A theoretical and experimental study of geometric networks

Citation for published version (APA):
Farshi, M. (2008). *A theoretical and experimental study of geometric networks*. [Phd Thesis 1 (Research TU/e / Graduation TU/e), Mathematics and Computer Science]. Technische Universiteit Eindhoven.
https://doi.org/10.6100/IR630219

DOI:
10.6100/IR630219

Document status and date:
Published: 01/01/2008

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 04. Oct. 2023

A Theoretical and Experimental Study of Geometric Networks

Mohammad Farshi

A Theoretical and Experimental Study of Geometric Networks

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Eindhoven, op gezag van de
Rector Magnificus, prof.dr.ir. C.J. van Duijn, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op dinsdag 8 april 2008 om 16.00 uur

door

Mohammad Farshi

geboren te Yazd, Iran

Dit proefschrift is goedgekeurd door de promotor:

prof.dr. M.T. de Berg

Copromotor:
dr. J. Gudmundsson

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Farshi, Mohammad

A theoretical and experimental study of geometric networks/ door Mohammad Farshi. -
Eindhoven : Technische Universiteit Eindhoven, 2008.
Proefschrift. - ISBN 978-90-386-1135-8
NUR 993
Subject headings: computational geometry / data structures / algorithms
CR Subject Classification (1998): I.3.5, E.1, F.2.2

Promotor: prof.dr. M.T. de Berg (Technische Universiteit Eindhoven)

Copromotor: dr. J. Gudmundsson (National ICT Australia)

Kerncommissie:
prof.dr. M.H. Overmars (Utrecht University)
prof.dr. M. Smid (Carleton University)
prof.dr. J.J. van Wijk (Technische Universiteit Eindhoven)

Ministry of Science, Research and Technology
Islamic Republic of Iran

The work in this thesis is supported by the Ministry of Science, Research and Technology of I. R. Iran under scholarship no. 800.341.

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

IPA dissertation series 2008-12

© Mohammad Farshi 2008. All rights are reserved. Reproduction in whole or in part is prohibited without the written consent of the copyright owner.

Printing: Eindhoven University Press

Illustration cover: a 2-spanner on 532 US cities [NS07].

Contents

Preface

اول دفتر به نام ایزد دانا   صانع وپروردگار حی توانا

(سعدی شیرازی)

This thesis is the result of almost four years of work where I have been accompanied and supported by many people. I now have the pleasant opportunity to express my gratitude to all of them.

First of all, I would like to express my deep and sincere gratitude to my supervisors, Mark de Berg and Joachim Gudmundsson for giving me the possibility to work under their supervision. Thanks to Mark who took the risk of accepting me, as a person with no knowledge in computational geometry, in his group and to Joachim who was my daily supervisor and who helped me to understand the concept of spanners, discuss problems and read my manuscripts, not only when he was in TU/e but also after he left Eindhoven. This work would not have been possible without their support and encouragement, and I am grateful for their valuable friendship.

I would also like to thank my distinguished co-authors during my PhD study Mohammad Ali Abam, Hee-Kap Ahn, Mark de Berg, Joachim Gudmundsson, Panos Giannopoulos, Christian Knauer, Michiel Smid, and Yajun Wang, the results in this thesis is the product of our joint work. I would like to express my thanks to the people who made the joint works possible: Alexander Wolf and Xavier Goaoc for inviting me to the Korean Workshop on Computational Geometry and the organizers of the workshop on geometric networks and metric space embedding at Schloss Dagstuhl. I also thank Joachim Gudmundsson for inviting me to NICTA and his hospitality during my visit.

The members of my thesis committee are gratefully acknowledged for reading the thesis, providing useful comments and being present at my defense session. It was my privilege to have Mark de Berg, Joachim Gudmundsson, Mark Overmars, Michiel Smid and Jack van Wijk in the thesis committee and Rolf Klein in the defense opposition.

My PhD study was supported by a scholarship from the Ministry of Science, Research and Technology of I. R. Iran. I would like to take this opportunity to thank them for their support. I also would like to thank Khosro Piri, Farhad Rahmati and Mohammad Hossein Abdollahi, academic representatives and directors of Iranian students in Europe for their help.

I was very lucky to work with absolutely fantastic people in the Algorithms group. I thank them all for making my years at Eindhoven delightful. Special thanks go to Sonja Joosten for her help at the start of my study and Astrid Volkers for her help at the final stages of my work. I also thank my great officemates Yuval Nir, Mark Schroders and Peter Hachenberger.

A heart-felt thanks goes to Mohammad Ali Abam for his valuable friendship, enjoyable discussions, which always come with nice ideas, and tea breaks. I learned a lot from him and I'm looking forward to continue to work with him.

I must thank friends/families whose company made my and my family's life much more enjoyable and the weekend/holiday meetings were our most wonderful times in the Netherlands. I would like to mention the following families: Cheema, Eslami, Ghasemzadeh, Mousavi, Moosavi Nezhad, Nazarpoor, Nikoufard, Rezaeian, Talebi, Vahedi. I would also like to express my gratitude to Mohammad Ali Abam, Mohammad Eslami, Hamed Fatemi, Amir Hossein Ghamarian, Mohammad Ghasemzadeh, Kamyar Malakpoor, Mohammad Reza Mousavi, Mohammad Moosavi Nezhad, Mahmoud Nikoufard, Reza Rezaeian, Mohammad Samimi, Saeid Talebi, Mostafa Vahedi for their kind friendship.

I cannot end without thanking my family, on whose constant encouragement and love I have relied throughout my life. I am grateful to my parents for their unconditional support, unflinching courage and conviction during my study. I would like to thank my wife Hamideh and my son Alireza for their unwavering support, patience and understanding during this time. It is to them that I dedicate this work, with love and gratitude.

همه بودی از بود او هست نام   تمام اوست دیگر همه ناتمام

(نظامی گنجوی)

Chapter 1

Introduction

1.1 Geometric networks

A *network* is, informally speaking, a collection of "objects" with certain "connections" between the objects. An obvious example of a network is a computer network. Here the objects are computers and there is a connection between two computers if there is a physical cable connecting them. Other obvious examples are road or railway networks. In the latter type of network the objects are the stations, and the connections are the tracks connecting the various stations.

There are also many other types of networks, however, where the connections do not necessarily have a physical realization. For example, in social sciences one studies *social networks*, where the objects could be people and two people are connected if they have a certain social relationship— see Figure 1.1. Another example is formed by biological networks such as neural networks, gene regulatory networks or protein-protein interaction networks— see Figure 1.2.

Sometimes networks can be rather small—the network in Figure 1.1 for example is quite small—but sometimes they can also be huge, like the Internet (which has more than 500 million hosts) and the webgraph—a graph whose nodes correspond to static pages on the web and whose arcs correspond to links between these pages—which has billions of pages that are connected by billions of links. For example, in 2003 Google search engine indexed 1.6 billions of URLs and this increased to 4.2 billions in 2004.

