$ sample points, this means that we have to check at most $\mathcal{O}(n/(m\varepsilon^{2k}))$ candidate sample points that might define a flock.

Now, we perform a $(1+\varepsilon/(2+\varepsilon))$-approximate range counting query in T for each sample point $q = (x_1, y_1, \ldots, x_k, y_k)$, where the query range is the intersection of the k pipes $(x_i, y_i, 2i-1, (1+\varepsilon/2)r)$, $1 \leqslant i \leqslant k$. We prove the approximation bound:

Lemma 4.7 *The algorithm is a $(1+\varepsilon)$-approximation algorithm.*

Proof. The $(1+\varepsilon/(2+\varepsilon))$-approximation of the range query ensures that no (m, k, r')-flock with $r' > (1+\varepsilon)r$ is reported: as we query pipes of radius $(1+\varepsilon/2)r$, the maximum distance from a grid query point to a counted entity is bounded by $(1+\varepsilon/2)\cdot(1+\varepsilon/(2+\varepsilon))r = (1+\varepsilon)r$.

Next, we show that each (m, k, r)-flock is reported by the algorithm. Assume that f is an (m, k, r)-flock in the considered time interval I. We prove that the approximation algorithm returns an $(m, k, (1+\varepsilon)r)$-flock g such that $f \subseteq g$.

Let $(x_1, y_1, \ldots, x_k, y_k) \in \mathbb{R}^{2k}$ be a point that induces an (m, k, r)-flock f with respect to I. We look only at one time-step $t_i \in I$. By the cell spacing it is obvious that there are sample points $(\ldots, x_i^q, y_i^q, \ldots) \in \Gamma$ such that the Euclidean distance from (x_i^q, y_i^q) to (x_i, y_i) is less than $\varepsilon r/2$. This means that the disk (in $\mathbb{R}^2$) with radius $(1+\varepsilon/2)r$ centered at the projection of q completely contains the disk with radius r centered at (x_i, y_i). Thus, when checking the sample points $(\ldots, x_i^q, y_i^q, \ldots)$ all entities of f are in range for time-step t_i. As this holds analogously for all other time-steps the algorithm reports an $(m, k, (1+\varepsilon)r)$-flock g such that $f \subseteq g$. ♣

Lemma 4.8 *The algorithm reports at most τn $(m, k, (1+\varepsilon)r)$-flocks. It runs in $\mathcal{O}(\frac{\tau n k^2}{m\varepsilon^{2k}}(\log n + 1/\varepsilon^{2k-1}))$ time and requires $\mathcal{O}(\tau n)$ space.*

Proof. The number of reported flocks is trivially bounded by n, the number of entities, times τ, the number of time-steps. At each of the $(\tau - k + 1)$ time intervals the algorithm builds a skip-quadtree of the n elements from scratch. In total this requires $\mathcal{O}(\tau k n \log n)$ time, according to Lemma 4.2. Next a counting query is performed for each of the $\mathcal{O}(n/(m\varepsilon^{2k}))$ candidate sample points in Γ; each query requires $\mathcal{O}(k^2(\log n + 1/\varepsilon^{2k-1}))$ time, thus the total time needed to perform all $n(\tau - k + 1)$ queries is as stated in the lemma.

The space needed to build the skip-quadtree for each time interval is $\mathcal{O}(kn)$, and since we only maintain one tree at a time the bound follows.

♣

4.4 Minimize the number of reported flocks

The general (theoretical) approach described in Section 4.3 has the following disadvantage: As every entity is tested, a flock consisting of exactly m elements can be reported up to m times. This may get even worse if a flock is found whose number of entities exceeds m. Below we briefly discuss three approaches how reporting this redundant information could be avoided. The main idea for all of them is to reduce the number of reported flocks. The last approach abandons the restriction that a flock defining region always has to be disk.

This section is concluded by a brief discussion how to deal with the issue that a flock can be together for more than the required k time steps.

Each entity is part of at most one flock.

In theory one object can be part of many flocks at the same time, while in practice this seems unreasonable. Thus, the first method we propose guarantees that an object belongs to at most one flock at a time.

The strategy for this approach is very simple. If a counting query reports a flock then the entities involved in the flock are marked and the skip-quadtree is updated so that the marked entities will not be counted again. The additional time that we have to spend updating the tree is $\mathcal{O}(nk \log n)$ per time-step, thus $\mathcal{O}(\tau nk \log n)$ in total. The number of reported flocks is trivially bounded by $\tau n/m$.

Each entity is part of at most a constant number of flocks.

The above approach minimizes the number of reported flocks; however, it also overlooks a lot of flocks. Therefore we chose to use a different approach in the experiments which guarantees a higher level of correctness while bounding the number of flocks that an entity may belong to simultaneously.

The idea is that when a flock is found every input point within the query region will be marked, so that no query will be performed with those points as centers. Using a simple packing argument it follows that the maximal number of flocks an entity can be part of during a time-step is bounded by 2^{2k}. By an array we can easily keep track of the points that shall not be checked as aflock-defining centers.

Extending flocks that have been found.

In such an approach we also assume that each entity can only be part of at most one flock. Once a flock is found, we first check whether we can reasonably extend it, which means we may manipulate the disk as flock-defining region if it seems reasonable to join objects closeby. There are many ways to do this that work in practice, however, guaranteed theoretical bounds are hard to prove.

We discuss one possible approach: assume we found a flock f when querying $Q(p)$, the query range of $p \in \mathbb{R}^{2k}$. We then use the skip-tree to run an approximate range reporting query on the query region $Q(p)$ reduced by a factor c, where c is some constant larger than 1. Let $N(p)$ be the set of entities found by this query. Next, we perform the usual approximate range counting queries for each entity $q \in N(p)$. If we find 'another' flock f' for q, we merge f and f', see Figure 4.5a. For the output we will report both, the coordinates $(x_1^p, y_1^p, \ldots, x_k^p, y_k^p)$ and $(x_1^q, y_1^q, \ldots, x_k^q, y_k^q)$ of p and q, respectively. Thus, so far, the flock defining region has

become the union of two disks, then we go on iteratively. If c is 1, the set $N(p)$ would contain all objects in the query range $Q(p)$ and we could upgrade the counting query to a reporting query. However, this would yield strange mergers that are not desired, see Figure 4.5b. For the $(1+\varepsilon)$-approximation we could replace the approximate range reporting query by just checking the neighbouring grid cells of $p \in \Gamma$. These give us the candidates for which a flock could potentially be extended.

As in the first strategy—each entity is part of at most one flock—we mark all the entities of a flock after a flock has been found that cannot be extended any more. Updating the skip-tree with respect to these marks costs $\mathcal{O}(nk \log n)$ per time-step, thus $\mathcal{O}(\tau nk \log n)$ in total.

For each entity we perform at most one approximative range reporting query per time-step, one of each is in $\mathcal{O}(k^2(\log n + 1/\varepsilon^{2k-1}))$, i.e. in total this requires the same amount of time as for all counting queries. We have to set additional marks in the skip-tree in order to prevent that we report a point twice for one time-step. This works analogously as for the 'flock' marks and in the same amount of time. The number of reported flocks is bounded by $\tau n/m$, however, we still need $\mathcal{O}(\tau n)$ space to report them.

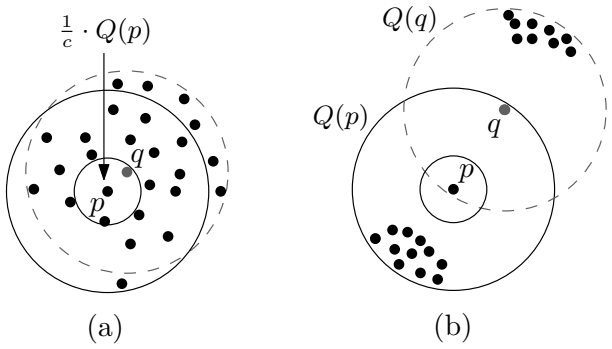


Figure 4.5: (a) the flock found with center p is extended by q, (b) the flock found with center p should not be extended by q.

Prioritizing long-duration flocks.

Furthermore, we would like to report the entire time interval the flock is defined for. That is, if a flock stays together for longer than k time intervals then the longer interval should be reported.

The idea is to keep track of which flock an object belongs to at each step. Then as a preprocessing step of each iteration the algorithm tests if the flock is still together, if so, they are removed from the set of entities and the remaining entities are processed as described above.

Testing whether a flock still exists for a subsequent time step can be done by performing a number of approximate range counting queries in the plane. For this we check every new position of a flock member whether it induces a new flock-defining region that contains enough of the old flock members to extend the flock.

4.5 Experiments

In this section we report on the performed experiments. We describe the experimental setup, i.e. the hard- and software used for the experiments, we briefly explain the methods and we present and discuss the resulting running times with respect to different input parameters.

4.5.1 Setup

We used a Linux operated off-the-shelf PC with an Intel Pentium-4 3.6 GHz processor and 2 GB of main memory. The data structures and algorithms were implemented in C++ and compiled with the Gnu C++ compiler. The running times are reported as seconds. All our trajectories used in the experiments were created artificially. Each trajectory of length k was generated as a single point in $2k$ dimensions. Hence, we now focus from now on on the characteristics of these higher dimensional point sets that were generated. The point sets differ in size (10,000 - 160,000 points; one algorithm was run with more than 1 million points), in length of the time interval (4–16 time-steps) and also in the distribution of the points (uniformly random or clustered).

To ensure that our algorithms indeed find flocks, we arranged 10% of the points in each point-set in such a way that they form randomly positioned flocks. Each of the flocks has $m = 50$ entities in a circle of radius $r = 50$ (hence the number of artificially inserted flocks is 0.002 times the total number of points). As it is unlikely to have flocks that were generated by accident, we inserted those artificial flocks to make sure that the methods correctly find them.

The remaining 90% of the points were randomly distributed, either uniformly or in clusters. Although it is improbable it is possible for these points to interfere or extend our artificial flocks. The purpose of the clustered point sets is that they are more likely to resemble real data, and hence it is interesting to compare the impact of different distributed point sets on the running times of our methods. Each cluster was generated by choosing randomly a cluster center and then distributing (with a Gaussian distribution) a number of points around that center. However, the number, distribution, radius and density of the clusters was chosen such that it is unlikely (although it can happen) to create flocks by accidents, i.e. a cluster is not dense enough to form a flock because its radius is much larger than the flock radius. Choosing the clusters in this way makes a comparison between the results for clustered and uniformly randomly distributed point sets easier, as the differently distributed points can have a strong effect on the height and width of the created tree structures.

In our experiments each point coordinate of an input point is an integer taken from the interval $[0, ..., 2^{13}]$ or $[0, ..., 2^{16}]$. Note that each generated data instance contains the coordinates of points for a certain number of time-steps τ, and in the experiments on that instance, we always looked for flocks of at least $m = 50$ entities in a circle with radius $r = 50$ and of length k with $k = \tau$.

Note that our artificially created instances are in a sense not very realistic: if we translate a randomly created point in $\mathbb{R}^{2k}$ back to the k different positions in the plane, these positions may be fairly chaotic. However, for the functionality of our algorithms this seems to be irrelevant since for them it is more important how dense the points lie. Unfortunately, it is very hard to get any meaningful real-world data.

When performing a range query, ε influences the approximate region to be queried. One could expect that a larger value of ε can lead to shorter running times and more flocks that

are found, because the descent in the tree can be stopped earlier and the query region can become larger. However, apart from very marginal fluctuations, this behaviour could not be observed in initial experiments. Our point sets and therefore our trees in the experiments are rather sparsely filled. Hence, the squares corresponding to most of the leaves in the tree (which correspond to single points in a point set) are still quite large compared to the flock radius r and also to $(1+\varepsilon)r$. Furthermore, it often seems that the point sets are too sparse to find any random flocks. Therefore we refrained from reporting results for different ε and only use $\varepsilon = 0.05$.

4.5.2 Methods

We compare the results of four methods called *box* [$(8+\varepsilon)$-approximation], *pipe* [$(2+\varepsilon)$-approximation], *no-tree* and *pruning*. All of them mark points that were found to belong to a flock, and in the further course of the algorithm these marked points are not used as potential flock centers, see Section 4.4 for a discussion. The output of the algorithms are the centers of the flocks found. The box and pipe method are named after the shape of region that we query for the corresponding method and are explained in Sections 4.3.1 and 4.3.2, respectively.

The no-tree method (which was implemented for the sake of comparison) does not use a tree as underlying structure. It consists of two nested loops: the outer loop runs over all input points and specifies a potential flock center. Tthe inner loop runs over all input points as well and computes the distance between a point and the potential flock center. If there are enough points within a ball (around the potential flock center) of double flock-radius (see the proof of Lemma 4.2 for an explanation why the radius is doubled) then the algorithm reports a flock. Hence, the no-tree method is a 2-approximation.

The pruning method takes advantage of the fact that each flock of a certain length k is also a flock of length $k^* < k$. Therefore all points not involved in flocks of length k^* cannot be involved in flocks of length k. The method works as follows. As a first step we compute flocks of length 4 using the box method. Then we build a new tree containing only those points that were contained in flocks during the first step. This drastically reduces the number of points. We then again apply the box method on the new tree for the entire length k. Applying the box method means that the approximation factor of the pruning method is $(8+\varepsilon)$.

4.5.3 Results

We ran the experiments with few generated point sets for each combination of point-set characteristics, such as number of points, number of time-steps and point distribution. The results were very similar for fixed characteristics and hence the tables below show the numbers for only one collection of point-sets with the specified characteristics. The results of the algorithms are depicted in Table 4.1, where the coordinates of the points are chosen from the interval $\left[0, ..., 2^{16}\right]$. The columns below 'input' specify the number of points and the number of time-steps, and the columns below 'uniformly' and 'clustered' show the number of flocks found and the running times needed when performing the box-, pipe- and no-tree-algorithm on the corresponding input. We also performed the same experiments on point-sets where the coordinates where chosen from $\left[0, ..., 2^{13}\right]$. Table 4.2 shows those results. The results for the method with pruning are given in Table 4.3. Because of the similarity of the results for a different number of time-steps, we only report the results for 16 time-steps in that table. Table 4.4 shows the results of the no-tree method for a large number of time-steps and a

small number of entities. All tables show the results, i.e. the number of flocks found and the running time in seconds, only for $\varepsilon = 0.05$, because no big influence of different values of ε was observed. From our point of view the running times are much more important than the number of flocks found. Hence, the number of flocks are shown here only for the sake of completeness. These numbers are indicated in *italics* in case they deviate from the number of artificially inserted flocks. (In most cases the methods found exactly as many flocks as were artificially inserted.)

4.5.4 Discussion

Flat trees in high dimensions.

One general observation is that the running times of our algorithms are increasing with the number of time-steps (i.e. with the number of dimensions). Recall that an internal node of an octree has 2^d children where d is the number of dimensions. Using 16 time-steps means 32 dimensions which translates to more than 4 billion quadrants, i.e. children of an internal node (in our approach we only store non-empty children in a list, which reduces storage space but increases time complexity). In an experiment with 160,000 points in 32 dimensions it is very unlikely that many of the randomly distributed points (not in flocks) fall into the same quadrant. Therefore the tree is very flat, i.e. have only a very small depth, which results in high running times.

Number of flocks.

Most of the times the algorithms found exactly as many flocks as were artificially put into the point-sets. A few times more flocks were found but only in instances with a small number of time-steps. This is reasonable since if points that are not belonging to an artificially inserted flock, form a flock at all, then it is more likely that this happens for a small number of time-steps.

One case is remarkable (see Table 4.2, on clustered points, $n = 160K$, $k = 4$), where the box method found more than 1300 flocks while the pipe method found only 320 flocks.

This difference can be explained by recalling that the volume of the pipe query region is strictly smaller than the volume of the box query region. Let us consider one single flock F, that was artificially generated. The large number of flocks found by the box method indicates that for that instance the distribution of the points and clusters (in combination with a high number of points and a small coordinate space) is dense enough so that there are many points that lie inside the box query region that is centered at some point in F but not inside the pipe query region that is also centered at some point in F (note that the difference in volume of these two query regions exponentially increases with the number of dimensions). By nature of our methods, many of these points can contribute to the total number of flocks as a flock center, hence, counting F multiple times. On top of that it also might be that the points are dense enough to form random flocks of radius at most $\sqrt{8} + \varepsilon$.

In some of our experiments we observed that the algorithms found fewer flocks than artificially were inserted. This can happen if two flocks are close to each other and fall into one query region and hence will be counted as one flock by the algorithm.

input		uniformly distributed						clustered					
		box		pipe		no-tree		box		pipe		no-tree	
n	k	flocks	time	flocks	time	flocks	time	flocks	time	flocks	time	flocks	time
10K	4	20	0	20	0	20	5	20	0	20	1	20	5
10K	8	20	2	20	1	20	5	20	1	20	0	20	5
10K	16	20	2	20	1	20	6	20	1	20	0	20	5
20K	4	40	1	*41*	0	40	21	40	0	40	1	40	20
20K	8	40	7	40	5	40	21	40	1	40	0	40	22
20K	16	40	13	40	10	40	25	40	3	40	2	40	25
40K	4	80	0	80	1	80	83	80	1	80	0	80	83
40K	8	80	32	80	22	80	87	80	2	80	2	80	87
40K	16	80	62	80	44	80	99	80	8	80	7	80	101
80K	4	160	3	*163*	3	160	332	160	3	160	2	160	332
80K	8	160	129	160	88	160	347	160	6	160	4	160	346
80K	16	160	244	160	182	160	392	160	30	160	29	160	392
160K	4	320	8	*321*	10	320	1326	320	8	320	5	320	1327
160K	8	320	441	320	316	320	1391	320	20	320	15	320	1384
160K	16	320	986	320	768	320	1576	320	102	320	93	320	1564

Table 4.1: Results for $\varepsilon = 0.05$ and a large coordinate space, where trajectories are sparsely distributed, i.e. the position-coordinates are in $[0, ..., 2^{16}]$. The number of flocks is reported and the running time (in seconds).

input		uniformly distributed						clustered					
		box		pipe		no-tree		box		pipe		no-tree	
n	k	flocks	time	flocks	time	flocks	time	flocks	time	flocks	time	flocks	time
10K	4	20	1	20	0	20	5	20	2	20	1	20	4
10K	8	20	8	20	6	20	6	20	2	20	2	20	6
10K	16	20	14	20	11	20	6	20	5	20	11	20	6
20K	4	40	1	40	4	40	20	40	2	40	1	40	20
20K	8	40	52	40	35	40	22	40	6	40	4	40	22
20K	16	40	83	40	58	40	25	40	17	40	44	40	25
40K	4	80	4	80	15	80	83	*81*	6	80	2	80	83
40K	8	80	237	80	166	80	87	80	16	80	21	80	87
40K	16	80	347	80	244	80	99	80	55	80	177	80	99
80K	4	160	10	160	57	160	333	*206*	16	160	8	160	332
80K	8	160	932	160	696	160	348	160	45	160	77	160	348
80K	16	160	1411	160	1124	160	394	160	164	160	594	160	395
160K	4	320	29	320	201	320	1326	*1317*	42	320	27	320	1331
160K	8	320	3179	320	2658	320	1393	320	124	320	238	320	1392
160K	16	320	6015	320	4226	320	1575	320	692	320	2306	320	1576

Table 4.2: Results for $\varepsilon = 0.05$ and a smaller coordinate space, where trajectories are densely distributed, i.e. the position-coordinates are in $[0, ..., 2^{13}]$. The number of flocks is reported and the running time (in seconds).

Coordinate space $[0, ..., 2^{13}]$ vs. $[0, ..., 2^{16}]$.

Somewhat surprising might be that the algorithms ran faster on point-sets with coordinates in $[0, ..., 2^{16}]$ were much faster than those with point-sets with coordinates in $[0, ..., 2^{13}]$ (all other parameters were the same). One explanation is that in a bigger underlying space (i.e. where the coordinates are in $[0, ..., 2^{16}]$) it is more likely that the query region falls into a single square corresponding to a quadtree node. Due to the sparseness of the point-sets the algorithms are likely to find just a single point in that square. On the other hand in a smaller underlying space the query region might intersect more squares, which results in more subsequent queries, which in turn takes more time.

Uniformly distributed vs. clustered data.

When comparing the results of the uniformly distributed point-sets with the clustered point-sets it becomes evident that our tree-based algorithms nearly always perform better on the clustered data. This behaviour could be expected because, as we have seen from the experiments in general, uniformly distributed points result in trees that are rather flat (especially for higher dimensions). Before exploring what this means, recall that an internal node of an original (compressed) quadtree has 2^d nodes as children, where d is the dimension. In our modified quadtree a node will not have pointers to all these children, but will have a list associated that only contains the non-empty children. Now consider e.g. a quadtree for 16 time-steps. Then the root node R of that tree has 2^{32} children. If we store $n = 160,000$ uniformly distributed points (in 32 dimensions) in that quadtree, we will most likely have that all children nodes of R correspond to quadrants of the tree that are very sparsely filled. A quadrant that contains only one point will not create another level in the tree structure. Hence, for high dimensions it is unlikely that we have trees with large height. Querying such a flat tree, however, is expensive as this involves checking all 2^d children or, as in our case, traversing the entire list of non-empty children, which is likely to have $O(n)$ length in high dimensions. It is a 'good balance' between height and width of a tree that allows fast query times. Clustered data sets are more likely to create trees that are deeper on some branches or subtrees, and therefore the algorithm will descent on those subtrees cutting off everything not contained in them. As expected, the no-tree method (which is not using a tree) is not affected by the two different types of data.

No-tree method vs. box method vs. pipe method.

We observe that the no-tree method's running times are quadratic in the number of points and not influenced by the number of time-steps, as expected. On the other hand the box and pipe algorithms are strongly influenced by the number of time-steps and the number of points. As discussed above for high dimensions the box and pipe methods operate on an underlying tree that is very flat. A large query region in combination with a small coordinate space causes their behaviour to become similar (although with a big overhead) to the no-tree method. The difference between the box and pipe method is caused by the different data structure they use. The box method uses the more complex skip-quadtree, while the pipe method uses a compressed quadtree.

Pruning.

Table 4.3 shows the running times of the pruning method for $k = 16$ and $\varepsilon = 0.05$. The impressive impact of the pruning step is illustrated in Figure 4.6 where the running times of the usual box and the pruning method are shown for point sets with coordinates in $[0, ..., 2^{13}]$.

Depending on the density and distribution, even some point-sets with more than 1 million points can be dealt with by the pruning method within a few minutes. Furthermore, we observed that the number of time-steps hardly has an influence on the running times. An exception are the clustered point sets with coordinates in $[0, ..., 2^{13}]$ and a large number of points, where we experienced much longer running times and a strong correlation between the number of time-steps and running time. Also the point distribution (uniformly or clustered) does not affect the running times on point-sets with coordinates in $[0, ..., 2^{16}]$. However, for point-sets with coordinates in $[0, ..., 2^{16}]$, we observe much longer running times for the clustered point-sets. This can be explained by noting that after the pruning step it is likely that the remaining points form a flock also for more time-steps. Therefore, almost every query to the data structure gives a flock and hence, the number of queries drastically decreases. For the clustered point sets with coordinates in $[0, ..., 2^{13}]$, however, the probability of random flocks is much higher. The fact that the pruning method sometimes finds less flocks than the box method can be explained by noting that the pruning method performs two runs of the box method each of which can handle the points in a different order. Therefore the second run of the box method can encounter points which do not belong to any flock.

input		coordinates from $[0, ..., 2^{13}]$				coordinates from $[0, ..., 2^{16}]$			
		uniformly		clustered		uniformly		clustered	
n	k	flocks	time	flocks	time	flocks	time	flocks	time
10K	16	20	0	20	1	20	1	20	0
20K	16	40	1	40	2	40	1	40	0
40K	16	80	3	80	6	80	2	80	2
80K	16	160	11	160	15	160	3	160	3
160K	16	320	30	320	45	320	9	320	9
320K	16	*639*	82	*633*	303	640	26	640	25
640K	16	*1271*	194	*1268*	1796	1280	75	1280	75
1280K	16	*2501*	533	*2507*	9213	2560	249	2560	246

Table 4.3: Results for the pruning method for $\varepsilon = 0.05$. The number of flocks is reported and the running time (in seconds).

4.6 Concluding remarks

In this chapter we have presented different algorithms for finding flock patterns and analysed them theoretically as well as experimentally. From the experiments we have seen that our tree-based algorithms can perform very well. Especially for a small number of time-steps the resulting running times are often very small. However, they depend very much on the characteristics of the input point-sets, which motivates more research and experiments, preferably on real-world data.

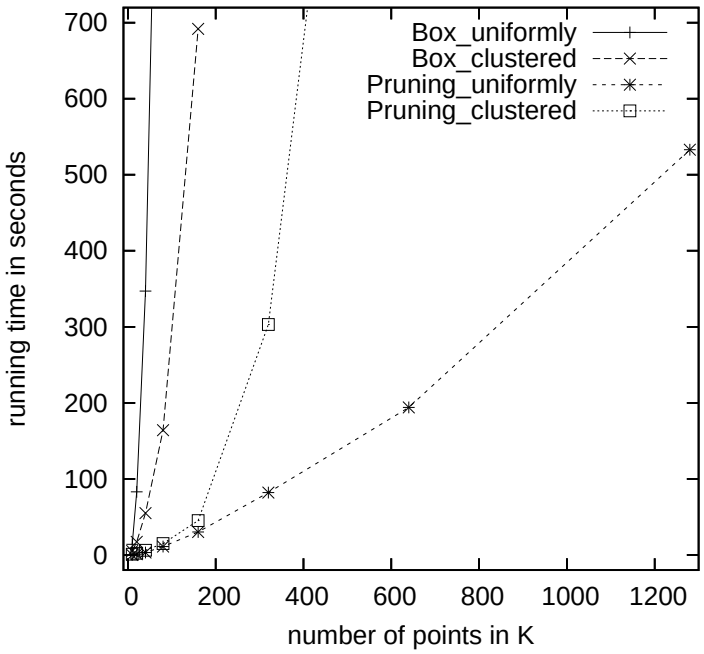


Figure 4.6: Running times for $k = 16$, $\varepsilon = 0.05$ and point coordinates from $[0, ..., 2^{13}]$.

For a larger number of time-steps the no-tree method can be used. This method's running time is mainly influenced by the number of entities and not by the number of dimensions. Table 4.4 shows the performance of this algorithm for up to 40000 entities and up to 1000 time-steps. As we have seen from Tables 4.1 and 4.2, the characteristics (such as distribution and coordinate space) of the point sets have no influence on the running time of the no-tree method and therefore, Table 4.4 only shows the results for uniformly distributed points with coordinates in $[0, ..., 2^{16}]$. We see that even point sets with 1000 time-steps can be searched for flocks of length 1000 within a few minutes.

input	uniformly distributed with coordinates from $[0, ..., 2^{16}]$											
	$k = 32$		$k = 64$		$k = 125$		$k = 250$		$k = 500$		$k = 1000$	
n	flocks	time	flocks	time	flocks	time	flocks	time	flocks	time	flocks	time
10K	20	14	20	16	20	15	20	15	20	18	20	21
20K	40	59	40	67	40	69	40	73	40	85	40	88
40K	80	235	80	265	80	278	80	294	80	332	80	337

Table 4.4: Results of the no-tree method, $\varepsilon = 0.05$. The number of flocks is reported and the running time (in seconds).

Hence, for a small number n of entities and many time-steps, we can use the no-tree method, which has a running time quadratic in n. For many entities and few time-steps k our tree-based methods perform very well. They have a running time exponential in the number

of dimensions of the tree, i.e. they are exponential in k. Thus, we are faced with a trade-off. One approach to tackle the case of many entities and many time-steps has recently been developed by Al-Naymat et al. [ANCG07]. They process the data in a preprocessing step that reduces the number of dimensions (i.e. time-steps) by random projection. In experiments it was shown [ANCG07] that the tree-based methods perform very well on the data with reduced dimensionality. As a conclusion we see that the idea of projecting trajectories into points in higher dimensional space is very viable for finding flocks in spatio-temporal data.

This chapter is a first step towards practical algorithms for finding spatio-temporal patterns, such as flocks, encounters and convergences. Future research should not only include more efficient approaches to compute these patterns but also more complicated patterns, e.g. hierarchical patterns or repetitive patterns.

Chapter 5

A Geometric Dispersion Problem

This chapter completes the *analyzing* part and the whole thesis. Though the input, given unit disks in a rectangular region, do not indicate a geometric network at all, the analysis of a nearest-neighbor graph on a special set of unit disks comprises the key to the solution of the dispersion problem we deal with.

The chapter is based on conference publication [7]: Marc Benkert, Joachim Gudmundsson, Christian Knauer, Esther Moet, René van Oostrum and Alexander Wolff: It was invited to the special issue of the *International Journal of Computational Geometry and Application* and is currently under review.

5.1 Introduction

The geometric dispersion problem we consider in this chapter is placing points in a restricted area of the plane such that the points are located as far away as possible from a set of points that is already present, and simultaneously as far away from each other as possible as well. This problem has applications in surveying, in non-photorealistic rendering, and in obnoxious facility location.

In surveying one may be interested in certain soil or water parameters. For a given domain, measurements of some parameter are known for a given set of locations, and one has the means to do extra measurings or set up a number of new measuring stations. It is desired that these new locations are nicely distributed over the area that has not yet been covered by the given fixed locations.

In non-photorealistic systems 3D models are to be rendered, e.g., in an oil painting style. For an example, see Figure 5.1. When a model is rendered "painterly", instead of computing the color of each pixel by ray casting, the color is solely based on lighting and model properties. The idea is to generate a number of brush strokes. Each of these starts at a selected pixel that defines the color for all pixels that are covered by this brush stroke. The aim is to distribute the locations of the brush strokes more or less evenly.

In the general facility location problem a set of customers is given that is to be served from a set of facilities. The goal is to place a number of facilities in such a way that the distance to the closest facility is minimized over all customers. In other words, facilities are desirable, and customers like to be close to them. In obnoxious, or undesirable, facility location problems, the opposite is true: customers now consider it undesirable to be in the proximity of these facilities, and the goal is to maximize the distance to the closest facility

Figure 5.1: The packing problem we consider in this chapter occurs in non-photorealistic rendering, e.g., when using an oil painting style.

over all customers. Examples of such undesirable facilities are, for instance, nuclear power plants or garbage dumps.

There are several models for and variants of the problem, see the surveys by Cappanera [Cap99] and Tóth [T04], and the monographs [Rog64, ZT99]. The problem we will study is a special case of the following problem.

Problem 1 (PackingScaledTranslates) *Let $\alpha \in (0, 1]$ be a fixed real, let $S \subset \mathcal{R}^2$ be a shape of positive area, and let αS be the result of scaling S at the origin with a factor of α. Given an area $A \subset \mathcal{R}^2$, pack at least m disjoint translates of αS into A, where m is the maximum number of disjoint translates of S that can be packed into A.*

Note that we do not know the value of m a priori. The special case that we study

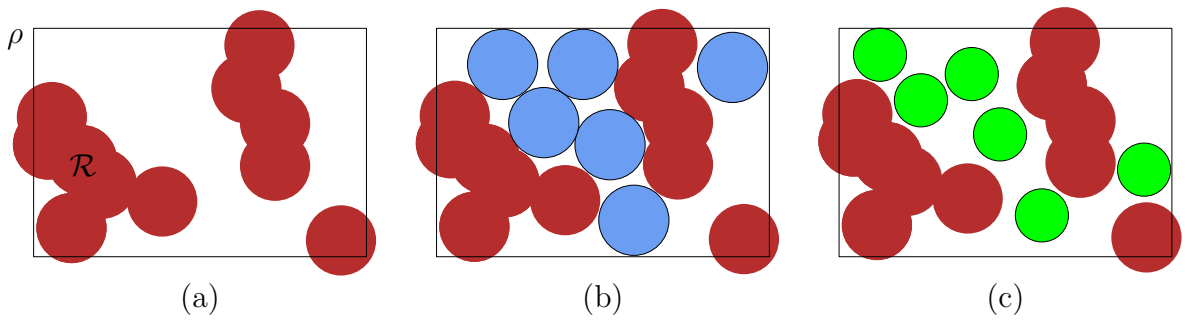


Figure 5.2: (a) The input: a rectangle ρ and a set $\mathcal{R}$ of (possibly overlapping) unit disks. (b) An optimal solution where $m = 6$ unit disks are packed in $\rho \setminus \bigcup \mathcal{R}$. (c) A solution with m disks of radius 3/4.

is referred to as the PackingScaledDisks problem. In this case S is the unit disk and $A = \rho \setminus \bigcup \mathcal{R}$ is a given rectangle ρ minus a set $\mathcal{R}$ of n given obstacle disks (see Figure 5.2). It is easy to see (and we will give the proof in Section 5.2) that greedily placing radius-α disks in A solves PackingScaledDisks for $\alpha \leq 1/2$. The main contribution of this chapter is a polynomial-time algorithm for $\alpha = 2/3$.

In their pioneering work Hochbaum and Maass [HM85] described a polynomial-time approximation scheme (PTAS) for the problem of packing the maximum number of translates of a fixed shape into a region given by the union of cells of the unit grid. The problem is known to be NP-hard [FPT81].

Even though the corresponding geometric dispersion problem looks very similar, inapproximability results have been shown. Baur and Fekete [BF01] investigated the complexity of different variants of the geometric dispersion problem, one of which is a special case of PackingScaledTranslates where S is the axis-parallel unit square and A is a rectilinear polygon with n vertices. We refer to this problem as PackingScaledBoxes. Baur and Fekete showed that it cannot be solved in polynomial time for $\alpha > 13/14$ unless $\mathcal{P} = \mathcal{NP}$.

They also gave a 2/3-approximation for PackingScaledBoxes. Their algorithm can be sketched as follows. First they compute a set S of at least $2m/3$ disjoint unit squares in the given polygon A, using the PTAS of Hochbaum and Maass. Then they use the space occupied by the squares in S to pack at least m squares of side length 2/3 in A. Their algorithm runs in $O(n^{40} \log m)$ time. If the user does not insist on an explicit list of all squares in the output, the running time becomes $O(n^{38})$, i.e., strongly polynomial.

The two main steps of our 2/3-approximation for PackingScaledDisks are conceptually the same as those of Baur and Fekete's algorithm. To make it work for disks, however, we need a more refined analysis. First of all we have to adjust the PTAS of Hochbaum and Maass to packing disks. From this PTAS we need $8m/9$ unit disks. We also need a more involved geometric argument that guarantees sufficient space for the radius-2/3 disks. It is based on a matching in a nearest-neighbor graph of the unit disks. As the algorithm of Baur and Fekete, our algorithm is exclusively of theoretical interest—the involved constant (hidden by the big-Oh notation) is extremely large: roughly 1620^{1620}, see Section 5.4.

For completeness we mention that Ravi et al. [RRT94] considered two *abstract* dispersion problems to which they refer as MAX-MIN and MAX-AVG. In those problems a set of n objects with their pairwise distances and a number $k < n$ is given, and the task is to pick k of the given objects such that the minimum (resp., average) distance among them is maximized. It is known that both versions are NP-hard. Ravi et al. showed that unless $\mathcal{P} = \mathcal{NP}$ there is no constant-factor approximation for MAX-MIN in the case of arbitrary distances. For the case that the distances fulfill the triangle inequality they gave factor-2 and factor-4 approximation algorithms for MAX-MIN and MAX-AVG, respectively. The former is optimal. Wang and Kuo [WK88] investigated a variant of MAX-MIN where the objects are points in Euclidean space. They gave an $O(\max\{kn, n \log n\})$-time algorithm for the one-dimensional case and showed that it is NP-hard to solve the two-dimensional case. For MAX-AVG, Ravi et al. [RRT94] solved the one-dimensional case within the same time bound while the two-dimensional problem is still open.

We show that greedily packing works for $\alpha \leq 1/2$ in the next section. We outline our algorithm for $\alpha = 2/3$ in Section 5.3. We give the details in Sections 5.4–5.8, and conclude in Section 5.9.

5.2 A simple greedy strategy

Here we present a simple greedy approach that guarantees a 1/2-approximation. We use the term r-disk as shorthand for a disk of radius r. Consider an initially empty set $\mathcal{B}$. The greedy algorithm will iteratively add 1/2-disks to $\mathcal{B}$ until no more disks can be added. More exactly, in each round a 1/2-disk D is added to $\mathcal{B}$ if D lies entirely within ρ and does not intersect any of the disks in $\mathcal{B}$ or $\mathcal{R}$.

Observation 5.1 *For $\alpha \leq 1/2$ greedily packing region A solves* PackingScaledDisks *in $O((m+n)^2)$ worst-case or in $O((m+n)\log(m+n))$ expected time.*

Proof. The following simple charging argument shows that we will place at least m disks in this way. Consider an arbitrary (optimal) placement Π of m disjoint unit disks in A. We charge a disk D in Π whenever the center of an α-disk falls into D. We claim that all disks in Π get charged at least once during the execution of the greedy algorithm. Assume this was not the case. Then we could place a α-disk concentrically into an uncharged unit disk. This would contradict the termination condition of the greedy algorithm. Thus each of the m disks in Π has in fact been charged. Since the placement of a α-disk causes the charging of at most one disk in Π, the greedy algorithm places at least m α-disks as stated.

The greedy algorithm can be implemented as follows. First, we compute the union of the disks in $\mathcal{R}$ scaled by a factor of two. Second, we intersect this union with a copy of ρ shrunken by one unit in each direction. The resulting region—call it A'—contains the centers of all unit disks that lie in ρ and do not intersect any disk in $\mathcal{R}$. Then we pick any vertex v on the boundary of A', subtract the unit disk centered at v from A' and repeat until A' is empty. As Aurenhammer [Aur88] has observed, (the boundary of) a union of disks can be represented via the *power diagram* of the disks. A power diagram is a generalization of a Voronoi diagram where the distance used is the *power distance* instead of the Euclidean distance. The power distance of a point p from a disk of radius r and center c is defined by $|pc|^2 - r^2$. Given s disks of arbitrary radii, their power diagram can be computed incrementally, either in $O(s^2)$ worst-case time [AE84] or by randomized algorithms for *abstract Voronoi diagrams* in $O(s \log s)$ expected time [KMM93]. Thus, the greedy algorithm can be implemented to run in $O((\tilde{m}+n)^2)$ worst-case or in $O((\tilde{m}+n)\log(\tilde{m}+n))$ expected time, where $\tilde{m} \geq m$ is the number of disks in the output.

The lemma follows by observing that $\tilde{m} = O(m+n)$. This holds again due to a simple charging argument: if we charge each disk placed by the greedy algorithm to the closest disk in $\Pi \cup \mathcal{R}$, then each of the $m+n$ disks in this set gets charged at most a constant number of times. ♣

Clearly the greedy algorithm for $\alpha = 1/2$ works also for other shapes that are convex and point-symmetric. The running time, however, will depend on how fast unions of these shapes can be computed incrementally.

5.3 Algorithm outline

We now give a rough outline of our 2/3-approximation algorithm for the problem PackingScaledDisks, see Algorithm 5.3. We refer to our algorithm as DiskPacker. For $r > 0$ and a set $R \subseteq \mathcal{R}^2$ let the *r-freespace* of R, denoted $\mathcal{F}_r(R)$, be the set of the centers of all r-disks

DISKPACKER($\rho, \mathcal{R}$)

Compute the 1-freespace $\mathcal{F}_1 = \mathcal{F}_1(\rho \setminus \bigcup \mathcal{R})$ and the extended 1-freespace $\mathcal{F}_1^{\otimes}$.
Use the PTAS of Hochbaum and Maass [HM85] to compute a set of at least $8/9 \cdot m$ disjoint unit disks in $\mathcal{F}_1^{\otimes}$.
Greedily fill possible gaps in the remaining freespace with further unit disks as long as this is possible.
Let $\mathcal{B}$ be the set of unit disks placed by PTAS and post-processing.
Define a metric $\text{dist}(\cdot, \cdot)$ on $\mathcal{B}$.
Compute the nearest-neighbor graph $G = (\mathcal{B}, \mathcal{E})$ with respect to dist.
Find a sufficiently large matching in G.
for each pair $\{D, D'\}$ of unit disks in the matching **do**
    Place three 2/3-disks in a region defined by D and D'.
for each unmatched unit disk $D \in \mathcal{B}$ **do**
    Place one 2/3-disk in D.
return $\mathcal{B}_{2/3}$, the set of all placed 2/3-disks.