From these examples it is clear that networks form a fundamental model in a variety of application areas. It is not surprising therefore, that there has been a lot

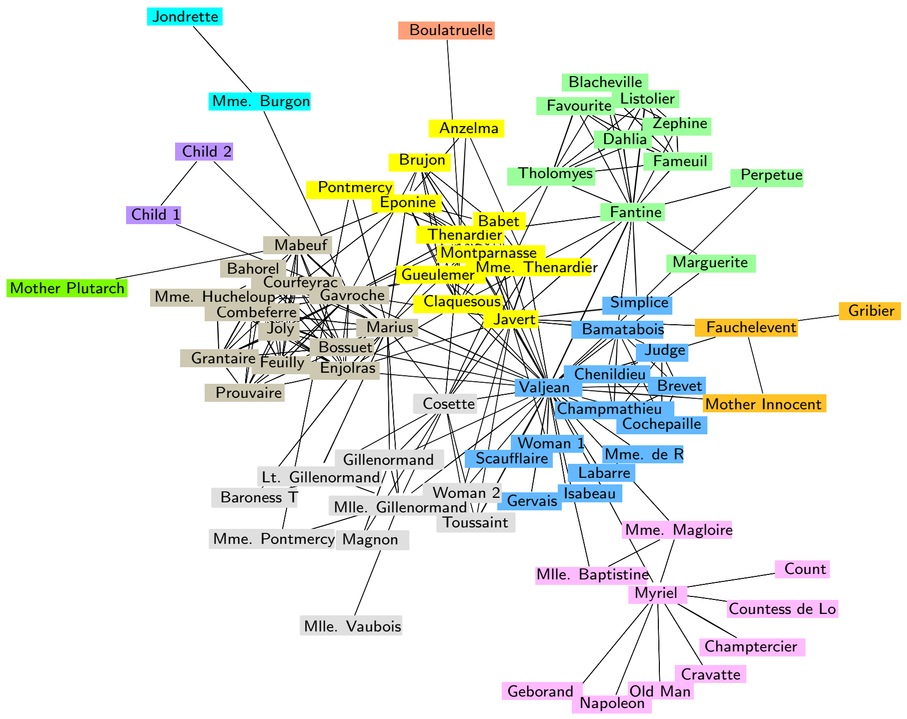


Figure 1.1: The network of interactions between major characters in the novel *Les Miserables* by Victor Hugo, divided into 11 communities represented by different colors [NG04].

of research on designing, analyzing, and optimizing networks. The mathematical concept corresponding to networks are *graphs*. (In the sequel, we will use the terms graph and network interchangeably.) A graph G is a pair (V, E) where V is a (usually finite) set of *nodes* and $E \subset V \times V$ is the set of connections between the nodes. Based on the network, we can make the graph (edge/vertex) weighted or directed/undirected. For example for a graph which models a road network, the weight of an edge can represent the length of the road. Also by making it directed we can show one-way or two-way roads.

In some applications it is relevant to assume that the set of vertices of the graph is a subset of a metric space and the weight of each edge in the graph is the distance between its endpoints. A metric space is defined as a set where a distance between elements of the set is defined. The distance function $\mathbf{d}$ (called a metric) should be non-negative, symmetric, have $\mathbf{d}(x, y) = 0$ if and only if $x = y$, and satisfy the triangle inequality. Obviously any set of points in the plane with the Euclidean distance as a distance function makes a metric space. As a more complex example, for each graph $G(V, E)$ with positive edge weights, we can easily define a distance function (or metric) $\mathbf{d}$ such that $(V, \mathbf{d})$ makes a metric space. To

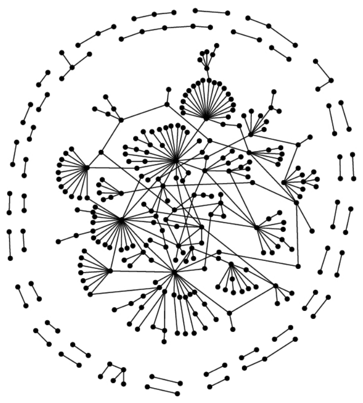


Figure 1.2: A yeast protein interaction network [MS02].

this end, we define the distance between each pair $(u, v) \in V^2$ as the length of the shortest path between u and v in G. We call $(V, \mathbf{d})$ the *metric space induced by the graph G.*

A *geometric* network comes from adding geometry to a network. More precisely, if the vertex set of the network is a subset of d-dimensional Euclidean space, and the metric is the Euclidean metric, then the network is a geometric network. Geometric networks model naturally (at least approximately) many real-life networks, such as road networks, railway networks, and so on. In this case we can use geometric properties to design or analyze a network. In this thesis we always consider undirect geometric networks, unless explicitly stated otherwise.

1.2 t-Spanners

When designing a network for a given set V of points, several criteria can be taken into account. In many applications it is important to ensure a fast connection between every pair of points in V. For this it would be ideal to have a direct connection between every pair of points—the network would then be a complete graph—but in most applications this is unacceptable due to the high costs. This leads to the concepts of spanners, as defined below. Spanners were introduced by Peleg and Schäffer [PS89] in the context of distributed computing and by Chew [Che86] in a geometric context.

Let V be a subset of a metric space and for each u and v in V, let $\mathbf{d}(u, v)$ denote the distance between u and v in the metric space. The aim is to design a network which ensures a "fast" connection between each pair of points but with a sub-quadratic number of edges. Note that the number of edges in the complete graph is quadratic in the number of vertices. To obtain a "good" graph with a sub-quadratic number of edges we have to allow a (hopefully short) detour between pairs of points. For example, instead of asking for a direct connection between each pair of points, we are allowed to have a small detour. The length of the detour is given by a real number $t > 1$. Given the parameter t, a connection between u and v in the graph G is "good" if the distance between u and v in the graph G, denoted by $\mathbf{d}_G(u,v)$, is at most t times the distance between u and v. We call such a path a *t-path* between u and v in the graph G. The ratio $\mathbf{d}_G(u,v)/\,\mathbf{d}(u,v)$ is called the *dilation* between u and v in G. A graph G is a t-spanner of its vertex set if the dilation between each pair of vertices in G is bounded by t. Figure 1.3 is an example of 1.5-spanner on 532 US cities.

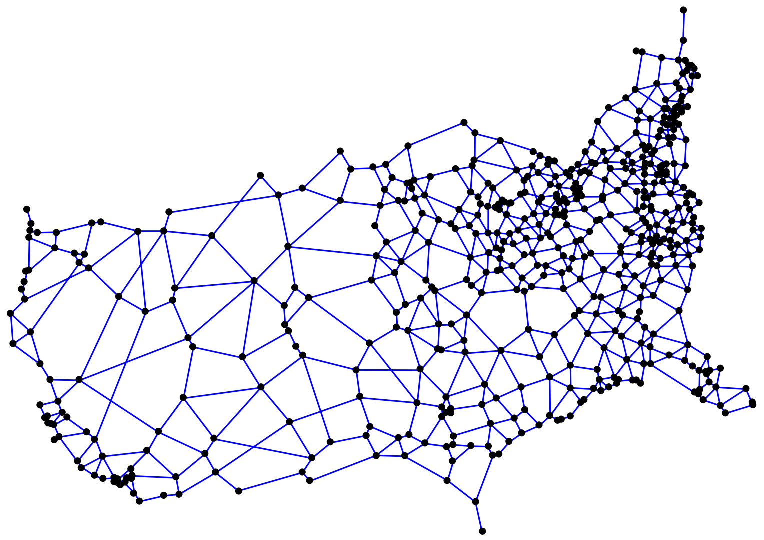


Figure 1.3: A 1.5-spanner networks on 532 US cities [NS07].

Here is the formal definition of t-spanner.

Definition 1.2.1 (t-spanner of a point set) *A graph $G(V,E)$ is a t-*spanner *of V, for a real number $t > 1$, if for each pair of points $u, v \in V$, we have that*

$$\mathbf{d}_G(u,v) \leq t \cdot \mathbf{d}(u,v).$$

The *dilation*, or *stretch factor*, of a network $G(V,E)$ is the minimum t for which G is a t-spanner of V. When a geometric graph satisfies the t-spanner condition, we call it a *geometric* t-spanner.

Intuitively, the t-spanner property is stronger than the graph connectivity property, i.e. we should not only have a path between each pair of points in the graph but also the length of the path should be close to the distance between the pair of points. The parameter t decides how close the t-spanner approximates the complete graph. In other words, the closer t is to one, the closer the t-spanner is to the complete graph.

We can easily extend Definition 1.2.1 to a t-spanner of a given graph. Recall that for any two vertices u and v in a graph G with positive edge weights, $\mathbf{d}_G(u, v)$ denotes the distance between u and v in the graph V, that is, the length of the shortest path between u and v in G.

Definition 1.2.2 (t-spanner of a graph) *Let $G(V, E)$ be a graph with positive edge weights. A graph $G'(V, E')$ on the same vertex set but with edge set $E' \subseteq E$ is a* t-spanner *of G if for each pair of vertices $u, v \in V$ we have that*

$$\mathbf{d}_{G'}(u, v) \leq t \cdot \mathbf{d}_G(u, v).$$

By comparing Definition 1.2.1 and Definition 1.2.2 one can see two differences. First, a t-spanner of a given graph G is a subgraph of G. Therefore computing a t-spanner of a graph G is sometimes called a *pruning* of G. The second difference is that, to check the t-spanner condition, the distance between each pair in the t-spanner is compared to the distance between them in the input graph. Note that in Definition 1.2.2 it is not necessary that the vertices of the input graph belong to a metric space. Instead we use the metric space induced by the input graph to measure the distance between them.

A t-spanner on a point set V—which is subset of a metric space—can be seen as a t-spanner of the complete graph on V, denoted by $G_c(V)$, where the weight of each edge in the complete graph is the distance between its endpoints.

The main question in designing t-spanners is whether spanners exist that have "good" properties. The desirable properties are the following:

Size: The number of edges in the graph. This is the most important measurement and generating spanners with a small, ideally near-linear, size is desirable. The fact that geometric spanners with a near-linear number of edges and small dilation exist has made the construction of spanners one of the fundamental tools in the development of approximation algorithms for geometrical problems.

Degree: The maximum number of edges incident to a vertex. This property has been shown to be useful in the development of approximation algorithms [GLNS02a, Yao82] and for the construction of ad hoc networks [ALW$^+$03, Li03] where small degree is essential in trying to develop fast localized algorithms.

Weight: The weight of a network $G(V, E)$ is the sum of the edge weights. (Recall that for geometric graphs the weight of an edge is simply the Euclidean distance between its two endpoints.) The best that can be achieved is a constant times the weight of the minimum spanning tree, denoted $wt(MST(V))$. Low weight spanners have found applications in areas such as metric space searching [NP03, NPC02]—see Section 1.3.2—and broadcasting in communication networks [FPZW04]—see Section 1.3.3. Also, it has been used in the construction of several approximation algorithms, see [CL00, RS98].

Spanner diameter: Defined as the smallest integer $\mathbb{D}$ such that for any pair of vertices u and v in V, there is a t-path in the graph between u and v containing at most $\mathbb{D}$ edges. For wireless ad hoc networks it is often desirable to have small spanner diameter since it determines the maximum number of times a message has to be transmitted in a network. If a graph has spanner diameter $\mathbb{D}$ then it is said to be a $\mathbb{D}$-hop network.

It has been shown that for any set V of n points in $\mathbb{R}^d$ and for any fixed $t > 1$ there exists a t-spanner with $\mathcal{O}(n)$ edges, constant degree, and whose total weight is $\mathcal{O}(wt(MST(V)))$ [DN97, NS07]. Arya *et al.* [AMS99] designed a randomized algorithm which generates a t-spanner of expected linear size and expected logarithmic spanner diameter.

Note that some of the above properties are competing, e.g., a graph with constant degree cannot have constant spanner diameter, and a graph with small spanner diameter cannot have a linear number of edges [ADM$^+$95].

In most cases we are interested in t-spanners, where t is close to 1. More precisely, we are interested in schemes where we can specify any $t > 1$ and then obtain a spanner of dilation t. To emphasize the fact that t is close to 1, we will sometimes speak about $(1 + \varepsilon)$-spanners instead of t-spanners.

1.3 Why spanners?

Spanners for complete graphs as well as for arbitrary weighted graphs find applications in robotics, network topology design, distributed systems, design of parallel machines, and many other areas. Recently spanners found interesting practical applications in areas such as metric space searching [NP03, NPC02] and broadcasting in communication networks [ALW$^+$03, FPZW04, Li03].

Several well-known theoretical results also use the construction of t-spanners as a building block. For example, Rao and Smith [RS98] made a breakthrough by showing an optimal $\mathcal{O}(n \log n)$-time approximation scheme for the well-known Euclidean *traveling salesperson problem*, using t-spanners (or banyans). Similarly, Czumaj and Lingas [CL00] showed approximation schemes for minimum-cost multi-connectivity problems in geometric graphs. The problem of constructing geometric spanners has received considerable attention from a theoretical perspective, see [ADD$^+$93, ADM$^+$95, AMS99, BGM04, DHN93, DN97, DNS95, GLN02, Kei88, KG92, LL92, LNS02, Sal91, Vai91], the surveys [Epp00, GK07, Smi00] and the book by Narasimhan and Smid [NS07]. Note that significant research has also been done in the construction of spanners for general graphs, see for example, the book by Peleg [Pel00] or the recent work by Elkin and Peleg [EP04] and Thorup and Zwick [TZ01]. In this section we mention some of the applications of t-spanners.

1.3.1 Approximate minimum spanning tree

The problem of finding a minimum weight spanning tree (MST) of a given graph has recently attracted a lot of attention. There are linear time algorithms for computing MST in a randomized expected case model [KKT95] or with the assumption that edge weight are integers [FW94]. For deterministic comparison-based algorithms, slightly superlinear bounds are known [GGST86].