Figure 5.3: The framework of our algorithm.

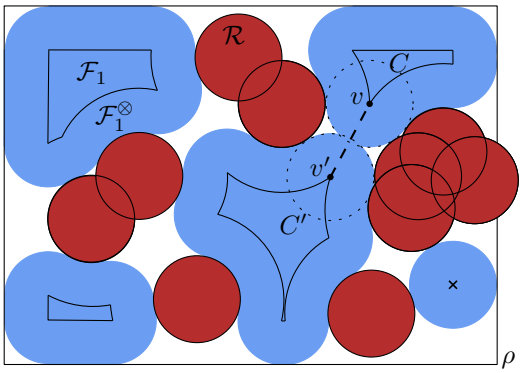


Figure 5.4: The 1-freespace $\mathcal{F}_1^{\otimes}$ (light shaded) and a shortcut vv' (dashed) between the connected components C and C' of $\mathcal{F}_1$.

that are completely contained in R. Let the *extended r-freespace* $\mathcal{F}_r^{\otimes}(R)$ be the Minkowski sum of $\mathcal{F}_r(R)$ and an r-disk.

Recall that $A = \rho \setminus \bigcup \mathcal{R}$. We first compute the sets $\mathcal{F}_1 = \mathcal{F}_1(A)$ and $\mathcal{F}_1^{\otimes} = \mathcal{F}_1^{\otimes}(A)$, see Figure 5.4. Then, we apply the PTAS of Hochbaum and Maass [HM85] to $\mathcal{F}_1^{\otimes}$. For any positive integer t, the PTAS places at least $(1 - 1/t)^2 \cdot m$ unit disks into $\mathcal{F}_1^{\otimes}$, where m is the maximum number of unit disks that can be packed into $\mathcal{F}_1^{\otimes}$. For more details, see Section 5.4. As we aim for a set of at least $8m/9$ unit disks we set $t = 18$ since this is the smallest integer for which $(1 - 1/t)^2$ exceeds $8/9$. After running the PTAS we greedily fill possible gaps in the remaining freespace with further unit disks. Let $\mathcal{B}$ be the set of unit disks placed by PTAS and post-processing, and let $m' \geq 8m/9$ be the cardinality of $\mathcal{B}$.

Then we compute a set $\mathcal{B}_{2/3}$ of at least $9m'/8 \geq m$ disjoint 2/3-disks in A. We obtain $\mathcal{B}_{2/3}$ in two steps. First, we compute a matching in the nearest-neighbor graph $G = (\mathcal{B}, \mathcal{E})$ of $\mathcal{B}$ (see Section 5.6) with respect to a metric $\text{dist}(\cdot, \cdot)$ that we will specify in Section 5.5. Second, we define a region for each pair of unit disks in the matching such that (a) we can place three

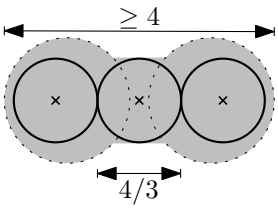


Figure 5.5: Packing three 2/3-disks (bold) in the region (shaded) spanned by a pair of unit disks (dotted).

2/3-disks in each region (see Figure 5.5) and (b) the regions are pairwise disjoint. The main part of the chapter (see Section 5.7) is proving that property (b) actually holds. For each unmatched unit disk we place a 2/3-disk such that their centers coincide, see Section 5.8. Next we describe each step of Algorithm DiskPacker in more detail.

5.4 Adjusting the PTAS of Hochbaum and Maass

The PTAS of Hochbaum and Maass [HM85] packs a near-optimal number of squares into a grid-aligned rectilinear region. In this section we detail how to adjust their PTAS to our situation: we want to pack disks instead of squares, our region A is not rectilinear but a rectangle with holes that are unions of disks, and we do not have an underlying grid.

For packing (square) objects of size $k \times k$ into some given region R, the shifting technique of Hochbaum and Maass [HM85] works as follows. Let t be a positive integer. Take a square grid of cell size $kt \times kt$. For each grid cell that intersects R, solve the corresponding local packing problem in $f(t)$ time optimally, where f is some function. Let the set of all packed objects be $L_{0,0}$. Then shift the grid by the vector (ki, kj) for $i = 0, \ldots, t-1$ and $j = 0, \ldots, t-1$. Each shifted grid $\Gamma_{i,j}$ yields a solution $L_{i,j}$. Let OPT be an optimal solution and let $\mathrm{OPT}_{i,j}$ be the set of all objects in OPT that do not intersect the boundaries of the grid cells in $\Gamma_{i,j}$. By the pigeon-hole principle there is a pair $(i^\star, j^\star)$ such that $\mathrm{OPT}_{i^\star,j^\star}$ contains at least $(1 - 1/t^2)$ times as many objects as OPT. Note that each solution $L_{i,j}$ contains at least as many objects as $\mathrm{OPT}_{i,j}$. Thus $L_{i^\star,j^\star}$ is a $(1 - 1/t^2)$-approximation of OPT. Hochbaum and Maass state that each local packing problem can be solved in $f(t) = \tilde{N}^{t^2}$ time, where $\tilde{N}$ is the number of objects that can be packed in the corresponding $(k \times k)$-square. This yields a total running time of $O(k^2 t^2 N^{t^2})$, where N is the number of unit squares needed to cover region R.

Now we detail how to apply the shifting technique to the problem of packing unit disks in the region $A = \rho \setminus \bigcup \mathcal{R}$. Note that a unit disk fits into a (2×2)-square. Thus if we choose $k = 2$, the above analysis concerning the approximation factor carries over. It remains to show how to solve the local packing problems optimally. So let S be a grid cell of size $(2t \times 2t)$ that intersects A and that may contain or intersect obstacle disks, i.e., disks in $\mathcal{R}$. Let OPT_S be an optimal solution for packing unit disks in S.

We first make OPT_S canonical as follows. We go through the disks in OPT_S from left to right. For each disk D we move D as far left as possible – either horizontally or following the boundary of obstacle disks or previously moved disks. If D happens to hit another disk D' such that their centers lie on the same horizontal line, we move D along the lower half of the boundary of D' until D hits another object or until D reaches the lowest point of D'. In the former case D has reached its destination, in the latter case we continue to move D horizontally until the next event occurs, which we treat as before. If D hits the left edge of

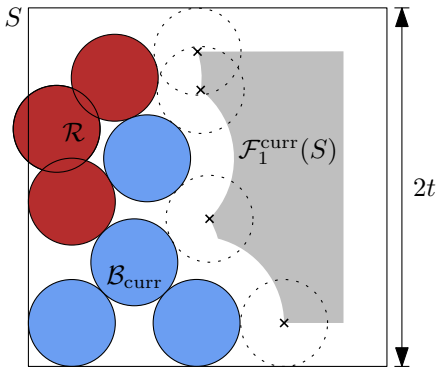


Figure 5.6: Example situation for one step in the incremental procedure of the disk-placing PTAS.

S, we move D downwards along the edge until D hits either another disk or the bottom edge of S.

This process yields the following. Every disk D in the canonical version OPT'_S of OPT_S touches at least two other objects (disks or square edges) with its left side (including the south pole). These are the objects that stop D from moving further to the left, and they completely determine the position of D. Using this fact we now compute OPT'_S by generating *all* canonical solutions, each of them by incrementally adding unit disks from left to right.

Let n_S be the number of obstacle disks that intersect S. These disks are contained in a $((2t+4)\times(2t+4))$-square. In 1910 Thue [Thu10] proved that the density of any arrangement of non-overlapping unit disks in the plane is at most $\pi/\sqrt{12}$, the ratio of the area of the unit disk to the area of the circumscribed regular hexagon. Using Thue's result, we know that $n_S \leq (\pi/\sqrt{12}) \cdot (2t+4)^2 < 5t^2$ for $t \geq 12$. In each step of our incremental procedure we have at most $\max\{n_S+1, 2t\}$ choices to place a unit disk such that its left side touches two previously placed objects (again including obstacles and edges of S). Figure 5 shows a set $\mathcal{R}$ of red disks and a set $\mathcal{B}_{\text{curr}}$ of blue disks that have already been placed. The set of possible choices to place the next disk has been marked by dotted circles. Their centers correspond to locally leftmost points in $\mathcal{F}_1^{\text{curr}}(S)$, the 1-freespace in S with respect to the obstacle disks and the previously placed disks (see the shaded region in Figure 5.6). Let $\tilde{N}$ be the number of disks in OPT_S. Again by Thue [Thu10], we know that $\tilde{N} < 5t^2$. Thus there are at most $f'(t) = (5t^2)^{5t^2}$ choices, which can be enumerated within $O(f'(t))$ time. For the PTAS, this yields a running time of $O(t^2 \cdot (N/t^2) \cdot (5t^2)^{5t^2}) = O(N)$, where $N = O(n+m)$ denotes the area of ρ. This is due to the fact that for one position of the grid we have $O(N/t^2)$ local packing problems, and there are $O(t^2)$ grid positions. Recall from the previous section that we have to apply the PTAS for $t = 18$. Thus we have a polynomial-time algorithm, but the constants hidden by the big-Oh notation are extremely large.

If after applying the PTAS the remaining 1-freespace $\mathcal{F}_1^{\text{curr}}(\rho)$ of the whole instance is not empty, we greedily place further unit disks centered on points in $\mathcal{F}_1^{\text{curr}}(\rho)$ until $\mathcal{F}_1^{\text{curr}}(\rho)$ is empty. This is to make sure that small components (especially those which offer space for only one unit disk) are actually packed optimally. We will need this for our final counting argument, see Section 5.8. Let $\mathcal{B}$ be the set of all disjoint unit disks that are placed by the PTAS and in the greedy post-processing phase. Let m' be the cardinality of $\mathcal{B}$.

5.5 The freespace and a metric on unit disks

We briefly recall the setting. We are given a set $\mathcal{R}$ of n unit disks whose centers lie in a rectangle ρ, see Figure 5.2(a). The disks in $\mathcal{R}$ are allowed to intersect. Let $A = \rho \setminus \bigcup \mathcal{R}$. We first compute the freespace $\mathcal{F}_1 = \mathcal{F}_1(A)$. According to Aurenhammer [Aur88] the union of n disks can be computed in $O(n \log n)$ time and the complexity of its boundary is linear in n. We apply Aurenhammer's algorithm to the disks in $\mathcal{R}$ scaled by a factor of 2. Then we intersect the resulting union with ρ shrunken by 1 unit. This yields $\mathcal{F}_1$ and $\mathcal{F}_1^{\otimes}$ in $O(n \log n)$ time.

Next, we want to introduce a metric $\mathrm{dist}(\cdot,\cdot)$ on unit disks in $\mathcal{F}_1^{\otimes}$. The idea of our algorithm is to use the connected components of $\mathcal{F}_1$ to identify all maximal regions where we can place 2/3-disks. To guarantee that all such regions are discovered we need to join components of $\mathcal{F}_1$ that are not connected but still can hold 2/3-disks in the space between them. This is the idea behind the following definition.

Definition 5.1 *Let C and C' be two connected components of $\mathcal{F}_1$, and let v and v' be vertices on the boundaries of C and C', respectively. We say that the straight-line segment vv' is a* shortcut *if $|vv'| \leqslant 2/3 \cdot \sqrt{11} \approx 2.21$, where $|pq|$ denotes the Euclidean distance of points p and q. Let $\mathcal{S}(C, C')$ be the set of all shortcuts induced by C and C'. We set*

$$\mathcal{F}_1^{+} = \mathcal{F}_1 \cup \bigcup_{\substack{C,C' \in \mathcal{F}_1 \\ s \in \mathcal{S}(C,C')}} s.$$

Figure 5.4 depicts $\mathcal{F}_1$, $\mathcal{F}_1^{\otimes}$, and a shortcut vv'. Throughout this chapter we will use upper-case letters to denote disks and the corresponding lower-case letters to denote their centers. Now, we are ready to define our metric for a connected component of $\mathcal{F}_1^{+}$, see Figure 5.7.

Definition 5.2 *Let D and D' be unit disks that lie in $\mathcal{F}_1^{\otimes}$. Let d and d' be their respective centers. The distance* $\mathrm{dist}(D, D')$ *of D and D' is the length of the geodesic $g(d, d')$ of d and d' with respect to $\mathcal{F}_1^{+}$. The* tunnel $T(D, D')$ *of D and D' is the union of all points in A within distance* 1 *of a point on $g(d, d')$.*

From the definition of $\mathcal{F}_1^{\otimes}$ it is easy to see that any 2/3-disk $D_{2/3}$ centered at a point of $g(d, d')$ does not intersect any disk in $\mathcal{R}$. (This will also follow from Lemma 5.2.) Thus $D_{2/3}$ is contained in the tunnel $T(D, D')$. Since $\mathcal{R}$ is the union of a set of unit disks the geodesic between two points in $\mathcal{F}_1^{+}$ can only consist of line segments and arcs of radius 2, see Figure 5.7(b).

Recall that our algorithm computes a matching in the nearest-neighbor graph $G = (\mathcal{B}, \mathcal{E})$ induced by the metric $\mathrm{dist}(\cdot,\cdot)$ on the set $\mathcal{B}$ of unit disks that we get from the PTAS by Hochbaum and Maass. For each pair $\{D, D'\}$ of disks in the matching we now define a region $\mathcal{A}(D, D')$ into which we later place three 2/3-disks as in Figure 5.5. An obvious way to define this region would be to take the union of all 2/3-disks centered at points of the geodesic between d and d' in $\mathcal{F}_{2/3}$. Our definition is slightly more involved (for illustration see Figure 5.7), but it will simplify the proof of the main theorem in Section 5.6. The theorem states that two such regions are disjoint if the corresponding unit disks are pairwise disjoint. This makes sure that the 2/3-disks which we place into the regions are disjoint.

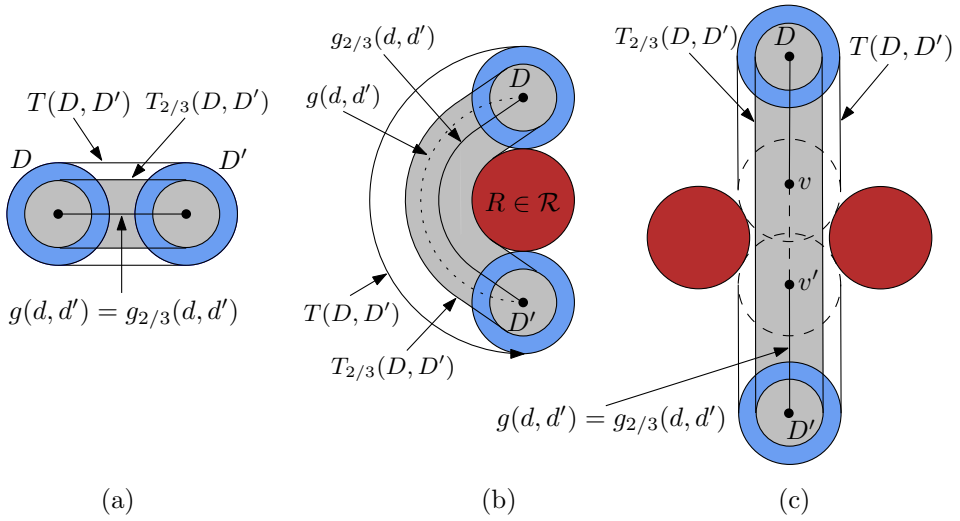


Figure 5.7: The geodesic $g(d, d')$ (a) in the unrestricted case, (b) in the presence of obstacles, and (c) in case of a shortcut.

Definition 5.3 *Let D and D' be unit disks in $\mathcal{F}_1^{\otimes}$. Let $g_{2/3}(d, d')$ be a geodesic from d to d' in $\mathcal{F}_{2/3}(T(D, D'))$. Then the* 2/3-tunnel *$T_{2/3}(D, D')$ of D and D' is defined as all points in A within distance 2/3 of a point on $g(d, d')$. Finally define the* placement region *$\mathcal{A}(D, D')$ of D and D' be $D \cup D' \cup T_{2/3}(D, D')$.*

According to Chang et al. [CCK+05] the geodesics $g(d, d')$ and $g_{2/3}(d, d')$ from d to d' can be computed in $O(n^2 \log n)$ time.

5.6 The nearest-neighbor graph

Recall that m is the maximum number of disjoint unit disks that fit in $\mathcal{F}_1^{\otimes}$. From (our variant of) the PTAS of Hochbaum and Maass [HM85] we get a set $\mathcal{B}$ of $m' \geq 8m/9$ disjoint unit disks in $\mathcal{F}_1^{\otimes}$. Then we compute the nearest-neighbor graph $G = (\mathcal{B}, \mathcal{E})$ induced by the metric $\text{dist}(\cdot, \cdot)$. We consider G a directed graph, where an edge (C, D) is in G if D is the nearest neighbor of C, for $C, D \in \mathcal{B}$. In case of a tie, we pick any of the nearest neighbors of C, so every vertex in G has exactly one outgoing edge.

As for the algorithm we next find a matching in G and place three 2/3-disks in the placement region $\mathcal{A}(C, D)$ for each pair $\{C, D\}$ in the matching. Finally we place a 2/3-disk for each unmatched disk in $\mathcal{B}$. In Section 5.8 we show that in this way we place at least $9m'/8 \geq m$ disks of radius 2/3 in total.

For nearest neighbor graphs of points in Euclidean space it is well-known that the maximum degree of the graph is bounded by 6. However, our setting is quite different so for completeness we include the following lemma.

Lemma 5.1 *The nearest-neighbor graph $G = (\mathcal{B}, \mathcal{E})$ with respect to* dist *is plane and has maximum degree 7.*

Proof. The following standard argument shows that G is plane. Assume that (C, D) and (E, F) are in G and that $g(c, d)$ and $g(e, f)$ intersect. Let p be one of the intersection points

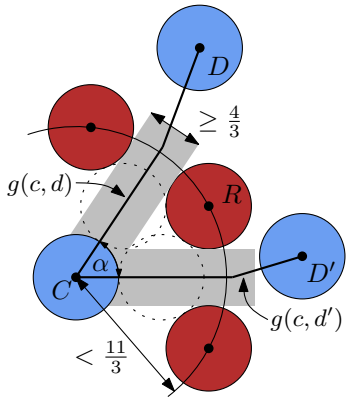


Figure 5.8: Illustration for the proof of Lemma 5.1.

of $g(c,d)$ and $g(e,f)$. This implies that the lengths of $g(c,p)$, $g(d,p)$, $g(e,p)$ and $g(f,p)$ must all be equal. Then, however, either $g(c,d)$ or $g(e,f)$ cannot be a geodesic—a contradiction to (C,D) or (E,F) being in G.

It remains to show the degree bound. Let C be an arbitrary unit disk in $\mathcal{B}$. Since by construction of G the outdegree of each vertex is 1, it suffices to bound the indegree of C by 6.

Let $D, D' \in \mathcal{B}$ be two disks for which (D,C) and (D',C) are in $\mathcal{E}$. Consider the geodesics $g(c,d)$ and $g(c,d')$. By construction they end in C with a straight-line segment, see Figure 5.8. Now, it is enough to show that the angle α between $g(c,d)$ and $g(c,d')$ in C is larger than $360°/7$. We analyze the setting for which α is minimized. Note that if D and D' touched C, then we would immediately have $\alpha \geq 60°$ since D and D' are disjoint. To minimize α, the disks D and D' must lie at some distance from C, but such that the inequalities $\mathrm{dist}(C,D) \leqslant \mathrm{dist}(D,D')$ and $\mathrm{dist}(C,D') \leqslant \mathrm{dist}(D,D')$ are still obeyed. Hence there must be some disk in $\mathcal{R}$ that lies in the wedge-like region between the geodesics $g(c,d)$ and $g(c,d')$. Among these disks let R be the one closest to C.