In the geometric case, for any dimension d, one can find a minimum spanning tree of a set of n points in $\mathcal{O}(n^2)$ time by constructing the complete graph and computing all the edge weights (=pairwise distances) in $\mathcal{O}(n^2)$ time, and then running any MST algorithm (such as Prim's algorithm). Most of the faster algorithms for the geometric case use a simple idea: find a sparse subgraph of the complete graph which contains an MST, and then compute an MST of this sparse subgraph. In the plane, the Delaunay graph is the appropriate graph to use since it only has $\mathcal{O}(n)$ edges and can be constructed in $\mathcal{O}(n \log n)$ time [SH75]. It is not helpful in higher dimensions because the Delaunay graph can have a quadratic number of edges [Eri01].

Salowe [Sal91] showed that if G' is a (geometric) t-spanner of a graph G and $wt(MST(G))$ denotes the weight of the minimum spanning tree of G, then $wt(MST(G')) \leq t \cdot wt(MST(G))$. Using this result, we can use any (sparse) t-spanner of a graph to compute an approximate minimum spanning tree of the graph. For more details about approximating the minimum spanning tree see the survey by Eppstein [Epp00].

1.3.2 Metric space searching

The problem of "approximate" proximity searching in metric spaces is to find the elements of a set which are "close" to a given query under some similarity criterion. Similarity searching has become a fundamental computational task in a variety of application areas, including multimedia information retrieval, pattern recognition, computer vision and biomedical databases. In such environments, an exact match has little meaning, and proximity/distance concepts (similarity/dissimilarity) are typically much more fruitful for searching. In all of these applications we have a metric which shows the similarity between objects. The smaller the distance is between two object, the more similar they are — see [ZADB06] for more details.

A typical query q is:

- find all elements in the database which are within distance r from q.
- find the k closest elements to q in the database.

If the database contains n elements, we can answer a query by performing $\mathcal{O}(n)$ distance computations. But evaluating distances is expensive and the goal is to reduce the number of distance evaluations. In general, there are several methods to reduce the number of distance evaluations, see the survey [CNBYM01]. A widely used technique for this is AESA [Rui86]. The main drawback of this technique is that it computes all pairwise distances and stores them in a matrix which requires a lot of space.

Navarro *et al.* [NPC02] used a t-spanner as a data structure for metric space searching to reduce the space needed for the AESA. The key idea is to regard the t-spanner as an approximation of the complete graph of distances among the objects, and to use it as a compact device to simulate the large matrix of distances required by successful search algorithms such as AESA. The t-spanner property implies that we can use the shortest path in the t-spanner to estimate any distance with bounded error factor t.

They propose several t-spanner construction, update, and search algorithms and experimentally evaluated them. The experiments show that their technique is competitive against current approaches, and that it has a great potential for further improvements.

1.3.3 Broadcasting in communication networks

Wireless networks consist of a set of wireless devices (called nodes) which are spread over a geographical area. These nodes are able to perform processing as well as communicating with each other. The nodes can communicate via multi-hop wireless channels: a node can reach all nodes inside its transmission region while nodes far away from each other communicate through intermediate nodes. Wireless communication networks have applications in various situations such as emergency relief, environmental monitoring, and so on.

There are two common types of wireless networks: *sink-based* networks and *ad hoc* networks. In a sink-based network, like cellular wireless networks, there is one or multiple sink nodes which are in charge of collecting data from all nodes and managing the whole network. On the other hand, in ad hoc networks there are no such sink nodes and all nodes are equal in terms of communication and network management.

Energy consumption and network performance are the most critical issues in wireless ad hoc networks, because wireless devices are usually powered by batteries only and have limited computing capability and memory.

A wireless ad hoc network is modeled by a set V of n wireless nodes distributed in a two-dimensional plane. Each node has the same maximum transmission range R which, by a proper scaling, we can assume that all nodes have the maximum transmission range to be equal to 1. These wireless nodes define a unit disk graph, denoted by $UDG(V)$, in which there is an edge between two nodes if the Euclidean distance between them is at most 1. The most common power-attenuation model in the literature claims that the power needed to support a link (u, v) is $\|uv\|^{\beta}$, where $\|uv\|$ is the Euclidean distance between u and v and β is a real constant between 2 and 5 depending on the wireless transmission environment. So the power consumption of a network is a function of the weight of the network. The minimum possible weight for a connected network is the weight of a minimum spanning tree, however, this might have low performance.

Several graph theoretical models are used to design ad hoc networks with low energy consumption and good performance—see [Wat05]. Using a low weight t-spanner of the unit disk graph is one of the ways—see [ALW^{+}03]. This gives us more flexibility to construct a network which has low energy consumption as well as good performance, like bounded degree. Note that in ad hoc networks, a network with small node degree, has a better chance to has small interference.

1.3.4 Proteins visualization

One of the most important open problems in bioinformatics is the problem of protein folding. A protein is a long chain of molecules called amino acids. In nature there exist 20 different amino acids and several experiments show that the 3D-structure of a protein is completely determined by the sequence of amino acids.

The protein-folding problem is the problem of determining the 3D-structure of a protein given its amino acid sequence. Various computational methods have been applied to tackle the protein-folding problem, with varying success. Among these are neural networks, approximation algorithms, metaheuristics, branch-and-bound, distributed systems and computational geometry.

Nowadays, using high computing power and large scale storage, researchers are able to computationally simulate the protein-folding process in atomistic details. Such simulations often produce a large number of folding trajectories, each consisting of a series of 3D conformations of the protein under study. As a result, effectively managing and analyzing such trajectories is becoming increasingly important.

Recently, Russel and Guibas [RG05] suggested using geometric spanners for mapping a simulation to a more discrete combinatorial representation. They apply geometric spanners to discover the proximity between different segments of a protein across a range of scales, and track the changes of such proximity over time. This makes the task of understanding and exploring the space of protein motions easier. Using their structure it is possible to visualize proteins in motion which none of the commonly used software packages such as `RasMol` [Ras], `ProteinExplorer` [Mar02], or `SPV` [GP97] have been able to achieve.

1.4 Thesis overview

In this thesis we consider several problems related to the design and analysis of geometric networks.

In Chapter 2, we introduce the concept of region-fault tolerant spanners for planar point sets, and prove the existence of region-fault tolerant spanners of small size. For a geometric graph G on a point set V and a region F, we define $G \ominus F$ to be what remains of G after the vertices and edges of G intersecting F have been removed. A *$\mathcal{C}$-fault tolerant t-spanner* is a geometric graph G on V such that for any convex region F, the graph $G \ominus F$ is a t-spanner for $G_c(V) \ominus F$, where $G_c(V)$ is the complete geometric graph on V. Fault-tolerant spanners provide high levels of availability and reliability in network connections. These networks keep their good properties, even after some part of the network is destroyed e.g. by a natural disaster.