Due to the way the disks in $\mathcal{B}$ have been placed, R cannot be very far from C, otherwise there would be a disk in $\mathcal{B}$ closer to D or D' than C. For the same reason, there must be other disks in $\mathcal{R}$ on the opposite sides of $g(c,d)$ and $g(c,d')$. The disks in $\mathcal{R}$ must leave a corridor of width at least $2 \cdot 2/3$, otherwise there would not be a shortcut between them that the geodesics can use. This yields $|rc| < 11/3$. If the distance were larger, the presence of an additional disk in $\mathcal{B}$ would immediately contradict (D,C) and (D',C) being in $\mathcal{E}$, see the indicated dotted disks in Figure 5.8. By construction the distance of any point on the geodesics $g(c,d)$ and $g(c,d')$ to r is at least $5/3$. Now some simple trigonometry yields $\alpha > 52.2°$, which is greater than $360°/7 \approx 51.4°$. If we repeat the above construction we can place at most six disks $D, D', D'', \ldots$ such that C is the nearest neighbor of all of them. ♣

From now on we call $\{C,D\} \subseteq \mathcal{B}$ a *nearest pair* if (C,D) or (D,C) is an edge in G, i.e., if D is closest to C or C is closest to D (among the disks in $\mathcal{B}$). Recall that the placement region $\mathcal{A}(C,D)$ of C and D was defined as $C \cup D \cup T_{2/3}(C,D)$. As a nearest pair $\{C,D\}$ can be in the matching, we have to prove the following two statements:

(i) three 2/3-disks fit into $\mathcal{A}(C,D)$ and

(ii) for any nearest pair $\{E,F\}$, where C, D, E, and F are pairwise disjoint, it holds that

$$\mathcal{A}(C,D) \cap \mathcal{A}(E,F) = \emptyset.$$

Note that we do not have to care whether $\mathcal{A}(C,D)$ intersects $\mathcal{A}(C,E)$ because a matching in G contains at most one pair out of $\{C,D\}$ and $\{C,E\}$. Three 2/3-disks obviously fit into $\mathcal{A}(C,D)$ since C and D do not intersect, see Figure 5.5. Thus, statement (i) is true. The remaining part of this chapter will focus on proving statement (ii).

5.7 Placement regions of nearest pairs are disjoint

By the definition of shortcuts, unit disks whose centers lie in different components of $\mathcal{F}_1^+$ do not intersect. This immediately yields that $\mathcal{A}(C,D) \cap \mathcal{A}(E,F) = \emptyset$ holds for $\{C,D\}$ and $\{E,F\}$ being nearest pairs from different components. Thus it remains to show that $\mathcal{A}(C,D) \cap \mathcal{A}(E,F) = \emptyset$ for nearest pairs $\{C,D\}$ and $\{E,F\}$ that lie in the same component.

We split the proof into two parts. The first part (Lemma 5.4) shows that $T_{2/3}(C,D)$ does not intersect any disk other than C and D. The second part (Theorem 5.1) shows that no two 2/3-tunnels $T_{2/3}(C,D)$ and $T_{2/3}(E,F)$ intersect. We start with two technical lemmas that we need to prove the first part.

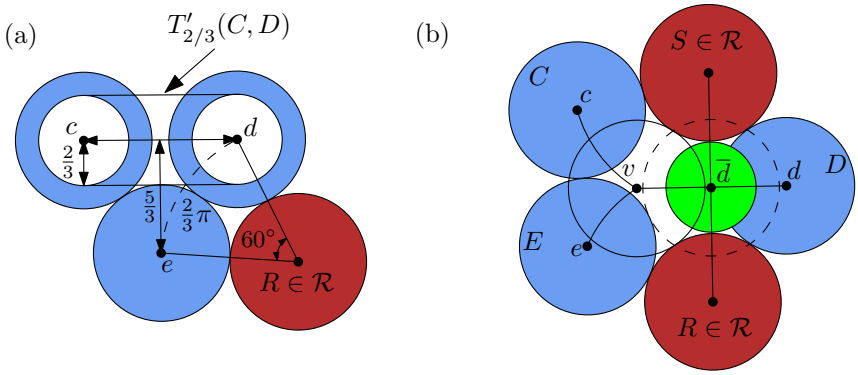


Figure 5.9: Illustrations for (a) the proofs of Lemmas 5.2–5.3 and (b) case 2 in the proof of Lemma 5.4.

Lemma 5.2 *Let C and D be two unit disks in $\mathcal{F}_1^{\otimes}$. If $|cd| \leqslant \frac{2}{3}\sqrt{11}$ then $g_{2/3}(c,d)$ is a straight-line segment.*

Proof. Let $T'_{2/3}(C,D)$ be the Minkowski sum of a 2/3-disk and the line segment cd, see Figure 5.9(a). If $g_{2/3}(c,d)$ is not a line segment, then a disk E in $\mathcal{B} \cup \mathcal{R}$ intersects $T'_{2/3}(C,D)$. We establish a lower bound on $|cd|$ for this to happen. Note that C, D and E are pairwise disjoint as C and D are disks in $\mathcal{B}$.

Clearly, the minimum distance between c and d is attained if E and $T'_{2/3}(C,D)$ only intersect in a single point and furthermore, both E and C as well as E and D intersect in a single point. This means that $|ce| = |de| = 2$. Moreover, the Euclidean distance between e and the straight-line segment cd is $1 + \frac{2}{3} = \frac{5}{3}$. By Pythagoras' theorem we calculate $|cd|$ to be at least $\frac{2}{3}\sqrt{11}$. This means that $T'_{2/3}(C,D)$ is contained in $A = \rho \setminus \bigcup \mathcal{R}$.

If C and D belong to different components of $\mathcal{F}_1$, they must be connected via a shortcut according to Definition 5.1. Thus, $g_{2/3}(c,d)$ is a line segment. ♣

Lemma 5.3 *Let D and E be two unit disks in $\mathcal{F}_1^{\otimes}$ that touch each other. Then* $\mathrm{dist}(D, E) \leqslant \frac{2}{3}\pi$.

Proof. Let D and E be two unit disks in $\mathcal{F}_1^{\otimes}$ that touch each other, as illustrated in Figure 5.9(a). The length of the curve $g(D, E)$ is maximized if there is an obstacle disk R that touches D and E and no shortcut could be taken. In this case $g(D, E)$ describes a circular arc of radius 2 spanning 60°, thus its length is $\frac{1}{6} \cdot 2 \cdot 2\pi = \frac{2}{3}\pi$. ♣

Now, we are ready to prove the first part:

Lemma 5.4 *Let $\{C, D\} \subseteq \mathcal{B}$ be a nearest pair. No disk of $\mathcal{B} \cup \mathcal{R} \setminus \{C, D\}$ intersects $T_{2/3}(C, D)$.*

Proof. From the definition of freespace and Definitions 5.2 and 5.3 it immediately follows that neither $T(C, D)$ nor $T_{2/3}(C, D)$ are intersected by a disk in $\mathcal{R}$. Thus, it remains to prove that apart from C and D no disk in $\mathcal{B}$ intersects $T_{2/3}(C, D)$.

Without loss of generality, assume that C is the disk in $\mathcal{R}$ closest to D. The proof is by contradiction, i.e., we assume that there is a disk $E \in \mathcal{B}$ that intersects $T_{2/3}(C, D)$.

First, we move a unit disk on $g(C, D)$ from the position of D to the first position in which it hits E. Denote the disk in this position by $\overline{D}$. We claim that the length of $g(\overline{d}, e)$, within $\mathcal{F}_1^{+}$, is shorter than $g(c, \overline{d})$. This obviously contradicts C being the nearest neighbor of D, and would thereby complete the proof of the lemma.

We have to consider two cases for the upper bound on the length of $g(\overline{d}, e)$.

Case 1: If $\overline{d}$ is in $\mathcal{F}_1$ and there is an obstacle disk $R \in \mathcal{R}$ that touches $\overline{D}$ and E, then the length of $g(\overline{d}, e)$ is maximized and $g(\overline{d}, e)$ is an arc of radius 2 and spanning 60°. Lemma 5.3 yields $g(\overline{d}, e) \leqslant \frac{2}{3}$. It might be that $g(\overline{d}, e)$ does not lie entirely within $\mathcal{F}_1$. However, as $\overline{D}, E \subseteq \mathcal{F}_1^{\otimes}$, there must then be a shortcut that would shorten the length of $g(\overline{d}, e)$ even further.

Next we give a lower bound on the length of $g(c, \overline{d})$. Since E touches $T_{2/3}(C, D)$ and $\overline{D}$ it follows that C and $\overline{D}$ are disjoint, otherwise E could not intersect $T_{2/3}(C, D)$. Consequently $C, \overline{D}$ and E are pairwise disjoint and, according to Lemma 5.2, the Euclidean distance between c and $\overline{d}$ is greater than $\frac{2}{3}\sqrt{11}$. Putting the two bounds together we get:

$$g(\overline{d}, e) \leqslant \frac{2}{3} < \frac{2}{3}\sqrt{11} < g(c, \overline{d}).$$

Case 2: If $\overline{d}$ is not in $\mathcal{F}_1$ then $\overline{d}$ must lie on a shortcut vv' and the unit disk $\overline{D}$ intersects at least one disk in $\mathcal{R}$. Let v be the endpoint of the shortcut (v, v') closest to E and let $\overline{D}_{2/3}$ be the 2/3-disk centered at $\overline{d}$. Note that v must lie in the same component as e, thus $g(\overline{d}, e)$ consists of a straight-line segment from $\overline{d}$ to v followed by the geodesic $g(v, e)$. The length of $g(\overline{d}, e)$ is maximized if the angle $\angle v\overline{d}e$ is maximized. This is the case if a unit disk $R \in \mathcal{R}$ touches E and $\overline{D}_{2/3}$ and a unit disk $S \in \mathcal{R}$ touches $\overline{D}_{2/3}$ such that $\overline{d}, r$ and s are collinear, as shown in Figure 5.9(b). By parametrization it follows that the length of $g(\overline{d}, e)$ is maximized if the length of $\overline{d}v$ is maximized, which is bounded by $1/3\sqrt{11}$, according to Lemma 5.2. By Pythagoras' theorem we can now compute the coordinates of e, they are $\approx (-1.8182, -0.8333)$, where the coordinate system is fixed by $\overline{d} = (0, 0)$ and $r = (0, -5/3)$. The geodesic $g(v, e)$ consists of an arc of radius 2, applying the cosine theorem then yields that the length of $g(v, e)$ is approximately 1.1105, which gives that $g(\overline{d}, e) < \frac{1}{3}\sqrt{11} + 1.105$.

Next we need a lower bound on the length of $g(c, \overline{d})$. Using the same ideas as above, the position of c can be computed by Pythagoras' theorem to be $\approx (-1.9356, 1.1632)$ and the

length of the geodesic $g(\overline{c}, v)$ is at least 1.4607 using the cosine theorem. Thus,

$$g(\overline{d}, e) \leqslant \frac{1}{3}\sqrt{11} + 1.105 < \frac{1}{3}\sqrt{11} + 1.4607 < g(c, \overline{d}).$$

Since $g(\overline{d}, e)$ has been shown to be shorter than $g(c, \overline{d})$, in all cases, $\overline{d}$ must be closer to c than to e, which is a contradiction to the initial assumption. This completes the proof of the lemma.

Note that the disks involved in this construction can be moved such that the lengths of $g(\overline{d}, e)$ and $g(c, \overline{d})$ changes. However, the above construction minimizes their difference.

♣

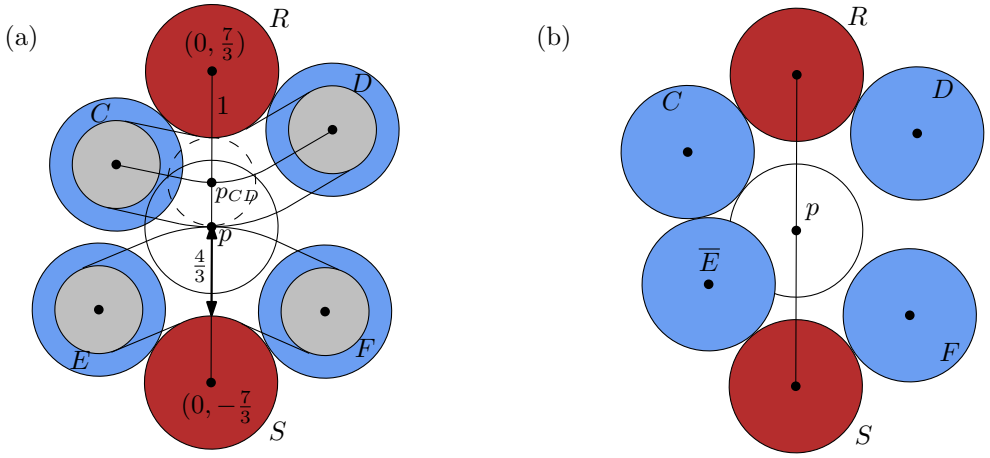


Figure 5.10: Illustrating the proof of (a) Theorem 5.1 and (b) case (ii) in the proof of Theorem 5.1.

Lemma 5.4 proves that no other disks apart from C and D intersect $T_{2/3}(C, D)$. It remains to prove that no two $\frac{2}{3}$-tunnels $T_{2/3}(C, D)$ and $T_{2/3}(E, F)$ intersect.

Theorem 5.1 *Let $\{C, D\}, \{E, F\} \subseteq \mathcal{B}$ be two nearest pairs such that C, D, E and F are pairwise disjoint, it holds that $T_{2/3}(C, D) \cap T_{2/3}(E, F) = \emptyset$.*

Proof. The proof is by contradiction again. We ssume that $T_{2/3}(C, D)$ and $T_{2/3}(E, F)$ intersect and show that this would either contradict $\{C, D\}$ or $\{E, F\}$ being nearest neighbors. Obviously it is enough to exclude the case that $T_{2/3}(C, D)$ and $T_{2/3}(E, F)$ intersect in a single point. We first characterize such an instance which helps us to conduct the contradiction proof.

For the sake of completeness we first have to exclude the case that $g(c, d)$ and $g(e, f)$ use the same shortcut: if they did, the geodesics $g_{2/3}(c, d)$ and $g_{2/3}(e, f)$ would intersect which immediately yields that $g(c, d)$ and $g(e, f)$ would intersect–a contradiction to Lemma 5.1.

Thus, we can assume that the intersection point p of $T_{2/3}(C, D)$ and $T_{2/3}(E, F)$ lies in $\mathcal{F}_1$ and neither $g(c, d)$ nor $g(e, f)$ takes a shortcut containing p. This in turn means that we can assume that no shortcut is taken at all. We observe that at least one of the disks $\{C, D, E, F\}$ intersects the unit disk τ with center p; otherwise there would be another disk in $\mathcal{B}$ located in the space between C, D, E and F which would immediately contradict $\{C, D\}$ as well as $\{E, F\}$ being nearest pairs. Without loss of generality, let C be a disk that intersects τ.

Let p_{CD} be the point on $g_{2/3}(C,D)$ such that $|pp_{CD}| = 2/3$, see Figure 5.10(a). Define p_{EF} similarly. We will assume that there is a vicinity of p_{CD} and p_{EF} in which $g_{2/3}(C,D)$ and $g_{2/3}(E,F)$ are arcs. The case when one vicinity of p_{CD} and p_{EF} is a straight line is easier and can be handled using similar arguments.

The curvature of $g_{2/3}(C,D)$ and $g_{2/3}(E,F)$ in a vicinity of p_{CD} and p_{EF} induces the existence of two disks $R, S \in \mathcal{R}$ as illustrated in Figure 5.10(a). Since R and S forces the curvature of $g_{2/3}(C,D)$ and $g_{2/3}(E,F)$ we may introduce the following coordinate system. The origin is p and the coordinates of r and s are $(0, \frac{7}{3})$ and $(0, -\frac{7}{3})$, respectively.

As a consequence of Lemma 5.2 we get that $g_{2/3}(C,D)$ and $g_{2/3}(E,F)$ start with a straight-line segment of length at least $\frac{1}{3}\sqrt{11}$, again see Figure 5.10(a). Thus, the curvature of $g_{2/3}(C,D)$ in p_{CD} infers that $|cp_{CD}| \geqslant \frac{1}{3}\sqrt{11}$ holds, which means that C either lies completely to the left of the y-axis or to the right. This holds analogously for the other disks. Without loss of generality, we assume that C and E lie to the left of the y-axis and D and F lie to the right, see Figure 5.10(a).

Note that we have to take into account the exact relationship behind the pairs $\{C,D\}$ and $\{E,F\}$ being nearest pairs, e.g., C could be the nearest neighbor of D, or D could be the nearest neighbor of C. We will prove the following:

(i) $\operatorname{dist}(C,E) < \operatorname{dist}(E,F)$

(ii) $\operatorname{dist}(C,E) < \operatorname{dist}(C,D)$

(iii) $\operatorname{dist}(D,F) < \operatorname{dist}(C,D)$

Item (i) says that C is closer to E than F is. Thus, in order for $\{E,F\}$ to be a nearest pair, E must be the nearest neighbor of F. We use this fact to show that (ii) and (iii) hold. Together, (ii) and (iii) comprise the contradiction: (ii) says that D is not the nearest neighbor of C, while (iii) says that C is not the nearest neighbor of D. Hence, $\{C,D\}$ cannot be a nearest pair.

(i): To prove that $\operatorname{dist}(C,E) < \operatorname{dist}(E,F)$ we will argue that $T(E,F)$ intersects C, i.e., there is a unit disk $\overline{E}$ whose center lies on $g(E,F)$ that intersects C and not F. Let $\overline{E}$ be defined by the left and bottommost point $\overline{e}$ on $g(E,F)$ such that $\overline{E}$ intersects C. This is illustrated in Figure 5.10(b). The proof of (i) can then be completed by showing that $\operatorname{dist}(C,\overline{E}) < \operatorname{dist}(\overline{E},F)$.

First we prove that there exists a position of $\overline{e}$ such that $\overline{E}$ intersects C, i.e., a unit disk cannot pass between C and S without intersecting C. This could only be achieved by maximizing $|cs|$. Recall that C intersects τ, thus, $|cs|$ is maximized if C touches R and τ, i.e., C takes its left and topmost position, as shown in Figure 5.10(b). Using Pythagoras' theorem we can compute the coordinates of c for this setting to be $\approx (-1.62, 1.17)$. From now on we will omit the sign $\approx$ when stating results of the calculations. Hence, it holds that $|cs| \leqslant 3.86$ which in turn yields that no unit disk can pass between C and S since this would require $|cs| \geqslant 4$.

Next, we minimize the distance $\operatorname{dist}(\overline{E},F)$ in order to get F to be closer to $\overline{E}$ than to C. For this, $\overline{E}$ should take its rightmost position touching C. This position is attained if C is as far as possible from S, i.e., $\overline{E}$ takes position $(-1.62, 1.17)$ again. Using Pythagoras' theorem, the coordinates of $\overline{e}$ is $(-1.29, -0.80)$. This means that $|\overline{e}p_{EF}| \geqslant 1.29$ and thus $\operatorname{dist}(\overline{E},F) \geqslant 1.29 + \frac{1}{3}\sqrt{11}$ as $|p_{EF}F| \geqslant \frac{1}{3}\sqrt{11}$ holds. According to Lemma 5.3 $\operatorname{dist}(C,\overline{E}) \leqslant \frac{2}{3}\pi$,

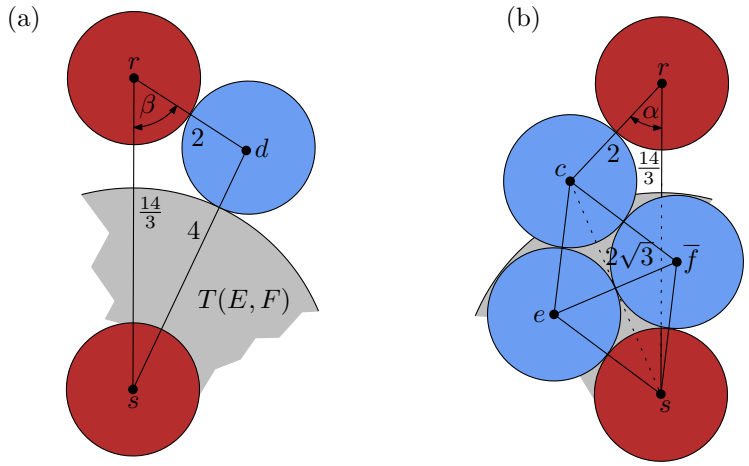


Figure 5.11: Illustration of the proof of the lower bound on $|cd|$ in case (ii).

and we have:

$$\operatorname{dist}(C, \overline{E}) \leqslant \frac{2}{3}\pi < 1.29 + \frac{1}{3}\sqrt{11} \leqslant \operatorname{dist}(\overline{E}, F),$$

which concludes (i).

(ii): To prove that $\operatorname{dist}(C, E) < \operatorname{dist}(C, D)$ always holds we first establish a lower bound on $\operatorname{dist}(C, D)$ and then an upper bound on $\operatorname{dist}(C, E)$. For the lower bound on we try to push C and D as close as possible together under the restriction that E can still be the nearest neighbor of F. To minimize $\operatorname{dist}(C, D)$, C should take its right and bottommost position and D should take its left and bottommost position.

For the bound on D we only use the fact that D is not allowed to intersect the tunnel $T(E, F)$. If it does, we would get $\operatorname{dist}(D, E) < \operatorname{dist}(E, F)$ by a similar argument as in (i). (Here, the corresponding point $\overline{f}$ can even lie further to the right than $(1.29, -0.80)$ as D does not have to intersect τ.) However, $\operatorname{dist}(D, E) < \operatorname{dist}(E, F)$ together with (i) would immediately contradict $\{E, F\}$ to be a nearest pair. Disk D takes its left and bottommost position without intersecting $T(E, F)$ if D touches R and is infinitesimal close to $T(E, F)$. For simplicity we assume that D touches $T(E, F)$, see Figure 5.11(a). Standard trigonometric calculations give the left and bottommost coordinates of d to be $(1.70, 1.29)$.

For the right and bottommost position of C we use the following arguments. Let $\overline{f}$ be the rightmost point on $g(E, F)$ such that $\overline{F}$ touches either C or E. We use that E has to be touched by C otherwise C is closer to F than E. We compute the right and bottommost position of C if $\overline{F}$ touches C and E at the same time, see Figure 5.11(b). Note, that this actually yields a position in which C is closer to F than E (with respect to our metric dist). Again, standard trigonometry gives that the right and bottommost coordinates of c is $(-1.35, 0.86)$.

Now a lower bound on $|cd|$ is the Euclidean distance between $(1.70, 1.29)$ and $(-1.35, 0.86)$ which is 3.08. We obtain the final lower bound on $\operatorname{dist}(C, D)$ by noting that both C and D touch R, if not the Euclidean distance between c and d could be shortened, hence the geodesic has to follow the circular arc around R, thus we get $\operatorname{dist}(C, D) > 3.49$.

By Pythagoras' theorem we can also compute the coordinates of $\overline{f}$ to be $(0.25, -0.35)$ – we will need them in the proof of (iii).

To prove (ii) it remains to show an upper bound on $\operatorname{dist}(C, E)$ which is less than 3.49. We

try to push C and E as far away from each other as possible, under the restriction that E is still the nearest neighbor of F. It is clear that C has to take its left and topmost position, which is already known as $(-1.62, 1.17)$, from (i), while E should take its left and bottommost position. Again, we only use that on the rightmost point $\overline{f}$ on $g(E, F)$ such that either C or E is touched by $\overline{F}$, it must be E that is touched. First, we compute the rightmost point $\overline{f}$ on $g(E, F)$ where $\overline{F}$ touches C taking position $(-1.62, 1.17)$ and touches S. As we know the coordinates of c and s, we can compute the coordinates of $\overline{f}$ by Pythagoras' theorem. It holds that $\overline{f}$ is $(-0.33, -0.36)$, see Figure 5.12(a). Now, E takes its left and bottommost position if it touches S and $\overline{F}$. Using Pythagoras' theorem again, we get that $e = (-1.87, -1.64)$. This yields the upper bound on $|ce|$ of 2.82. However, $g(C, E)$ may have to curve around S so we get $\text{dist}(C, E) < \pi$. Putting it together we get:

$$\text{dist}(C, E) < \pi < 3.49 < \text{dist}(C, D),$$

and we are done with (ii).

(iii): We use the lower bound on $\text{dist}(C, D)$ that was derived in (ii). Thus, we only have to show an upper bound on $\text{dist}(D, F)$ which is less than 3.49. For the upper bound we try to push D and F as far away from each other as possible, under the restriction that E can still be the nearest neighbor of F. For this, D has to take its right and topmost position while F has to take its right and bottommost position. We can assume that D takes position $(1.70, 1.29)$, the position of D which was responsible for the lower bound on $\text{dist}(C, D)$. This assumption is justified since, if D does not take position $(1.70, 1.29)$, we move D on $g(C, D)$ to this position, say $\overline{D}$, and show that $\text{dist}(\overline{D}, F) < 3.49$. Then, $\text{dist}(D, F) < \text{dist}(C, D)$ also holds since $\text{dist}(C, \overline{D}) > 3.49$.

To maximize $\text{dist}(D, F)$ we need F to take its bottommost position. Consider the disk P' that touches C and S and lies to the right of cs, see Figure 5.12(b). For the right and bottommost position of C, which is $(-1.35, 0.86)$, we already computed the position of P' to be $(0.25, -0.35)$. This implies that D does not intersect P' as $|dp'| > 2$ (recall that the position of D was decided in the previous section). We have shown that C, D and E do not intersect P', however, then F has to intersect P' otherwise there would be another disk in $\mathcal{B}$ located in the space between C, D, E and F which would immediately contradict $\{C, D\}$ as well as $\{E, F\}$ being nearest pairs. The disk F now takes its right and bottommost position if it touches P' and S. Using Pythagoras' theorem the position of F is shown to be $(1.84, -1.55)$.

As a result we get $|df| < 2.84$ but since $g(D, F)$ may have to curve around some other disk we get $\text{dist}(D, F) < 3.18$. Putting it together we get:

$$\text{dist}(D, F) < 3.18 < 3.49 < \text{dist}(C, \overline{D}) \leqslant \text{dist}(C, D),$$

and we are done with (iii) and, hence also the theorem. ♣

5.8 Placing the 2/3-disks

In the previous sections we have detailed how we compute a set $\mathcal{B}$ of $m' \geq 8m/9$ unit disks in A and how we determine the nearest-neighbor graph $G = (\mathcal{B}, \mathcal{E})$ with respect to the metric $\text{dist}(\cdot, \cdot)$. In order to finally place the 2/3-disks we compute a matching in G. Observe that G can consist of more than one connected component. We consider each connected component separately. We start with components of size 1. Let $\mathcal{B}_1$ be the set of all unit disks in $\mathcal{B}$ that

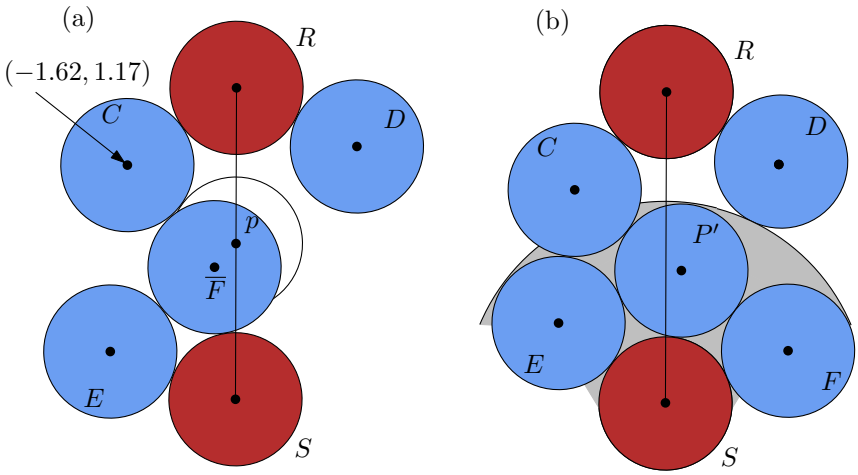


Figure 5.12: (a) The upper bound on $|ce|$ in case (ii). (b) Illustrating the proof of case (iii).

are singletons in G. Observe that each disk in $\mathcal{B}_1$ corresponds to a connected component of $\mathcal{F}_1$ that has been packed optimally since we applied a greedy post-processing step to the PTAS, see Section 5.4. Thus we do not lose anything if we place only one 2/3-disk for each of the $m'_1 = |\mathcal{B}_1|$ singleton unit disks. Let $\mathcal{B}_2 = \mathcal{B} \setminus \mathcal{B}_1$ and let $m'_2 = m' - m'_1$ be the number of disks in $\mathcal{B}_2$. Note that $m'_2 \geq 8(m - m'_1)/9$ since m'_2 is the number of disks in an 8/9-approximation of packing unit disks into those components of $\mathcal{F}_1$ that contain at least two unit disks. Finally let G_2 be the subgraph of G restricted to $\mathcal{B}_2$. We show that G_2 contains a matching with at least $m'_2/8$ edges.

By Lemma 5.1 each connected component $\mathcal{C}$ contains a spanning tree of degree at most 7. In that tree we repeatedly match a leaf with its unique incident vertex. When two vertices are matched they are removed from the tree. This may partition the tree into a forest. For each tree in the forest the process continues iteratively. This yields a matching in $\mathcal{C}$ that contains at least $\lceil |\mathcal{C}|/8 \rceil$ edges. Repeating this argument for each connected component yields a matching in G_2 that contains at least $\lceil m'_2/8 \rceil$ edges.

According to Theorem 5.1 and Lemma 5.4 we can pack three 2/3-disks in $\mathcal{A}(C, D)$ for every matched pair $\{C, D\}$ such that these 2/3-disks are pairwise disjoint. For each of the remaining unmatched disks in G we place one 2/3-disk in each unit disk. Let $\mathcal{B}_{2/3}$ be the set of the 2/3-disks that we placed, including those for the m'_1 unit disks in $\mathcal{B}_1$. By construction no two disks in $\mathcal{B}_{2/3}$ intersect. The cardinality of $\mathcal{B}_{2/3}$ is at least $3\lceil m'_2/8 \rceil + 3\lfloor m'_2/4 \rfloor + m'_1 \geq 9m'_2/8 + m'_1$. Since $m'_2 \geq 8(m - m'_1)/9$, the set $\mathcal{B}_{2/3}$ contains at least m disks, and we can conclude with the following theorem.

Theorem 5.2 *Algorithm* DiskPacker *is a polynomial-time* 2/3*-approximation for the problem* PackingScaledDisks.

5.9 Conclusion

Naturally our main result is purely of theoretic interest. In terms of running time the bottleneck is our disk-packing variant of the PTAS of Hochbaum and Maass, which we apply with an approximation factor of 8/9. To obtain an algorithm with better running time one might

try to consider larger components, or improve the number of matched disks. However, to get a considerable improvement it seems unavoidable to use a completely different approach. Are there fast algorithms for values of α between 1/2 and 2/3?

From a theoretical point of view it would be desirable to reconsider the square case in order to narrow the gap between the lower bound 2/3 and the upper bound 13/14 of Baur and Fekete [BF01]. Their inapproximability result can be adapted to the disk case, but would yield an upper bound very close to 1, leaving an even larger gap than in the square case.

Acknowledgments

We thank Esther Moet and Marc van Kreveld for very helpful comments. We are also very indebted to the anonymous referees, whose comments have helped us a lot.

Deutsche Zusammenfassung

Geometrische Netzwerke

Geometrische Netzwerke sind das Rückgrat bei der Modellierung von Verkehrs-, Waren- und Informationsströmen – ob in der Eisenbahnnetzplanung, dem VLSI-Layout oder der Analyse des Internets. Die Netzwerke werden als geometrische Graphen repräsentiert, in denen die Objekte durch Punkte modelliert werden, die z.B. Städte oder Personen entsprechen, und die Beziehungen selbst Punkte verbindende Linien oder Streckenzüge sind, die z.B. Straßen oder Verwandtschaften entsprechen. Die geometrische Komponente wird dabei zumeist durch die Einbettung der Objekte in die Ebene festgelegt, sofern diese Einbettung nicht selbst gesucht ist wie z. B. bei der Generierung von U-Bahnplänen, in denen die Lage der Bahnhöfe selten der tatsächlichen geographischen Lage entspricht. Die Probleme, die in dieser Disseration behandelt werden, beschränken sich auf den 2-dimensionalen Raum.

Die Dissertation besteht aus zwei Teilen, einem *konstruktiven* und einem *analytischen* Teil. Dabei werden in beiden Teilen ausschließlich Probleme betrachtet, in denen die geometrische Lage der teilnehmenden Objekte bereits durch die Probleminstanz festgelegt ist.

Im konstruktivem Teil werden Algorithmen vorgestellt, die auf den Objekten der Instanz ein Netzwerk konstruieren. Dieses Netzwerk muss gewissen Anforderungen genügen und optimiert im Idealfall ein vorgegebenes Gütekriterium. Ist die Optimierung im Einzelfall zu komplex, geben wir anstelle der Optimallösung lediglich eine Approximation dieser aus. Diese Verfahrensweise ist sämtlichen Problemen gemein, die im konstruktiven Teil betrachtet werden. Es handelt sich hierbei um die Konstruktion von *Manhattan-Netzwerken*, Netzwerken *minimaler Interferenz* sowie das Beschriften einer Punktemenge.

Im analytischen Teil spielen geometrische Netzwerke eine implizite Rolle. Es werden ein Mustererkennungsproblem und ein geometrisches Dispersionsproblem, verwandt mit einem Packproblem, betrachtet. Bei diesen beiden Problemstellungen steht die Analyse eines geometrischen Netzwerks im Vordergrund. Für das Mustererkennungsproblem soll die Verhaltensweise sich bewegender Objekte (z.B. Tiere) analysiert werden. Für das Packproblem birgt die Analyse eines geometrischen Netzwerks den entscheidenden Schlüssel zur Problemlösung.

Einteilen der Probleme in Komplexitätsklassen, Beweisen von Gütegarantien der Approximationen sowie asymptotische Laufzeitanalyse bilden das theoretische Grundprinzip, nach dem die Algorithmen untersucht und klassifiziert werden.

Manhattan-Netzwerke

Ein *Manhattan-Netzwerk* (MN) ist ein Netzwerk, das eine gegebene Menge von Punkten mit vertikalen und horizontalen Segmenten so verbindet, dass jedes Punktepaar eine bezüglich der *Manhattan-Metrik* kürzeste Verbindung besitzt. Ein triviales Manhattan-Netzwerk ist

z.B. das rechteckige Gitter, welches durch die Punktemenge induziert wird. Es scheint kombinatorisch schwer zu sein, ein MN zu finden, das die Gesamtlänge der enthaltenen Segmente minimiert. Der in der Dissertation vorgestellte Algorithmus knüpft an bestehende Forschung an und konnte zum Zeitpunkt der Veröffentlichung den besten Approximationsfaktor von 4 auf 3 verbessern. Das ausgegebene MN ist theoretisch also höchstens dreimal so lang wie ein minimales MN.

Da es möglich ist, das Problem durch ein gemischt-ganzzahliges Programm zu codieren, konnten wir unseren Algorithmus auch praktisch evaluieren. Auf verschiedenen Typen künstlich generierter Eingabedaten erreichte er im schlechtesten Fall einen Approximationsfaktor von durchschnittlich 1.3 und ist somit signifikant besser als die theoretische Gütegarantie erwarten lässt, vergleiche auch Abbildung 6.

In der Zwischenzeit ist es den französischen Forschern Chepoi, Nouioua und Vaxés [CNV05] gelungen, die bestehende Forschung, unsere eingeschlossen, so zu verfeinern, dass sie einen, allerdings deutlich langsameren, Algorithmus mit theoretischer Gütegarantie von 2 angeben können. Dabei wird unser gemischt-ganzzahliges Programm relaxiert und die resultierenden, reellwertigen Variablen werden geschickt gerundet.

Netzwerke minimaler Interferenz

In diesem Szenario sind die in der Ebene gegebenen Punkte Sendestationen, die untereinander Nachrichten austauschen wollen. Da ein Sendevorgang zweier Stationen jedoch Interferenzen verursacht, soll die Menge der möglicherweise gleichzeitig sendenden Paare beschränkt werden. Eine Kante im gesuchten Netzwerk signalisiert die Erlaubnis für die beiden inzidenten Stationen, Nachrichten untereinander zu versenden. Ist die direkte Kommunikation zwischen A und B verboten, so wird die Nachricht über Zwischenstationen geroutet. Ein interferenzminimales Netzwerk enthält eher kurze Kante, da die Interferenz für sie nicht so hoch ist.

Wir konstruieren Netzwerke minimaler Interferenz für die folgenden Kriterien: Zusammenhang, Kürze der geometrischen Länge der Verbindungen und Anzahl der benötigten Zwischenstationen um eine Nachricht von A nach B zu verschicken.

Zudem betrachten wir zwei Modelle: Im exakten Modell nehmen wir an, dass ein Sendesignal genau die Kreisscheibe um den sendenden Punkt erreicht. Im Schätzmodell kennen wir die genaue Position nicht, an der das Signal verloren geht. Wir gehen davon aus, dass das Signal in jedem Fall die Distanz d zur angestrebten Empfangsstation zurücklegt und dann irgendwo in einer Entfernung von d bis $(1 + \varepsilon)d$ zur Sendestation verlorengeht. Das Schätzmodell ermöglicht es uns, die entwickelten Algorithmen für das exakte Modell unter einer sinnvollen praktischen Annahme nochmals zu beschleunigen.

Konstruktion eines bipartiten Netzwerks für ein Beschriftungsproblem

Bei diesem Problem handelt es sich um eine Kombination aus einem Graphenzeichen und einem Beschriftungsproblem: Beschriftet werden sollen Punkte, die in einem Rechteck R liegen. Für die Beschriftung der Punkte werden dabei uniforme Beschriftungsrechtecke reserviert, die rechts und/oder links von R liegen. Gesucht ist eine optimale Zuordnung der Punkte zu den Beschriftungsrechtecken. Das Problem ist somit eine Mischung eines rein geometrischen und eines klassischen Graphenproblems.

Vollständig definiert wird das Problem durch die Art der Verbindung, mit denen die Punkte und Beschriftung verbunden werden. Wir betrachten zwei Verbindungstypen: der

erste Typ besteht entweder aus einem einzigen horizontalen Segment oder beginnt mit einem vertikalen Segment (*po*-leader), an das sich dann ein horizontales Segment anschließt. Beim zweiten Typ ist ein diagonales Segment als erstes Segment erlaubt (*do*-leader), siehe Abb. 7a und Abb. 7b.

Die Optimierungskriterien für eine gute Lesbarkeit der Beschriftung sind zum einen minimale Anzahl von Kantenknicken und zum anderen minimale Gesamtlänge aller Kanten. Für die Längenminimierung verwenden wir hauptsächlich sogenannte Sweepline-Algorithmen, während die Algorithmen für die Knickminierung auf dynamischem Programmieren beruhen. Die Algorithmen wurden implementiert und praktisch evaluiert.

Analyse von Bewegungsmustern in Objektschwärmen

Eine Studie in Alaska bildete die Motivation zu dieser Forschungsarbeit: Karibus wurden mit GPS-Sensoren ausgestattet und deren Zugverhalten über einen längeren Zeitraum beobachtet, um Rückschlüsse auf Herdenbildung der Tiere ziehen zu können.