We prove that any set V of n points admits a $\mathcal{C}$-fault tolerant $(1+\varepsilon)$-spanner of size $\mathcal{O}(n \log n)$, for any constant $\varepsilon > 0$; if adding Steiner points is allowed then the size of the spanner reduces to $\mathcal{O}(n)$, and for several special cases we show how to obtain region-fault tolerant spanners of size $\mathcal{O}(n)$ without using Steiner points. We also consider *fault-tolerant geodesic t-spanners*: this is a variant where, for any disk D, the distance in $G \ominus D$ between any two points $u, v \in V \setminus D$ is at most t times the geodesic distance between u and v in $\mathbb{R}^2 \setminus D$. We prove that for any point set V we can add $\mathcal{O}(n)$ Steiner points to obtain a fault-tolerant geodesic $(1+\varepsilon)$-spanner of size $\mathcal{O}(n)$. These results are based on [AdBFG07].

In applications—think of road networks, for instance—a spanner network is sometimes expanded by adding one or more extra connections. The main question is then how to do the expansion such that the resulting network is as good as possible. In Chapter 3 we study a problem of this type. In particular, we consider the problem of adding an edge to a given network such that the dilation of the resulting network is minimized. We present one exact algorithm and several approximation algorithms. The best approximation algorithm computes a $(2+\varepsilon)$-approximation of the optimal solution in $\mathcal{O}(nm + n^2 \log n)$ time using $\mathcal{O}(n^2)$ space, where n is the number of vertices and m is the number of edges in the input network. For the special case, when the dilation of the input network is constant, we can improve the approximation factor to $1+\varepsilon$ and the running time to $\mathcal{O}(n^2)$. These results are based on [FGG05a].

Chapter 4 studies the problem of dilation optimal edge deletion. More precisely we are given a geometric network in the plane and we want to find an edge in the network such that its removal minimizes, or maximizes, the dilation in the network. An obvious application is when we want to remove some connections in an existing network, e.g. due to budget consideration, and we want to know which edges should be removed to minimize the effect on the quality of the network. We solve the problem in the restricted case when the network is a simple cycle. A randomized algorithm is presented which, given a cycle on a set of n points, computes in $\mathcal{O}(n \log^3 n)$ expected time, the edge of the cycle whose removal results in a polygonal path of smallest possible dilation. It is also shown that the edge whose removal gives a polygonal path of largest possible dilation can be computed in $\mathcal{O}(n \log n)$ time. If the input cycle is a convex polygon, the latter problem can be solved in $\mathcal{O}(n)$ time. Finally, it is shown that given a cycle C, for each edge e of C, a $(1-\varepsilon)$-approximation to the dilation of the path $C \setminus \{e\}$ can be computed in $\mathcal{O}(n \log n)$ total time. These results are based on [AFK$^+$07].

In Chapter 5 we present algorithms for computing the spanner diameter of a t-spanner. This is the first algorithm for computing spanner diameter of a t-spanner, to the best of our knowledge. The time complexity of the most efficient algorithm is $\mathcal{O}(\mathbb{D} \cdot mn)$, where n is the number of vertices, m is the number of edges and $\mathbb{D}$ is the spanner diameter of the input graph, and it requires $\mathcal{O}(n)$ space. We also compare the running time of the presented algorithms experimentally. These results are based on [FG06].

The empirical study of algorithms is a rapidly growing research area. Implementing algorithms and testing their performance shows their efficiency in practice and bring the algorithms to the practical stage. In Chapter 6 we experimentally study the performance and quality of the most common t-spanner algorithms for points in the Euclidean plane. The experiments are discussed and compared to the theoretical results and in several cases we suggest modifications that are implemented and evaluated. The quality measurements that we consider are the number of edges, the weight, the maximum degree, the spanner diameter and the number of crossings. We compare the running times of the algorithms and suggest some improvements. This is the first time an extensive comparison has been made between the construction algorithms of t-spanners. These results are based on [FG05] and [FG07].

Finally in Chapter 7 we conclude the thesis and state some open problems.

Chapter 2

Region-Fault Tolerant Spanners

2.1 Introduction

As we mentioned before, geometric networks have applications in VLSI design, telecommunications, robotics and distributed systems. The major issues with designing such a network are performance and reliability. The spanner concept captures the performance when short connections between the points are important. The main question is then whether spanners exist that have a small dilation and a small, ideally near-linear, number of edges. Other desirable properties of a spanner are for example that the total weight of the edges is small, or that the maximum degree is low. As discussed in the introduction, such spanners do indeed exist: it has been shown that for any set P of n points in the plane and for any fixed $\varepsilon > 0$ there exists a $(1+\varepsilon)$-spanner with $\mathcal{O}(n/\varepsilon)$ edges, $\mathcal{O}(1/\varepsilon)$ degree, and whose total weight is $\mathcal{O}(wt(MST(P))/\varepsilon^4)$, where $wt(MST(P))$ is the weight of a minimum spanning tree of P [DN97, NS07].

Reliability is concerned with the fact that in many applications the nodes and/or links in a network may fail. In a computer network, for instance, nodes may fail because computers can crash, and in a road network links may fail because roads can become inaccessible due to accidents or maintenance. A network is reliable when it retains its good properties even after some nodes or links fail. With respect to spanners this means there should still be a short path between any two nodes in what remains of the spanner after the fault.

Fault-tolerant spanner were introduced by Levcopolous *et al.* [LNS98]. In this paper and a follow-up paper [LNS02] they showed the existence of k-vertex (or: k-edge) fault-tolerant geometric spanners with $\mathcal{O}(nk \log n)$ edges. This was improved by Lukovszki [Luk99], who presented a fault-tolerant spanner with $\mathcal{O}(nk)$ edges, which is optimal. Later Czumaj and Zhao [CZ03] showed that a greedy approach produces a k-vertex (or: k-edge) fault-tolerant geometric $(1+\varepsilon)$-spanner with degree $\mathcal{O}(k)$ and total weight $\mathcal{O}(k^2 \cdot wt(MST(P)))$; these bounds are asymptotically optimal.

The papers on fault-tolerant spanners mentioned above all consider faults that can destroy an arbitrary collection of k vertices or edges. For geometric spanners, however, it is natural to consider *region faults*: faults that do not destroy an arbitrary collection of vertices and edges, but faults that destroy all vertices and edges intersecting some geometric fault region. This is relevant, for instance, when the spanner models a road network and a natural (or other) disaster makes all the roads in some region inaccessible. This is the topic of this chapter: we study the existence of small spanners in the plane[1] that are tolerant against region faults. Before we present our results, let us define region-fault tolerance more precisely.

Let $\mathcal{F}$ be a family of regions in the plane, which we call the *fault regions*. For a fault region $F \in \mathcal{F}$ and a geometric graph $\mathcal{G}$ on a point set P, we define $\mathcal{G} \ominus F$ to be the part of $\mathcal{G}$ that remains after the points from P inside F and all edges that intersect F have been removed from the graph—see Figure 2.1. (For simplicity we assume that a region fault F does not contain its boundary, i.e. only vertices and edges intersecting the interior of F will be affected.)

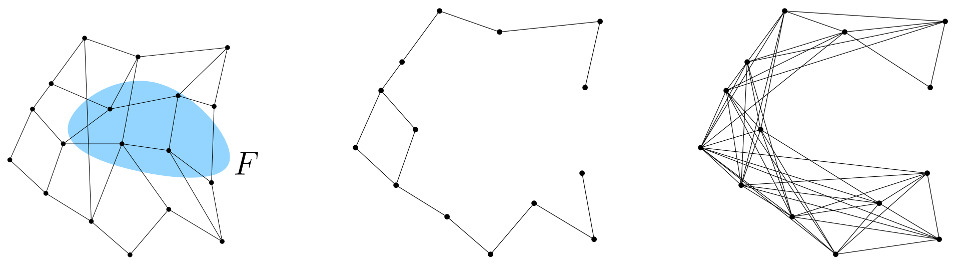


Figure 2.1: The input graph $\mathcal{G}$ and a fault region F, the graph $\mathcal{G} \ominus F$, and the graph $\mathcal{G}_c(P) \ominus F$.

Definition 2.1.1 *An $\mathcal{F}$-fault tolerant t-spanner is a geometric graph $\mathcal{G}$ on P such that for any region $F \in \mathcal{F}$, the graph $\mathcal{G} \ominus F$ is a t-spanner for $\mathcal{G}_c(P) \ominus F$, where $\mathcal{G}_c(P)$ is the complete geometric graph on P.*

[1]The concepts and many of the results carry over to d-dimensional Euclidean space. However, we feel the concept is mainly interesting in the plane, so we confine ourselves to the planar case in this chapter.

We are mainly interested in the case where $\mathcal{F}$ is the family $\mathcal{C}$ of convex sets.[2] We shall also consider the case where we are allowed to add Steiner points to the graph. In other words, instead of constructing a geometric network for P, we are allowed to construct a network for $P \cup Q$ for some set Q of Steiner points. Then we say that a graph $\mathcal{G}$ on $P \cup Q$ is a *$\mathcal{C}$-fault tolerant Steiner t-spanner* for P if, for any $F \in \mathcal{C}$ and any two points $u, v \in P \setminus F$, the distance between u and v in $\mathcal{G} \ominus F$ is at most t times their distance in $\mathcal{G}_c(P) \ominus F$.

We also study another variant of region-fault tolerance. In this variant we require that the distance between any two points u, v in $\mathcal{G} \ominus F$ is at most t times the geodesic distance between u and v in $\mathbb{R}^2 \setminus F$. Note that the geodesic distance in $\mathbb{R}^2 \setminus F$—that is, the length of a shortest path in $\mathbb{R}^2 \setminus F$—is never more than the distance between u and v in $\mathcal{G}_c(P) \ominus F$. We call a spanner with this property an *$\mathcal{F}$-fault tolerant geodesic t-spanner*. It is not difficult to show that finite size $\mathcal{F}$-fault tolerant geodesic spanners do not exist unless we are allowed to use Steiner points. Even in the case of Steiner points, finite size $\mathcal{F}$-fault tolerant geodesic spanners do not exist when $\mathcal{F}$ is the family $\mathcal{C}$ of all convex sets. Hence, we restrict our attention to $\mathcal{D}$-fault tolerant geodesic spanners, where $\mathcal{D}$ is the family of disks in the plane.

We obtain the following results.

- In Section 2.2 we present a general method to convert a well-separated pair decomposition (WSPD) [CK93] for P into a $\mathcal{C}$-fault tolerant spanner for P. We use this method to obtain linear-size $\mathcal{C}$-fault tolerant $(1+\varepsilon)$-spanners for points in convex position and for points distributed uniformly at random inside the unit square, and to obtain linear-size $\mathcal{C}$-fault tolerant Steiner $(1+\varepsilon)$-spanners for arbitrary point sets.

- In Section 2.4 we consider two special cases, fat triangulations and polygonal region faults with limited number of edge directions, for which linear-size $\mathcal{C}$-fault tolerant spanners can be obtained.

- In Section 2.5 we study small $\mathcal{C}$-fault tolerant (non-Steiner) spanners for arbitrary point sets. By combining a more relaxed version of the WSPD with ideas from Θ-graphs [Kei88], we show that any point set P admits a $\mathcal{C}$-fault tolerant $(1+\varepsilon)$-spanner of size $\mathcal{O}(n \log n)$.

- In Section 2.6 we address a slightly different problem. Instead of designing a $\mathcal{C}$-fault tolerant spanner, it is also interesting to check whether an existing network is $\mathcal{C}$-fault tolerant or not. In Section 2.6 we give an algorithm which, given a graph $\mathcal{G}$, checks whether it is fault tolerant under convex region faults.

[2] It is easy to see that there are no small region-fault tolerant t-spanners with respect to non-convex faults: if $\mathcal{HH}$ denotes the family of regions that are the union of two half-planes, then $\mathcal{G}_c(P)$ is the only $\mathcal{HH}$-fault tolerant t-spanner for P, for any finite t.

- In Section 2.7 we study the geodesic case. We show that for any set P of n points there exists a $\mathcal{D}$-fault tolerant geodesic Steiner $(1+\varepsilon)$-spanner with $\mathcal{O}(n)$ edges and $\mathcal{O}(n)$ Steiner points.

2.2 Constructing $\mathcal{C}$-fault tolerant spanners using the WSPD

In this section we show a general method to obtain a $\mathcal{C}$-fault tolerant spanner from a well-separated-pair decomposition of a point set P. Although in general the spanner can have $\Omega(n^2)$ edges, we show that for some special cases a smaller bound can be proven. We also show how to use the approach to obtain small Steiner spanners. Before we start we prove a general lemma showing that, when constructing $\mathcal{C}$-fault tolerant spanners, we can in fact restrict our attention to half-plane faults. This lemma will also be used in later sections. Let $\mathcal{H}$ be the family of half-planes in the plane.

Proposition 2.2.1 *A geometric graph $\mathcal{G}$ on a set P of points in the plane is a $\mathcal{C}$-fault tolerant t-spanner if and only if it is an $\mathcal{H}$-fault tolerant t-spanner.*

Proof. Obviously a graph is $\mathcal{H}$-fault tolerant if it is $\mathcal{C}$-fault tolerant. To prove the other direction assume that $\mathcal{G}$ is an $\mathcal{H}$-fault tolerant t-spanner and that $F \in \mathcal{C}$ is an arbitrary convex region fault. We need to prove that between every pair of points $u, v \in P \setminus F$ there is a path in $\mathcal{G} \ominus F$ of length at most t times the length of the shortest path in $\mathcal{G}'_c = \mathcal{G}_c(P) \ominus F$.