Die Idee zur algorithmischen Durchführung dieser Analyse ist es, die Zuglinie eines Tieres für τ Zeitschritte zunächst kanonisch als Punkt im $\mathbb{R}^{2\tau}$ einzubetten. Nachdem dies für alle Tiere geschehen ist, werden in $\mathbb{R}^{2\tau}$ Bereichsabfragen (range counting/reporting queries) geeignet durchgeführt, die das Bestehen von Herden erkennen und die Tiere einer Herde ausgeben können. Eine Herde ist dabei eine Ansammlung einer vorgegebenen Anzahl von Tieren, die für einen Mindestzeitraum nahe genug beieinander (Kreisscheibe) bleiben, siehe auch Abbildung 8.

Die Innovation unserer Arbeit gegenüber früheren Algorithmen liegt dabei in der simultanen Betrachtung der Zeitintervalle. Frühere Arbeiten haben jeden einzelnen Zeitschritt unabhängig voneinander analysiert und dabei versucht, durch den momentanen Aufenthaltsort der Tiere sowie die Interpolation ihrer Bewegungsrichtung Rückschlüsse auf Herdenbildungen zu ziehen.

Ein geometrisches Dispersionsproblem

Eine Variante eines klassischen Packungsproblems liegt dieser Arbeit zugrunde: In einer gegebenen Region wurde bereits eine gewisse Anzahl von Einheitskreisscheiben verteilt. Diese können z.B. Messpunkte zur Erhebung der Bodenqualität modellieren. Um den Rest der Region sowohl repräsentativ als auch effektiv zu erschließen, soll die maximale Anzahl weiterer disjunkter Einheitskreisscheiben in das Gebiet eingepasst werden. Dieses Problem ist $\mathcal{NP}$-schwer. Wir approximieren in folgendem Sinn: Wir versuchen, den Radius r der einzupassenden Kreisscheiben zu maximieren, so dass wir garantieren können, dass wir mindestens soviele Kreisscheiben vom Radius r einpassen wie Kreisscheiben vom Radius 1 eingepasst werden können. Sei diese Anzahl κ_1. Dies gelang uns für $r = 2/3$. Für unseren Algorithmus spielt die Konstruktion und Analyse eines sogenannten *Nächsten-Nachbar-Graphen* dabei die entscheidende Rolle.

Zunächst wenden wir einen bereits bestehenden Approximationsalgorithmus [HM85] zum Einpassen von Einheitskreisscheiben an, um eine Menge $\mathcal{K}$ von Einheitskreisscheiben zu erhalten, deren Kardinalität mindestens $8/9 \cdot \kappa_1$ ist. Auf dem Raum, der die Menge $\mathcal{K}$ und die gegebenen festen Kreisscheiben enthält, definieren wir eine Metrik, die – vereinfacht gesagt – die minimale Länge eines kürzesten Weges der Kreisscheibe K in die Position von Kreisscheibe K' angibt, ohne dass dabei eine der vorab eingezogenen Kreisscheiben berührt wird.

Bezüglich dieser Metrik bestimmen wir den Nächsten-Nachbar-Graphen der Menge $\mathcal{K}$. Dieser ist ein geometrisches Netzwerk, in dem die Menge der Kreisscheiben $\mathcal{K}$ der Menge der Knoten des Graphen entspricht und eine Kante eine Nearest-Neighbor-Beziehung der zwei inzidenten Kreisscheiben angibt. Mit Hilfe eines eines maximalen Matchings des Nearest-Neighbor-Graphen können wir eine Menge von disjunkten Kreisscheiben mit Radius $2/3$ anzugeben, deren Anzahl mindestens $9/8$ mal so gross ist wie die der Einheitskreisscheiben in $\mathcal{K}$. Da die Anzahl von Kreisscheiben in $\mathcal{K}$ mindestens $8/9 \cdot \kappa_1$ war, haben wir wie gewünscht insgesamt mindestens κ_1 Kreisscheiben mit Radius $r = 2/3$ platziert.

Glossar

Beim Aufschreiben dieser Dissertation hat mich das Verfassen der Einleitung vor die größte Herausforderung gestellt, da die Inhalte der einzelnen Kapitel größtenteils schon in vorzeigbaren Zustand, zumeist als Journalversionen, vorlagen. Da diese Inhalte in formaler Sprache verfasst sind und viele Definition, Sätze und Formeln enthalten, habe ich mich dazu entschieden, in einem kleinen Nachwort auch nochmal etwas lockerer zu Werke zu gehen und eine kleine Einführung für Nicht-Informatiker sowie ein paar interessante Fakten zum Drumherum der vorgestellten Arbeiten zu geben.

Hinweis! Der Abschnitt 5.9, *Anleitungen für Nicht-Informatiker*, ist also speziell den Leuten gewidmet, die dieses Buch in der Hand halten und nur entfernt etwas mit Informatik zu tun haben.

Während der vier Jahre meiner Doktorandenzeit stand ich oftmals vor der Herausforderung, Freunden oder Bekannten *verständlich* zu erklären, was ich denn eigentlich mache. Ok, mit den Komplexitätsklassen $\mathcal{P}$ und $\mathcal{NP}$ hab ich da natürlich gar nicht erst angefangen, meistens musste die Beschriftung eines U-Bahnlinienplans als praktische Anwendung von geometrischen Netzwerken herhalten. Hier möchte ich nun aber versuchen, die Grundwerkzeuge, mit denen wir in der theoretischen Informatik üblicherweise hantieren, all den Leuten näherzubringen, die nicht tagtäglich damit zu tun haben. Und trotz eindringlicher Warnung meines Betreuers („Da musst du ja bei der Erfindung der Turingmaschine anfangen!") versuche ich mich auch an den Klassen $\mathcal{P}$ und $\mathcal{NP}$.

Anleitungen für Nicht-Informatiker

Während meiner Zeit als Mathematikstudent sind wir immer ironisch mit der Tatsache umgegangen, dass Mathematiker gerne einfache Sachverhalte so verklausulieren, dass sie höchstens noch von anderen Mathematikern, aber unter keinen Umständen von Nicht-Mathematikern verstanden werden können, obwohl wir das selbst natürlich auch gerne mit Freude betrieben haben. Nun, beim Informatiker ist diese Neigung zwar nicht ganz so ausgeprägt, aber aufgrund der Tatsache, dass die meisten Informatiker auch eine mathematische Ausbildung vorzuweisen haben, dennoch vorhanden. Ein perfektes Werkzeug dafür ist die sogenannte *Groß-Oh-Notation*, die oftmals sogar noch von Informatikstudenten im Grundstudium als kryptisch angesehen wird. Im gewissen Sinne breche ich also ein Tabu, wenn ich im Abschnitt 5.9.1 versuche, ihr den Schrecken zu nehmen.

Die Groß-*O*h-Notation erfüllt natürlich auch einen sehr praktischen Zweck: Sie gibt kurz und prägnant an, wie schnell ein Algorithmus (asymptotisch) arbeitet. Ganz grob gesagt kann man dann Probleme in Klassen unterteilen, für die schnelle Algorithmen existieren und für die vermutlich nur langsame Algorithmen existieren. Hierbei handelt es sich um die Komple-

xitätsklassen $\mathcal{P}$ und $\mathcal{NPC}$, mehr dazu im Abschnitt 5.9.2. Abschließend erkläre ich, was ein *Approximationsalgorithmus* eigentlich ist und was man über das Resultat aussagen kann, das er liefert, siehe Abschnitt 5.9.3.

5.9.1 Was ist *asymptotische* Laufzeit?

Zuallerst sollte ein Algorithmus natürlich das korrekte Ergebnis liefern, aber mindestens genauso wichtig ist, wie *schnell* er das tut. Ein korrekter Algorithmus ist nichts wert, wenn die Menge an Daten, die er verarbeiten kann, zu begrenzt ist. Nun könnten wir hergehen und einfach die *absolute* Bearbeitungszeit eines Algorithms in, z.B. Millisekunden, messen und diese in Relation zur Eingabegröße setzen. Dies würde das Laufzeitverhalten allerdings nur unzureichend beschreiben. Zum Einen wären unsere gemachten Messungen natürlich abhängig von der verwendeten Hardware, zum Anderen kann man keine Aussage darüber treffen, wie schnell der Algorithmus bei Verwendung besserer Hardware arbeiten würde, bzw. wie groß die Daten sein dürften, mit denen wir den besseren Rechner füttern dürften, um in vergleichbarer Zeit zum schlechteren Computer und einer geringeren Datenmenge ein Ergebnis zu bekommen. Mehr Aufschluss über das allgemeine Laufzeitverhalten des Algorithmus bekommen wir dagegen schon, wenn wir die Laufzeitmesspunkte für verschiedene Eingabegrößen in ein Diagramm eintragen, siehe Abbilung 5.13. Hier wurden Laufzeitmesspunkte für den $O(n^5)$-Algorithmus zur Berechnung einer knickminimalen Beschriftung aus Abschnitt 3.3.1 aufgetragen. Was wir damit gemacht haben, ist bereits ein erster Schritt hin zur asymptotischen Laufzeitanalyse des Algorithmus: Wir versuchen, die Laufzeitkurve zu interpolieren um Rückschlüsse auf das Verhalten des Algorithmus für größere Eingabegrößen ziehen zu können.

Dies lässt sich natürlich auch mathematisch präzisieren. Die Eingabegröße bezeichnet man üblicherweise mit dem Parameter n. Wenn die Aufgabe z.B. darin besteht, ein Feld mit Zahlen zu sortieren, dann bezeichnet n die Anzahl der zu sortierenden Zahlen. Als theoretische Laufzeit geben wir nun die Anzahl der Berechnungsschritte in Abhängigkeit von n an, die nötig sind, um das Problem zu lösen. Ein erstes Ziel ist somit schon erreicht: Unsere Bewertung ist unabhängig von der verwendeten Hardware geworden. Nachdem wir die Anzahl der Berechnungsschritte angeben können, spielt für die tatsächliche Laufzeit auf einem bestimmten Computer (im idealisierten Fall) also nur noch die Zeit eine Rolle, die dieser Computer benötigt, um einen Berechnungsschritt auszuführen. Mathematisch verbleibt folgendes Problem zu lösen: Wie drücke ich aus, dass zwei Algorithmen ungefähr dieselbe Laufzeit haben? Nun, indem ich die Laufzeit einfach als Funktion in n angebe. Dazu folgendes Beispiel:

Wir können obiges Sortierproblem mit folgendem naiven Algorithmus lösen: Wir gehen das Feld der n Zahlen in einem ersten Durchlauf durch, um die kleinste Zahl zu finden. Hierfür brauchen wir $n-1$ Vergleichsoperationen. Danach bleiben $n-1$ Zahlen übrig, aus denen wir mit dem gleichen Verfahren wiederum die kleinste Zahl bestimmen. Alles in allem brauchen wir also

$$(n-1)+(n-2)+(n-3)+\cdots+2+1=\frac{1}{2}(n-1)n=\frac{1}{2}n^2-\frac{1}{2}n$$

Vergleichsoperationen um das Feld zu sortieren. Um es kurz zu machen, in Groß-Oh-Notation ist die Laufzeit unseres Algorithmus damit $O(n^2)$. Die entscheidende Rolle spielt dabei lediglich der Term, der die Laufzeit festlegt, falls die Eingabegröße n unendlich groß würde, welcher in unserem Fall $\frac{1}{2}n^2$ ist. Der Term $\frac{1}{2}n$ ist vernachlässigbar, da er für große Werte von n von $\frac{1}{2}n^2$ dominiert wird. Unsere Laufzeitfunktion nähert sich also, *asymptotisch*, für große n der Kurve $\frac{1}{2}n^2$ und kann schließlich nicht mehr von ihr unterschieden werden. Mathematisch

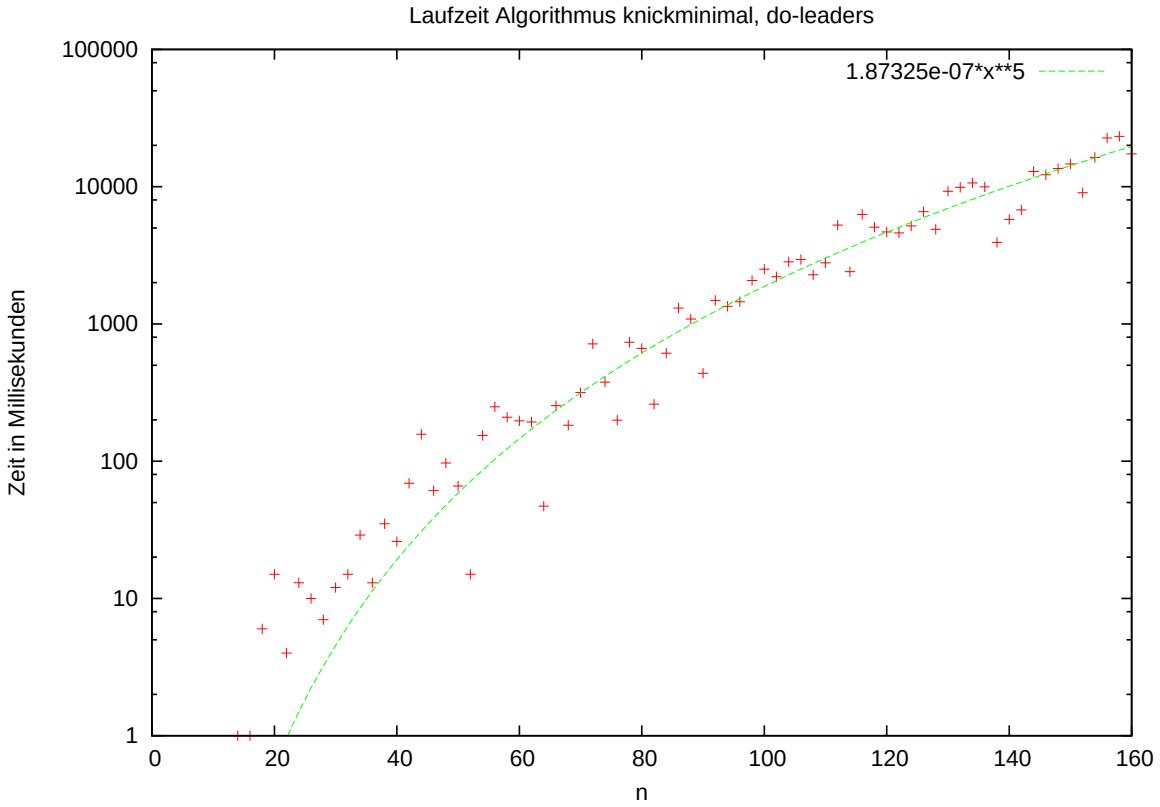


Abbildung 5.13: Die Laufzeitmesspunkte des $O(n^5)$-Algorithmus aus Abschnitt 3.3.1 und eine eingepasste Kurve von Grad 5, beides logarithmisch skaliert.

ausgedrückt heißt das: $\lim_{n\to\infty} \frac{(\frac{1}{2}n^2-\frac{1}{2}n)}{(\frac{1}{2}n^2)} = 1$. Konstante Faktoren, die nicht von n abhängen, werden in der Oh-Notation verschluckt, in unserem Fall gilt dies für $\frac{1}{2}$. Der Grund hierfür ist, dass konstante Faktoren mathematisch irrelevant sind, wenn man *verschiedene* von n abhängige Funktionen auf ihr Verhalten für n gegen ∞ untersucht. Dies wird am einfachsten wieder durch ein Beispiel illustriert: Vergleicht man einen Algorithmus, der $100n$ Berechnungsschritte benötigt mit einem Algorithmus, der $\frac{1}{2}n^2$ Berechnungsschritte benötigt, so sieht der Vergleich in der Oh-Notation also $O(n)$ zu $O(n^2)$ aus. Der erste Algorithmus ist also asymptotisch schneller. Zwar benötigt der erste Algorithmus für Eingabelängen kleiner als 200 mehr Berechnungsschritte als der zweite, für alle Eingabelängen über 200 schlägt er den zweiten Algorithmus aber um Längen. *Asymptotisch schneller* lässt sich also wie folgt rechtfertigen: Für genügend große Eingabelängen führt der asymptotisch schnellere Algorithmus immer weniger Berechnungsschritte durch als der asymptotisch langsamere. Ein optimaler Algorithmus für das Sortierproblem hat übrigens eine asymptotische Laufzeit von $O(n \log n)$.

Für den nächsten Abschnitt ist die folgende Definition von Relevanz:

Definition 5.4 *(Polynomialzeit) Falls die asymptotische Laufzeit eines Algorithmus höchstens $O(n^k)$ ist, wobei k eine von n unabhängige, natürliche Zahl ist, so heißt der Algorithmus* polynomiell *in der Eingegabegröße n.*

5.9.2 Welche Bedeutung haben die Komplexitätsklassen $\mathcal{P}$ und $\mathcal{NP}$?

$\mathcal{P}$ ist die Klasse der Probleme, die in Polynomialzeit von einer deterministischen Turingmaschine[1] zu lösen sind. $\mathcal{NP}$ ist die Klasse der Probleme, die in Polynomialzeit von einer nicht-deterministischen Turingmaschine zu lösen sind.

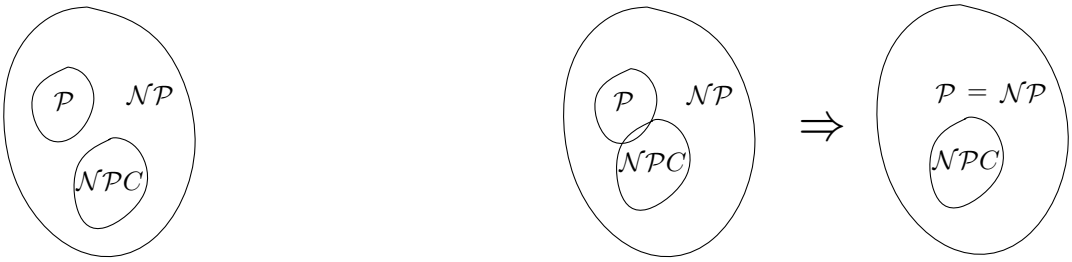
Punkt. Soweit zu den Definitionen der Komplexitätsklassen $\mathcal{P}$ und $\mathcal{NP}$. Würde ich, wie von meinem Betreuer befürchtet, nun tatsächlich anfangen zu erklären, was es denn nun genau mit diesen Definitionen auf sich hat, so wäre diese Dissertation vermutlich nochmal doppelt so lange geworden wie sie ohnehin schon ist. Deswegen möchte ich darauf verzichten und stattdessen auf die Bedeutung dieser Klassen und grob darauf eingehen, was man sich unter ihnen vorstellen kann.

Die Klasse $\mathcal{P}$ ist (noch) relativ einfach erklärt, in ihr liegen alle Probleme für die es einen Algorithmus gibt, der das Problem in Polynomialzeit löst. Der Informatiker spricht von *effizient* zu lösenden Problemen und betrachtet diese Probleme als einfache Probleme. In der Klasse $\mathcal{NP}$ liegen potentiell mehr und auch schwerere Probleme als in $\mathcal{P}$. Wobei wir hier allerdings schon bei den Knackpunkten angekommen sind, dem 'potentiell mehr' und 'schwerere'. Definitionsgemäß liegen alle Probleme, die in $\mathcal{P}$ liegen auch in $\mathcal{NP}$. Als ein schweres Problem betrachtet der Informatiker ein Problem, das sich nicht effizient lösen lässt. Aber bis heute konnte der Nachweis, dass ein solch schweres Problem existiert, das nicht in $\mathcal{P}$ liegt und das somit $\mathcal{NP}$ zu einer echten Oberklasse von $\mathcal{P}$ machen würde, nicht geführt werden. Allerdings wird die Annahme $\mathcal{P} \neq \mathcal{NP}$ unter den Informatikern als vermutlich gültig akzeptiert. Die Korrektheit dieser Annahme aber zu beweisen (oder doch zu widerlegen) bildet das derzeit wohl größte offene Problem der theoretischen Informatik.

Nun ein paar erläuternde Worte zu $\mathcal{NP}$. Die Klasse $\mathcal{NP}$ umfasst alle Probleme, um genau zu sein, alle *Entscheidungs*probleme, für die sich zumindest in Polynomialzeit verifizieren lässt, ob eine vorgeschlagene Lösungskonfiguration das Problem tatsächlich löst oder nicht. Damit ist auch sofort klar, dass $\mathcal{P} \subset \mathcal{NP}$ gilt, denn wenn ich ein Problem optimal in Polynomialzeit lösen kann, dann kann ich durch Vergleichen mit der Lösung natürlich für jede vorgeschlagene andere Lösungskonfiguration in Polynomialzeit entscheiden, ob sie das Problem ebenfalls löst. Allerdings bedeutet die obige Definition der Klasse $\mathcal{NP}$ nicht, dass $\mathcal{NP}$-Probleme überhaupt nicht zu lösen sind. Man geht jedoch davon aus, dass es $\mathcal{NP}$-Probleme gibt, die sich nicht effizient lösen lassen, also nicht in Polynomialzeit. Könnte man für ein Problem, welches nachgewiesenermaßen in $\mathcal{NP}$ liegt, also zeigen, dass es sich nicht effizient lösen lässt, so hätte man $\mathcal{P} \neq \mathcal{NP}$ bewiesen. Therotisch gibt es in der Klasse $\mathcal{NP}$ nochmal eine ausgewiesene Teilklasse besonders schwerer Probleme, die $\mathcal{NP}$-vollständigen Probleme (Klassenbezeichnung $\mathcal{NPC}$). Diese Probleme sind so schwer, dass ein einziger polynomialer Lösungsalgorithmus eines $\mathcal{NPC}$-Problems, Lösungsalgorithmen für alle $\mathcal{NP}$-Probleme induzieren würde. Man hätte dann $\mathcal{P} = \mathcal{NP}$ gezeigt. Abbildung 5.14 illustriert die angesprochenen Sachverhalte in einem Mengendiagramm.

Als Beispiel eines $\mathcal{NPC}$-Problems, das also vermutlich in $\mathcal{NP} \setminus \mathcal{P}$ liegt, möchte ich das *Pfad-Dilations-Problem* angeben: Für n gegebene Punkte in der Ebene soll derjenige Pfad bestimmt werden, der alle Punkte geradlinig verbindet und der als Graph die kleinste Dilation hat (Umweg des Weges zwischen zwei Punkten im Graphen verglichen mit deren Luftliniendistanz, siehe Abschnitt 0.1.1). In Abbildung 5.15 ist der optimale Dilationspfad und ein Pfad sehr schlechter, großer Dilation für eine Beispielpunktemenge angegeben.

[1] Eine ausführlichere Beschreibung ist etwa Wikipedia zu entnehmen, siehe http://de.wikipedia.org/wiki/Turingmaschine



(a) Die vermutliche Wahrheit.

(b) Die Auswirkung eines $\mathcal{NP}$-vollständigen Problems in $\mathcal{P}$.

Abbildung 5.14: Enthalten-Sein-Relationen der verschiedenen Komplexitätsklassen.

(a) Der optimale Pfad

(b) Ein schlechter Pfad.

Abbildung 5.15: Das *Pfad-Dilations-Problem.*

Das Pfad-Dilations-Problem wurde jüngst von Giannopoulos, Knauer und Marx [GKM07] als $\mathcal{NP}C$-Problem erkannt. Es lässt sich wie folgt korrekt als Entscheidungsproblem formulieren: Gegebenen n Punkte in der Ebene und ein Parameter $t > 1$, gibt es einen Pfad, der die n Punkte verbindet und der eine Dilation von höchstens t hat?

Die Zugehörigkeit zur Klasse $\mathcal{NP}$ ist offensichtlich: Wenn ich einen Pfad als Lösungsvorschlag gegeben habe, dann kann ich für jedes der $O(n^2)$ vielen Punktepaare mit Hilfe der Distanz im Graphen deren Dilation in Polynomialzeit berechnen und dann einfach entscheiden, ob die maximale Dilation, die aufgetreten ist, kleiner als t war. Falls dies zutrifft, ist der vorgeschlagene Pfad ein Pfad, der das Problem löst.

Offensichtlich ist das Problem allerdings auch nicht unlösbar: Ich kann jeden möglichen Pfad betrachten, dessen Dilation berechnen und bekomme die Lösung dann einfach als den Pfad, für den ich die kleinste Dilation gefunden habe. Dieses Vorgehen ist aus Informatiksicht gesehen aber ineffizient, da es $n!/2$ kombinatorisch verschiedene Möglichkeiten gibt, n Punkte durch einen Pfad zu verbinden. In der Tat habe ich mich während meiner Doktorandenzeit etwas eingehender mit dem Pfad-Dilations-Problem beschäftigt und auch einen solchen faktoriellen Lösungsalgorithmus implementiert. Die Ineffizienz dieses Algorithmus konnte ich dann daran ablesen, dass er zur Berechnung von Instanzen mit 8 Punkten bereits mehrere Sekunden gebraucht hat, und dass bei Instanzen mit mehr als 8 Punkten ein Memory-Overflow aufgetreten ist.

Sollte $\mathcal{NP} \neq \mathcal{P}$ wahr sein, würde das für das Pfad-Dilations-Problem bedeuten, dass kein polynomialer Lösungsalgorithmus existiert. Gelänge es Ihnen aber, lieber Leser, einen polynomialen Lösungsalgorithmus zu entwickeln, so hätten Sie damit gleichzeitig $\mathcal{NP} = \mathcal{P}$ gezeigt und ein weiteres Leben als ruhmreicher Informatiker wäre Ihnen gewiss.

5.9.3 Was ist ein Approximationsalgorithmus?

Approximationsalgorithmen werden üblicherweise für $\mathcal{NP}$-vollständige Probleme verwendet. Falls die Klassen $\mathcal{P}$ und $\mathcal{NP}$ tatsächlich verschieden sind, wovon man ausgeht, rechtfertigt das Wissen, dass ein $\mathcal{NP}$-vollständiges Problem keinen polynomialen Lösungsalgorithmus zulässt, ein approximatives Vorgehen. Ein *Approximationsalgorithmus* sollte daher aber eine effiziente Laufzeit haben und man sollte die Bereitschaft mitbringen, sich mit einer Lösung zufrieden zu geben, die (geringfügig) schlechter als eine optimale sein kann. Dass Vergleichen von Lösungen impliziert natürlich sofort das Vorhandensein einer Funktion, die die Güte der Lösung bewertet. Approximationsalgorithmen exisitieren also nur für solche Art von Problemen, bei denen also eine optimale Lösung *approximiert* wird. In Kapitel 1 ist diese Gütefunktion etwa die Länge des resultierenden Netzwerks, welche es in diesem Fall zu minimieren gilt. Die Gütegarantie des Approximationsalgorithmus gibt man durch seinen *Approximationsfaktor* an: Für die Manhattan-Netzwerke hat unser Algorithmus einen Approximationsfaktor von 3, was nichts anderes bedeutet, als dass wir garantieren können, dass das Netzwerk, das unser Algorithmus ausgibt, im schlechtesten Fall höchstens dreimal so lang ist wie ein minimales Manhattan-Netzwerk.

Wie bereits erwähnt, sind Approximationsalgorithmen hauptsächlich interessant, wenn sie polynomiale Laufzeit haben und ein Problem approximieren, das sich vermutlich nicht optimal in Polynomialzeit lösen lässt. In Spezialfällen können allerdings auch nicht-polynomielle Approximationsalgorithmen oder auch Approximationsalgorithmen für Probleme, die polynomiell gelöst werden können, von Interesse sein. Wenn sich z.B. ein Problem nur in kubischer ($O(n^3)$) Zeit optimal lösen lässt, so ist ein Faktor-1.5 Approximationsalgorithmus in linearer

Zeit ($O(n)$) durchaus algorithmisch von Interesse.

Interesting Facts

Chapter 2 - Interference-minimal Networks

There was a funny error in the conference version of this work. For the fuzzy model we adjusted the transmission power of the sending stations in an insufficient way. By chance they may have never reached any other station simply because the signal gets lost somewhere in the fuzzy region *before* arriving at the terminal station. This insufficieny was surprisingly never noticed by any reviewer or any auditor during the talks (in fact from Herman himself), but was, of course, fixed for the journal version.

In früheren Versionen dieser Arbeit befand sich ein lustiger Fehler: Für das Schätzmodell hatten wir die Sendestärke der Stationen so festgelegt, dass gesendete Signale möglicherweise nie an den vorgesehenen Empfangsstationen ankommen würden, weil sie schon vorher in der *Fuzzy*-Region verloren gehen würden. Glücklicherweise wurde diese kleine Unzulänglichkeit von keinem Gutachter oder Zuhörer eines Vortrags (außer Herman selbst) je bemerkt und in der endgültigen Version dann natürlich auch bereinigt.

Chapter 3 - Boundary Labeling

I did significant parts of the research for this work together with Mira Lee, Herman Haverkort and Martin Nöllenburg during the Fifth Korean Workshop on Computational Geometry. After we had unsuccessfully been trying greedy strategies for two days, Herman came to us in the morning of the third day and stated „I just had an idea ... in the shower! Why don't we apply dynamic programming?" And this has indeed well worked out. It seems like you could have the best ideas while showering.

Bedeutende Teile der Forschung an dieser Arbeit habe ich zusammen mit Mira Lee, Herman Haverkort und Martin Nöllenburg während eines Workshops gemacht. Nachdem wir zwei Tage lang vergeblich versucht hatten, Greedy-Strategien anzuwenden, kam Herman am morgen des dritten Tags beim Frühstück mit den Worten: „Hey, ich hatte gerade eine Idee ... ... unter der Dusche! Warum probieren wir es nicht mit dynamischen Programmieren?" auf uns zu. Und das hat in der Tat funktioniert. Anscheinend kann man die besten Ideen also auch während dem Duschen haben.

Chapter 4 - Detecting Flocks

Apart from the caribous this work has another, more politically incorrect motivation that stems from surveillance policy, for example tracing peoples by the signals of their mobiles. And indeed I heard that there was a request from the Australian Ministry of Defence to look at these kind of problems. This of course yielded perfect stuff for my class reunion: „I just come from Australia ... I worked there on behalf of the Australian Ministry of Defence." Sounds good!

Neben den Karibus hat diese Arbeit auch noch eine weitere Anwendung, die politisch eher bedenklich ist: Das Überwachen von Personen, z.B. anhand der Signale ihrer Handys. Und tatsächlich habe ich gehört, dass es eine dementsprechende Anfrage des Australischen Verteidigungsministeriums für diese Art von Probleme gab. Was natürlich perfekten Stoff für

mein Klassentreffen geliefert hätte, sofern ich hingegangen wäre: „Ja, ich komme gerade aus Australien, war dort unterwegs im Auftrag des Australischen Verteidigungsministerium."

Chapter 5 - Packing Discs

The result of this work is naturally only of theoretic interest as an algorithm with running time worse than $O(n^{324})$ can never be of any practical use. For this reason the first working title of the paper had been „A *slow* approximation algorithm". Well, most of the co-authors liked this title but eventually it broke down by the veto of one co-author and became „A *polynomial-time* approximation algorithm".

Das Resultat dieser Arbeit ist natürlich rein theoretischer Natur, da ein Algorithmus, dessen Laufzeit noch schlimmer als $O(n^{324})$ ist nie irgendeine Anwendung in der Praxis finden würde. Deshalb lautete der erste Arbeitstitel des Papers „A *slow* approximation algorithm". Im Prinzip hat dieser Titel (fast) allen Autoren gefallen, ist letztendlich dann aber doch am Veto eines Ko-Autors gescheitert und politisch korrekt in „A *polynomial-time* approximation algorithm" umbenannt worden.

List of publications

[1] Alexander Wolff, Marc Benkert, and Takeshi Shirabe. The minimum Manhattan network problem: Approximations and exact solutions. In *Proc. 20th European Workshop on Computational Geometry (EWCG'04)*, pages 209–212, Sevilla, 24–26 March 2004.

[2] Marc Benkert, Alexander Wolff, Florian Widmann, and Takeshi Shirabe. The minimum Manhattan network problem: Approximations and exact solutions. *Computational Geometry: Theory and Applications*, 35(3):188–208, 2006.

[3] Marc Benkert, Joachim Gudmundsson, Herman Haverkort, and Alexander Wolff. Constructing interference-minimal networks. In Jiří Wiedermann, Julius Stuller, Gerard Tel, Jaroslav Pokorný, and Mária Bieliková, editors, *Proc. 32nd Int. Conf. on Current Trends in Theory and Practice of Computer Science (SOFSEM'06)*, volume 3831 of *Lecture Notes in Computer Science*, pages 166–176. Springer-Verlag, 2006.

[4] Marc Benkert and Martin Nöllenburg. Improved algorithms for length-minimal one-sided boundary labeling. In *Proc. 23rd European Workshop on Computational Geometry (EWCG'07)*, pages 190–193, Graz, 19–21 March 2007.

[5] Marc Benkert, Herman Haverkort, Moritz Kroll, and Martin Nöllenburg. Algorithms for multi-criteria one-sided boundary labeling. In *Proc. 15th Int. Symp. Graph Drawing (GD '07)*, 2007. To appear.

[6] Marc Benkert, Joachim Gudmundsson, Florian Hübner, and Thomas Wolle. Reporting flock patterns. In Yossi Azar and Thomas Erlebach, editors, *Proc. 14th Annu. Europ. Symp. on Algorithms (ESA'06)*, volume 4168 of *Lecture Notes in Computer Science*, pages 660–671. Springer-Verlag, 2006.

[7] Marc Benkert, Joachim Gudmundsson, Christian Knauer, Esther Moet, René van Oostrum, and Alexander Wolff. A polynomial-time approximation algorithm for a geometric dispersion problem. In Danny Z. Chen and Der-Tsai Lee, editors, *Proc. 12th Annu. Int. Comput. Combinatorics Conf. (COCOON'06)*, volume 4112 of *Lecture Notes in Computer Science*, pages 166–175. Springer-Verlag, 2006.

[8] Michael Baur and Marc Benkert. Network comparison. In Ulrik Brandes and Thomas Erlebach, editors, *Network Analysis*, volume 3418 of *Lecture Notes in Computer Science Tutorial*, chapter 12, pages 318–340. Springer-Verlag, 2005.

[9] Marc Benkert, Joachim Gudmundsson, Christian Knauer, René van Oostrum, and Alexander Wolff. A polynomial-time approximation algorithm for a geometric dispersion problem. *International Journal of Computational Geometry and Applications*, 2007. To appear.

[10] Marc Benkert, Joachim Gudmundsson, Herman Haverkort, and Alexander Wolff. Constructing interference-minimal networks. *Computational Geometry: Theory and Applications*, 2007. To appear.

[11] Iris Reinbacher, Marc Benkert, Marc van Kreveld, Joseph S.B. Mitchell, Jack Snoeyink, and Alexander Wolff. Delineating boundaries for imprecise regions. *Algorithmica*, 2007. To appear.

[12] Damian Merrick, Martin Nöllenburg, Alexander Wolff, and Marc Benkert. Morphing polygonal lines: A step towards continuous generalization. In *Proc.23rd European Workshop on Computational Geometry (EWCG'07)*, pages 6–9, Graz, 2007.

[13] Marc Benkert, Martin Nöllenburg, Takeaki Uno, and Alexander Wolff. Minimizing intra-edge crossings in wiring diagrams and public transport maps. In Michael Kaufmann and Dorothea Wagner, editors, *Proc. 14th Int. Symposium on Graph Drawing (GD'06)*, volume 4372 of *Lecture Notes in Computer Science*, pages 270–281. Springer-Verlag, 2007.

[14] Iris Reinbacher, Marc van Kreveld, Tim Adelaar, and Marc Benkert. Scale dependent definitions of gradient and aspect and their computation. In Andreas Riedl, Wolfgang Kainz, and Gregory A. Elmes, editors, *Proc. 12th Intern. Symp. Spatial Data Handling (SDH'06)*, pages 863–879, 2006.

[15] Marc Benkert, Joachim Gudmundsson, Christian Knauer, Esther Moet, René van Oostrum, and Alexander Wolff. A polynomial-time approximation algorithm for a geometric dispersion problem. In *Proc. 22st European Workshop on Computational Geometry (EWCG'06)*, Delphi, 27–29 March 2006.

[16] Iris Reinbacher, Marc Benkert, Marc van Kreveld, Joseph S.B. Mitchell, and Alexander Wolff. Delineating boundaries for imprecise regions. In Gerth Stølting Brodal and Stefano Leonardi, editors, *Proc. 13th Annu. Europ. Symp. on Algorithms (ESA'05)*, volume 3669 of *Lecture Notes in Computer Science*, pages 143–154. Springer-Verlag, 2005.

[17] Iris Reinbacher, Marc Benkert, Marc van Kreveld, and Alexander Wolff. Delineating boundaries for imprecise regions. In *Proc. 21st European Workshop on Computational Geometry (EWCG'05)*, pages 127–130, Eindhoven, 9–11 March 2005.

[18] Marc Benkert, Florian Widmann, and Alexander Wolff. The minimum Manhattan network problem: A fast factor-3 approximation. In Jin Akiyama, Mikio Kano, and Xuehou Tan, editors, *Proc. 8th Japanese Conf. on Discrete and Computational Geometry (JCDCG'04)*, volume 3742 of *Lecture Notes in Computer Science*, pages 16–28. Springer-Verlag, 2005.

[19] Marc Benkert, Joachim Gudmundsson, Herman Haverkort, and Alexander Wolff. Constructing interference-minimal networks. In *Proc. 21st European Workshop on Computational Geometry (EWCG'05)*, pages 203–206, Eindhoven, 9–11 March 2005.

[20] Michael Baur, Marc Benkert, Ulrik Brandes, Sabine Cornelsen, Marco Gaertler, Boris Köpf, Jürgen Lerner, and Dorothea Wagner. visone - software for visual social network

analysis. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Proceedings of the 9th International Symposium on Graph Drawing (GD'01)*, volume 2265 of *Lecture Notes in Computer Science*, pages 463–464, 2002.