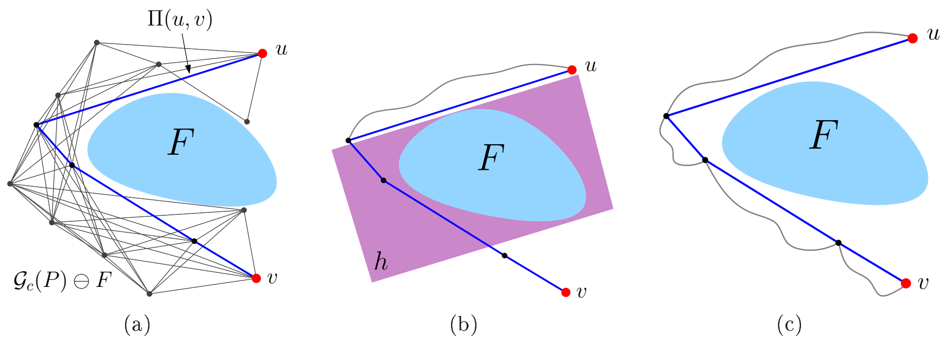


Figure 2.2: Illustration of the Proposition 2.2.1

If u and v are not connected in $\mathcal{G}'_c$ we are done. Otherwise, let $\Pi(u,v)$ be a shortest path between u and v in $\mathcal{G}'_c$, see Figure 2.2(a). We claim that for every edge (p,q) in $\Pi(u,v)$ there is a path in $\mathcal{G} \ominus F$ of length at most $t \cdot \|pq\|$. Since the edge (p,q) lies outside F and F is convex, there must be a half-plane h that

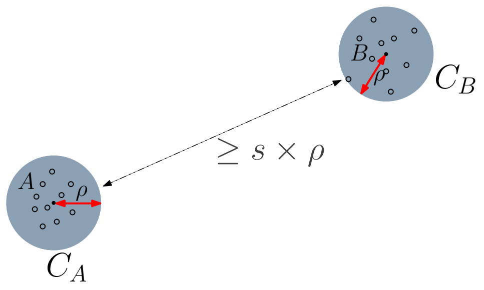


Figure 2.3: Definition of well-separated pair.

contains F but does not intersect (p, q), see Figure 2.2(b). Since $\mathcal{G}$ is an $\mathcal{H}$-fault tolerant t-spanner there is a path $\Pi(p, q)$ between p and q in $\mathcal{G} \ominus h$ of length at most $t \cdot \|pq\|$. Furthermore, since $F \subset h$ the path $\Pi(p, q)$ also exists in $\mathcal{G} \ominus F$, see Figure 2.2(c). The claim and, hence, the lemma follows. ◻

2.3 Well-separated pair decomposition

The well-separated pair decomposition (WSPD) was developed by Callahan and Kosaraju [CK95]. This powerful data structure has been used to solve a wide variety of geometric problems like N-body problems (used in astronomy, molecular dynamics, fluid dynamics and plasma physics), surface reconstruction [FR02] and many more applications. We briefly review this decomposition here because we will use it in the next chapters.

Definition 2.3.1 *[CK95] Let $s > 0$ be a real number, and let A and B be two finite sets of points in $\mathbb{R}^d$. We say that A and B are* well-separated *with respect to s, if there are two disjoint d-dimensional balls C_A and C_B, having the same radius, such that*

1. *C_A contains A,*
2. *C_B contains B, and*
3. *the minimum distance between C_A and C_B is at least s times the radius of C_A—see Figure 2.3.*

The parameter s will be referred to as the *separation constant*. The next lemma follows easily from Definition 2.3.1.

Lemma 2.3.2 *[CK95] Let A and B be two finite sets of points that are well-separated with respect to s, let x and p be points of A, and let y and q be points of B. Then*

(i) $\|xy\| \leq (1 + 4/s) \cdot \|pq\|$, *and*

(ii) $\|px\| \leq (2/s) \cdot \|pq\|$.

Intuitively, by Lemma 2.3.2, if A and B are well-separated, then the distance between a point in A and a point in B is roughly the same as the distance between the two sets A and B. Also the distance between a pair of points which both lie in one of the sets is much smaller than the distance between the two sets.

Definition 2.3.3 *[CK95] Let P be a set of n points in $\mathbb{R}^d$, and let $s > 0$ be a real number. A* well-separated pair decomposition *(WSPD) for P with respect to s is a sequence of pairs of non-empty subsets of P, $(A_1, B_1), \ldots, (A_m, B_m)$, such that*

1. *A_i and B_i are well-separated with respect to s, for $1 \leq i \leq m$.*
2. *for any two distinct points p and q of P, there is exactly one pair (A_i, B_i) in the sequence, such that (i) $p \in A_i$ and $q \in B_i$, or (ii) $q \in A_i$ and $p \in B_i$,*

The integer m is called the size *of the WSPD.*

In other words, a well-separated pair decomposition of a point set P consists of a set of well-separated pairs that cover all the pairs of distinct points, i.e., any two distinct points belong to the different sets of some pair.

Callahan and Kosaraju showed that for any point set in Euclidean space and for any constant $s > 0$, there always exist a WSPD of size $m = \mathcal{O}(s^d n)$ and it can be computed in $\mathcal{O}(s^d n + n \log n)$ time. In the geometric problems, when we need all point pairs in a set, we can easily use a WSPD of the point set as an approximation with linear size.

2.3.1 Constructing a $\mathcal{C}$-fault tolerant spanner

Callahan and Kosaraju [CK93] showed that the WSPD can be used to obtain a small $(1 + \varepsilon)$-spanner. Similar ideas were used earlier by Salowe [Sal91, Sal92] and Vaidya [Vai88, Vai89, Vai91]. To obtain the $(1 + \varepsilon)$-spanner one simply computes a WSPD $\mathcal{W}$ with respect to $s := 4 + 8/\varepsilon$, and then for each well-separated pair $(A, B) \in \mathcal{W}$ one adds an arbitrary edge connecting a point from A to a point in B.

Unfortunately this construction is not $\mathcal{C}$-fault tolerant, because a fault F can destroy the spanner edge that connects a pair (A, B), while some other edges between A and B (which are not in the spanner) may survive the fault. Hence, we need to add more than a single edge for (A, B). Let $\mathrm{CH}(A)$ and $\mathrm{CH}(B)$ denote the convex hulls of A and B, respectively. At first sight it seems that adding the two outer tangents of $\mathrm{CH}(A)$ and $\mathrm{CH}(B)$ to our spanner may lead to a $\mathcal{C}$-fault tolerant spanner, but this is not the case either. Instead, we will triangulate the region in between the two convex hulls in an arbitrary manner, as illustrated in Figure 2.4(a).

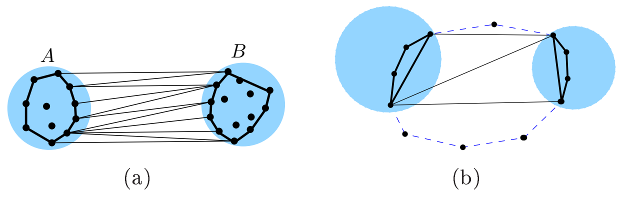


Figure 2.4: (a) Illustrating the construction of the WSPD-graph. (b) Points in convex position.