Bibliography

[ADM+95] Sunil Arya, Gautam Das, David M. Mount, Jeffrey S. Salowe, and Michiel Smid. Euclidean spanners: Short, thin, and lanky. In *Proc. 27th Annu. ACM Sympos. Theory Comput. (STOC'95)*, pages 489–498, 1995. 11

[AE84] Franz Aurenhammer and Herbert Edelsbrunner. An optimal algorithm for constructing the weighted Voronoi diagram in the plane. *Pattern Recogn.*, 17:251–257, 1984. 122

[AFK+92] Helmut Alt, Rudolf Fleischer, Michael Kaufmann, Kurt Mehlhorn, Stefan Näher, Stefan Schirra, and Christian Uhrig. Approximate motion planning and the complexity of the boundary of the union of simple geometric figures. *Algorithmica*, 8:391–406, 1992. 106

[Aga97] Pankaj K. Agarwal. Range searching. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 31, pages 575–598. CRC Press LLC, Boca Raton, FL, 1997. 11

[AGM+90] Stephen F. Altschul, Warran Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990. 21

[AHP05] Boris Aronov and Sariel Har-Peled. On approximating the depth and related problems. In *Proc.of 16th ACM-SIAM Symposium on Discrete Algorithms (SODA'05)*, pages 886–894, 2005. 100

[AHS05] Kamran Ali, Knut Hartmann, and Thomas Strothotte. Label layout for interactive 3D illustrations. *J. WSCG*, 13:1–8, 2005. 70

[AKS95] Pankaj K. Agarwal, Matthew Katz, and Micha Sharir. Computing depth orders for fat objects and related problems. *Comput. Geom. Theory Appl.*, 5:187–206, 1995. 106

[AM00] Sunil Arya and David M. Mount. Approximate range searching. *Computational Geometry: Theory and Applications*, 17:135–152, 2000. 60, 67

[ANCG07] Ghazi Al-Naymat, Sanjay Chawla, and Joachim Gudmundsson. Dimensionality reduction for long duration and complex spatio-temporal queries. In *Proc. of the 22nd ACM Symposium on Applied Computing*, pages 393–397. ACM, 2007. 118

[Aur88] Franz Aurenhammer. Improved algorithms for discs and balls using power diagrams. *J. Algorithms*, 9:151–161, 1988. 122, 126

[BET99] Marshall Bern, David Eppstein, and Shang-Hua Teng. Parallel construction of quadtrees and quality triangulations. *Internat. J. Comput. Geom. Appl.*, 9(6):517–532, 1999. 101

[BF01] Christoph Baur and Sándor P. Fekete. Approximation of geometric dispersion problems. *Algorithmica*, 30(3):451–470, 2001. 17, 121, 136

[BKPS06a] Michael A. Bekos, Michael Kaufmann, Katerina Potika, and Antonios Symvonis. Multi-stack boundary labeling problems. In S. Arun-Kumar and N. Garg, editors, *Proc. Foundations of Software Technology and Theoretical Computer Science (FSTTCS2006)*, volume 4337 of *Lecture Notes in Computer Science*, pages 81–92. Springer-Verlag, 2006. 71

[BKPS06b] Michael A. Bekos, Michael Kaufmann, Katerina Potika, and Antonios Symvonis. Polygon labelling of minimum leader length. In K. Misue, K. Sugiyama, and J. Tanaka, editors, *Proc. Asia Pacific Symposium on Information Visualisation (APVIS2006)*, volume 60 of *CRPIT*, pages 15–21, 2006. 71

[BKSW07] Michael A. Bekos, Michael Kaufmann, Antonios Symvonis, and Alexander Wolff. Boundary labeling: Models and efficient algorithms for rectangular maps. *Computational Geometry: Theory & Applications*, 36:215–236, 2007. 16, 70, 71, 72, 76, 77, 80, 86, 92, 93

[BvRWZ04] Martin Burkhart, Pascal von Rickenbach, Roger Wattenhofer, and Aaron Zollinger. Does topology control reduce interference? In *Proc. 5th ACM Int. Symp. Mobile Ad Hoc Networking and Computing (MobiHoc'04)*, pages 9–19, 2004. 15, 50, 52, 56, 66

[Cap99] Paola Cappanera. A survey on obnoxious facility location problems. Technical Report TR-99-11, University of Pisa, 1999. 120

[CCK+05] Ee-Chien Chang, Sung Woo Choi, DoYong Kwon, Hyungju Park, and Chee-K. Yap. Shortest path amidst disc obstacles is computable. In *Proc. 21th ACM Symposium on Computational Geometry*, pages 116–125, 2005. 127

[Cha00] Timothy M. Chan. Random sampling, halfspace range reporting, and construction of ($\leqslant k$)-levels in three dimensions. *SIAM Journal on Computing*, 30(2):561–575, 2000. 54

[Che89] L. Paul Chew. There are planar graphs almost as good as the complete graph. *J. Comput. Syst. Sci.*, 39:205–219, 1989. 11

[CK95] Paul B. Callahan and S. Rao Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *Journal of the ACM*, 42(1):67–90, January 1995. 57, 58, 64

[CNV05] Victor Chepoi, Karim Nouioua, and Yann Vaxés. A rounding algorithm for approximating minimum Manhattan networks. In Chandra Chekuri, Klaus Jansen,

José D. P. Rolim, and Luca Trevisan, editors, *Proc. 8th Intern. Workshop Approx. Algorithms for Combinatorial Optimization Problems (APPROX'05)*, volume 3624 of *Lecture Notes in Computer Science*, pages 40–51. Springer-Verlag, 2005. 14, 22, 47, 48, 138

[Das03] Dash Optimization Inc. *Xpress-Optimizer Reference Manual*. Warwickshire, U.K., 2003. 42

[dBvKOS00] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications, second edition*. Springer-Verlag, 2000. 12

[Dij59] Edsger W. Dijkstra. A note on two problems in connection with graphs. *Numerische Math.*, 1:269–271, 1959. 56

[DN97] Gautam Das and Giri Narasimhan. A fast algorithm for constructing sparse Euclidean spanners. *International Journal of Computational Geometry and Applications*, 7:297–315, 1997. 57

[EGS05] David Eppstein, Michael T. Goodrich, and Jonathan Z. Sun. The skip quadtree: a simple dynamic data structure for multidimensional data. In *Proc.of the 21st ACM Symposium on Computational Geometry (SoCG'05)*, pages 296–305, 2005. 101, 102

[Epp00] David Eppstein. Spanning trees and spanners. In Jörg-Rüdiger Sack and Jorge Urrutia, editors, *Handbook of Computational Geometry*, pages 425–461. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 2000. 9, 11

[FGG05] Mohammad Farshi, Panos Giannopoulos, and Joachim Gudmundsson. Finding the best shortcut in a geometric network. In *Proc. 21st Symposium on Computational Geometry (SoCG'05)*, Journal of the ACM, pages 327–335, 2005. 9

[FP99] Jean-Daniel Fekete and Catherine Plaisant. Excentric labeling: Dynamic neighborhood labeling for data visualization. In *[CHI]*, pages 512–519, 1999. 70

[FPT81] Robert Fowler, Mike Paterson, and Steven Tanimoto. Optimal packing and covering in the plane are NP-complete. *Information Processing Letters*, 12:133–137, 1981. 17, 121

[FRC01] A.U. Frank, J.F. Raper, and J.-P. Cheylan, editors. *Life and motion of spatial socio-economic units*. Taylor & Francis, London, 2001. 99

[GHvK02] Joachim Gudmundsson, Mikael Hammer, and Marc van Kreveld. Higher order Delaunay triangulations. *Computational Geometry – Theory & Applications*, 23(1):85–98, 2002. 54

[GKM07] Panos Giannopoulos, Christian Knauer, and Daniel Marx. Minimum-dilation tour is NP-hard (extended abstract). In *Proc. of the 23rd European Workshop on Computational Geometry (EWCG)*, Graz, Austria, 2007. 146

[GLN01] Joachim Gudmundsson, Christos Levcopoulos, and Giri Narasimhan. Approximating a minimum Manhattan network. *Nordic Journal of Computing*, 8:219–232, 2001. 21, 22, 23, 34, 43, 45

[GLN02] Joachim Gudmundsson, Christos Levcopoulos, and Giri Narasimhan. Improved greedy algorithms for constructing sparse geometric spanners. *SIAM Journal on Computing*, 31(5):1479–1500, 2002. 56

[GNS03] Joachim Gudmundsson, Giri Narasimhan, and Michiel Smid. Distance-preserving approximations of polygonal paths. In *Proc. 23rd Conf. Foundations Software Technology Theoretical Comput. Sci. (FSTTCS'03)*, pages 217–228, 2003. 64

[Gra18] Henry Gray. *Anatomy of the Human Body.* Lea & Febiger, Philadelphia, 1918. 70

[GvK06] Joachim Gudmundsson and Marc van Kreveld. Computing longest duration flocks in trajectory data. In *Proc. 14th ACM Symposium on Advances in GIS*, pages 35–42, 2006. 100

[GvKS04] Joachim Gudmundsson, Marc van Kreveld, and Bettina Speckmann. Efficient detection of motion patterns in spatio-temporal data sets. In I.F. Cruz and D. Pfoser, editors, *Proc.of the 12th Int. Symposium of ACM Geographic Information Systems*, pages 250–257, Washington DC, USA, 2004. 100, 101

[HdBG04] Herman Haverkort, Mark de Berg, and Joachim Gudmundsson. Box-trees for collision checking in industrial installations. *Computational Geometry Theory and Applications*, 28(2–3):113–135, 2004. 60

[HM85] Dorit S. Hochbaum and Wolfgang Maas. Approximation schemes for covering and packing problems in image processing and VLSI. *Journal of the ACM*, 32:130–136, 1985. 17, 121, 123, 124, 127, 139

[IS02] Sachiko Iwase and Hideo Saito. Tracking soccer player using multiple views. In *Proc.of the IAPR Workshop on Machine Vision Applications (MVA02)*, pages 102–105, 2002. 99

[JT92] Jerzy W. Jaromczyk and Godfried T. Toussaint. Relative neighborhood graphs and their relatives. *Proc. IEEE*, 80(9):1502–1517, 1992. 52

[Kei88] J. Mark Keil. Approximating the complete Euclidean graph. In R. Karlsson and A. Lingas, editors, *Proc. First Scandinavian Workshop on Algorithm Theory (SWAT'88)*, volume 318 of *Lecture Notes in Computer Science*, pages 208–213, Halmstad, Sweden, 5–8 July 1988. Springer-Verlag. 11

[KIA02] Ryo Kato, Keiko Imai, and Takao Asano. An improved algorithm for the minimum Manhattan network problem. In Prosenjit Bose and Pat Morin, editors, *Proc. 13th Annual International Symposium on Algorithms and Computation (ISAAC'02)*, volume 2518 of *Lecture Notes in Computer Science*, pages 344–356. Springer-Verlag, 2002. 14, 22, 23, 24

[KMM93] Rolf Klein, Kurt Mehlhorn, and Stefan Meiser. Randomized incremental construction of abstract Voronoi diagrams. *Computational Geometry: Theory and Applications*, 3:157–184, 1993. 122

[Kru56] Joseph B. Kruskal. On the shortest spanning subtree of a graph and the travelling salesman problem. *Proc. American Mathematical Society*, 7:48–50, 1956. 63

[KSB01] George Kollios, Stan Sclaroff, and Margit Betke. Motion mining: discovering spatio-temporal patterns in databases of human motion. In *Proc.of the ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, 2001. 99

[LAP03] Fumei Lam, Marina Alexandersson, and Lior Pachter. Picking alignments from (Steiner) trees. *Journal of Computational Biology*, 10:509–520, 2003. 14, 21

[Lee82] Der-Tsai Lee. On k-nearest neighbor Voronoi diagrams in the plane. *IEEE Trans. Comput.*, C-31:478–487, 1982. 54

[LI02] Patrick Laube and Stephan Imfeld. Analyzing relative motion within groups of trackable moving point objects. In M. J. Egenhofer and D. M. Mark, editors, *Proc. Geographic Information Science 2002 (GIS'02)*, volume 2478 of *Lecture Notes in Computer Science*, pages 132–144. Springer-Verlag, 2002. 99

[LvKI04] Patrick Laube, Marc van Kreveld, and Stephan Imfeld. Finding REMO – detecting relative motion patterns in geospatial lifelines. In P. F. Fisher, editor, *Proc.of the 11th Int. Symposium on Spatial Data Handling (SDH'04)*, pages 201–214, Berlin, 2004. Springer-Verlag. 100

[LW04] Xiang-Yang Li and Yu Wang. Minimum power assignment in wireless ad hoc networks with spanner property. In *Proc. IEEE Workshop on High Performance Switching and Routing (HPSR'04)*, pages 231–235, 2004. 56

[Mat93] Jiri Matoušek. Range searching with efficient hierarchical cuttings. *Discrete Comput. Geom.*, 10(2):157–182, 1993. 12, 55

[MH01] Harvey J. Miller and Jiawei Han, editors. *Geographic Data Mining and Knowledge Discovery.* Taylor & Francis, London, 2001. 99

[ML05] Kousha Moaveni-Nejad and Xiang-Yang Li. Low-interference topology control for wireless ad hoc networks. *Internatational Journal Ad Hoc & Sensor Wireless Networks*, 1(1–2):41–64, 2005. 50, 51, 66

[MT87] Alistair Moffat and Tadao Takaoka. An all pairs shortest path algorithm with expected time $O(n^2 \log n)$. *SIAM J. Comput.*, 16(6):1023–1031, 1987. 57

[NS07] Giri Narasimhan and Michiel Smid. *Geometric Spanner Networks.* Cambridge University Press, 2007. 57

[NW06] Martin Nöllenburg and Alexander Wolff. A mixed-integer program for drawing high-quality metro maps. In Patrick Healy and Nikola S. Nikolov, editors, *Proc.*

13th Int. Symposium on Graph Drawing (GD'05), volume 3843 of *Lecture Notes in Computer Science*, pages 321–333. Springer-Verlag, 2006. 7, 8

[Por] Porcupine caribou herd satellite collar project. `http://www.taiga.net/satellite/`. 99

[Pra99] Ravi Prakash. Unidirectional links prove costly in wireless ad-hoc networks. In *Proc. 3rd Int. Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications (DIALM'99)*, pages 15–22. ACM Press, 1999. 49

[Pri57] Robert C. Prim. Shortest connection networks and some generalisations. *Bell Systems Technical Journal*, pages 1389–1410, 1957. 50

[Ram99] Edgar A. Ramos. On range reporting, ray shooting and k-level construction. In *Proc. 15th Annu. ACM Sympos. Comput. Geom. (SoCG'99)*, pages 390–399, 1999. 54

[RHS01] John F. Roddick, Kathleen Hornsby, and Myra Spiliopoulou. An Updated Bibliography of Temporal, Spatial, and Spatio-temporal Data Mining Research. In J. F. Roddick and K. Hornsby, editors, *Temporal, spatial and spatio-temporal data mining, TSDM 2000*, volume 2007 of *Lecture Notes in Artificial Intelligence*, pages 147–163, Berlin, 2001. Springer-Verlag. 99

[Rog64] Claude A. Rogers. *Packing and Covering.* Cambridge University Press, Cambridge, England, 1964. 120

[RRT94] Sundaram S. Ravi, Daniel J. Rosenkrantz, and Giri Kumar Tayi. Heuristic and special case algorithms for dispersion problems. *Operations Research*, 42(2):299–310, 1994. 121

[RS91] Jim Ruppert and Raimund Seidel. Approximating the d-dimensional complete Euclidean graph. In *Proc. 3rd Canad. Conf. Comput. Geom. (CCCG'91)*, pages 207–210, 1991. 11

[SB00] Neil Sumpter and Andrew J. Bulpitt. Learning spatio-temporal patterns for predicting object behaviour. *Image Vision and Computing*, 18(9):697–704, 2000. 99

[SC03] Choon-Bo Shim and Jae-Woo Chang. A new similar trajectory retrieval scheme using k-warping distance algorithm for moving objects. In *Proc. of the 4th Int. Conference on Advances in Web-Age Information Management (WAIM'03)*, volume 2762 of *Lecture Notes in Computer Science*, pages 433–444. Springer-Verlag, 2003. 99

[SS00] Weiping Shi and Chen Su. The rectilinear Steiner arborescence problem is NP-complete. In *Proc. 11th Annu. ACM-SIAM Symp. Disc. Algorithms (SODA'00)*, pages 780–787, 2000. 48

[SV86] Robert Sedgewick and Jeffrey S. Vitter. Shortest paths in Euclidean space. *Algorithmica 1*, pages 31–48, 1986. 9

[SZ04] Mikkel Sigurd and Martin Zachariasen. Construction of minimum-weight spanners. In Susanne Albers and Tomasz Radzik, editors, *Proc. 12th Annual European Symposium on Algorithms (ESA'04)*, volume 3221 of *Lecture Notes in Computer Science*, pages 797–808. Springer-Verlag, 2004. 9

[TÓ4] Gabor Fejes Tóth. *Handbook of Discrete and Computational Geometry, 2nd Edition, Jacob E. Goodman and Joseph O'Rourke, editors*, chapter 2, Packing and covering. CRC Press LLC, Boca Raton, FL, 2004. 120

[Thu10] Axel Thue. Über die dichteste Zusammenstellung von kongruenten Kreisen in der Ebene. *Christiania Vid. Selsk. Skr.*, 1:3–9, 1910. 125

[VC06] Florian Verhein and Sanjay Chawla. Mining spatio-temporal association rules, sources, sinks, stationary regions and thoroughfares in object mobility databases. In *Proc.of the 11th Int. Conference on Database Systems for Advanced Applications (DASFAA'06)*, volume 3882 of *Lecture Notes in Computer Science*, pages 187–201. Springer-Verlag, 2006. 99

[vdS94] A. F. van der Stappen. *Motion Planning amidst Fat Obstacles*. Ph.D. dissertation, Dept. Comput. Sci., Utrecht Univ., Utrecht, Netherlands, 1994. 106

[vK98] Marc van Kreveld. On fat partitioning, fat covering, and the union size of polygons. *Comput. Geom. Theory Appl.*, 9(4):197–210, 1998. 106

[vRSWZ05] Pascal von Rickenbach, Stefan Schmid, Roger Wattenhofer, and Aaron Zollinger. A robust interference model for wireless ad-hoc networks. In *Proc. 5th Int. Workshop on Algorithms for Wireless, Mobile, Ad Hoc and Sensor Networks (WMAN'05)*, 2005. CD-ROM. 50, 51

[Wei97] Robert Weibel. Generalization of Spatial Data: Principles and Selected Algorithms. In Marc van Kreveld, Jürg Nievergelt, Thomas Roos, and Peter Widmayer, editors, *Algorithmic Foundations of Geographic Information Systems*, volume 1340 of *Lecture Notes in Computer Science*, chapter 5, pages 99–152. Springer-Verlag, 1997. 7

[WK88] D. W. Wang and Yue-Sun Kuo. A study on two geometric location problems. *Inform. Process. Letters*, 28(6):281–286, 1988. 121

[WS96] Alexander Wolff and Tycho Strijk. The Map-Labeling Bibliography. `http://i11www.ira.uka.de/map-labeling/bibliography/`, 1996. 69

[wtp06] Wildlife tracking projects with GPS GSM collars. `http://www.environmental-studies.de/projects/projects.html`, 2006. 99

[WW07] Dorothea Wagner and Thomas Willhalm. Speed-up techniques for shortest-path computation. In *Proc. 24th Int. Symp. on Theoretical Aspects of Computer Science (STACS'07)*, Lecture Notes in Computer Science, 2007. To appear. 10

[WWZ06] Dorothea Wagner, Thomas Willhalm, and Christos Zaroliagis. Geometric shortest path containers. *ACM Journal on Experimental Algorithms 10*, 2006. Article No. 1.03. 9

[Yao82] Andrew Chi Chih Yao. On constructing minimum spanning trees in k-dimensional spaces and related problems. *SIAM Journal on Computing*, 11(4):721–736, 1982. 11

[ZT99] Chuanming Zong and John Talbot. *Sphere Packings*. Springer-Verlag, 1999. 120

Thanks to ...

... Alex for being obtrusive (sometimes) in a positive sense which includes sharpening my grasp of academic writing and, hopefully also, improving my own abilites of academic writing.

... Joachim and Marc for showing me the easiness of doing research and for inviting me to Sydney and Utrecht, respectively.

... Martin for pushing me for longer working hours by his steady presence in the office (that was engaging).

... Damian, the native English speaker, for revising the introduction.

... all i11-staff, especially Michael, Martin, Étienne, Steffen and Marco, for helping me with LaTeX details or other stuff for which I had been too lazy to figure it out on my own. Though you knew of my laziness, you always helped me. :-)

... my parents—my dad insisted on being named here ;-)—for supporting me on my whole way to the PhD.

... the janitor who—within 4 years—never complained about me taking the bike to the office.

Curriculum Vitae

19.12.76	born in Berlin, Germany
21.06.96	A-levels at Friedrich Schiller Gymnasium Fellbach, Germany
02.09.96 – 30.09.97	alternative service in a residence for mentally disabled persons
01.10.97	enrolment at Universität Konstanz, Germany; major in mathematics, minor in computer science
07.01.03	graduation from Universität Konstanz; title of Master's Thesis: "Über den kleinsten Primitivrest *modulo p*"
01.04.03 – 31.03.07	research assistant of PD Dr. Alexander Wolff at Karlsruhe University; work on project "Geometric Networks and their visualization" funded by the German Research Foundation (DFG)
12.09.05 – 16.09.05	research stay at the University of Utrecht, The Netherlands, under supervision of Prof. Dr. Marc van Kreveld
4.10.05 – 23.12.05	research stay at NICTA, Sydney, Australia under supervision of Dr. Joachim Gudmundsson. Supported by a DAAD (German Academic Exchange Service) grant