Let $E(A, B)$ be the set of edges in the triangulation added between $\mathrm{CH}(A)$ and $\mathrm{CH}(B)$, and let $\mathcal{G}$ be the obtained graph with edge set $\mathcal{E} := \sum_{(A,B)\in\mathcal{W}} E(A, B)$. Note that any triangulation between $\mathrm{CH}(A)$ and $\mathrm{CH}(B)$ has the same number of edges. Throughout the chapter we will use the notation $|\cdot|$ to denote the number of elements in a set.

Lemma 2.3.4 *The graph $\mathcal{G}$ is a $\mathcal{C}$-fault tolerant $(1+\varepsilon)$-spanner for P of size $\sum_{(A,B)\in\mathcal{W}} |E(A, B)|$.*

Proof. The size of the graph is obviously $\sum_{(A,B)\in\mathcal{W}} |E(A, B)|$, so it remains to show that it is a $\mathcal{C}$-fault tolerant $(1+\varepsilon)$-spanner. Now we observe that for any half-plane h, $\{(A \setminus h, B \setminus h) : (A, B) \in \mathcal{W}\}$ is a WSPD for $P \setminus h$. Hence, by Proposition 2.2.1 and the properties of the WSPD it is sufficient to show the following: Let h be a half-plane fault, let u, v be points not in h, and let (A, B) be a pair with $u \in A$ and $v \in B$; then there is an edge $e \in E(A, B)$ between $\mathrm{CH}(A)$ and $\mathrm{CH}(B)$ that is outside h.

To see this we first prove that, given a point set P and a triangulation T of P, the graph $T \ominus h$ is connected for any half-plane h. Assume without loss of generality that h is below and bounded by a horizontal line. Since any point of $P \backslash h$ not on the convex hull must have an edge connecting it to a point further away from h, we can walk from p away from h along edges of T until we reach a point on the convex hull of P. Moreover, any two convex hull points in $P \backslash h$ can be connected by convex hull edges outside h. It follows that $T \ominus h$ is indeed connected.

Now consider any triangulation T on $A \cup B$ that includes $E(A, B)$. Then $T \ominus h$ must be connected. Since $u, v \notin h$, and $u \in A$ and $v \in B$, this means there must be an edge $e \in E(A, B)$ outside h. $\boxminus$

2.3.2 Linear-size spanners for special cases

The method described above can be used to get small $\mathcal{C}$-fault tolerant spanners for several special cases. For example, if P is in convex position then $|E(A,B)| \leq 3$ for any pair (A,B) in the decomposition, see Figure 2.4(b), so we get:

Theorem 2.3.5 *For any set P of n points in convex position in the plane and any $\varepsilon > 0$, there exists a $\mathcal{C}$-fault tolerant $(1+\varepsilon)$-spanner of size $\mathcal{O}(n/\varepsilon^2)$.*

Next we show that we can also get a $\mathcal{C}$-fault tolerant spanner whose expected size is linear if the point set P is generated by picking n points uniformly at random in the unit square.

Lemma 2.3.6 *Let P be a set of n uniformly distributed points in the unit square and A be a sub-square of the unit square. Then the expected number of points on the convex hull of $P \cap A$ is $\mathcal{O}(\log(n \cdot \mathbf{area}(A)))$.*

Proof. If n points are uniformly distributed in the unit square then it is known that the expected number of points on the convex hull of the points is $\mathcal{O}(\log n)$ [HP98, RS63].

Now let X be the number of points on the convex hull of $P \cap A$ and let $Y := |P \cap A|$. Clearly $\mathbf{EXP}[Y] = n \cdot \mathbf{area}(A)$. By the law of total expectation [Ros98, Proposition 4.1] if X and Y are two random variables then

$$\mathbf{EXP}[X] = \mathbf{EXP}\left[\mathbf{EXP}[X|Y]\right],$$

therefore

$$\begin{aligned}
\mathbf{EXP}[X] &= \mathbf{EXP}\left[\mathbf{EXP}[X\,|\,Y]\right] \\
&= \mathbf{EXP}[\mathcal{O}(\log(Y))] \\
&\leq \mathcal{O}\left(\log\left(\mathbf{EXP}[Y]\right)\right) \qquad \text{(Jensen's inequality [Ros98, p. 418])} \\
&= \mathcal{O}(\log(n \cdot \mathbf{area}(A))).
\end{aligned}$$

⊔⊓

Now we combine the ideas from the previous section with Lemma 2.3.6 to construct a $(1+\varepsilon)$-spanner of the uniformly distributed point set P.

Theorem 2.3.7 *Let P be a set of n points uniformly distributed in the unit square U. For any $\varepsilon > 0$ there is a $\mathcal{C}$-fault tolerant $(1+\varepsilon)$-spanner of expected size $\mathcal{O}(n/\varepsilon^2)$ for P.*

Proof. Construct a quadtree partitioning of U into smaller and smaller squares, until each square has size (side length) roughly $1/\sqrt{n}$. So the area of any leaf is roughly $1/n$ which means the expected number of points in a leaf region is $\mathcal{O}(1)$. The quadtree has $\mathcal{O}(n)$ leaves. Level ℓ of the quadtree corresponds to a regular subdivision of U into squares of size $1/2^\ell$. One can show that there exists a WSPD $\mathcal{W} := \{(A_i, B_i)\}_i$ of size $\mathcal{O}(n/\varepsilon^2)$ for P such that for each i, the pair (A_i, B_i) either corresponds to two squares at the same level, or A_i and B_i are both singleton points that lie in nearby cells (or the same cell) of the final subdivision. Moreover, if we denote by n_ℓ the number of pairs of the WSPD at level ℓ of the quadtree, then $n_\ell = \mathcal{O}(2^{2\ell}/\varepsilon^2)$. The existence of a WSPD with these properties follows rather directly from the results of Fischer and Har-Peled [FHP05]. For completeness we briefly sketch an argument for our setting.

For a node ν of the quadtree, let $P(\nu)$ denote the subset of points from P inside the square corresponding to ν. Consider a level ℓ of the quadtree. For each pair of nodes ν, ν' at level ℓ such that the point sets $P(\nu)$ and $P(\nu')$ are well-separated while the point sets of the parents of ν and ν' are not well-separated, we put the pair $(P(\nu), P(\nu'))$ into the WSPD. In addition, for each pair of leaf nodes μ, μ' such that $P(\mu)$ and $P(\mu')$ are not well-separated, we put a pair $(\{p\}, \{q\})$ into the WSPD for every pair $p \in P(\mu)$ and $q \in P(\mu')$. It is easy to verify that this indeed defines a WSPD. The bound on the number of pairs added for each level follows from a standard packing argument.

Now consider a square σ at level ℓ. By Lemma 2.3.6, because the area of σ is $1/2^{2\ell}$, the expected size of the convex hull of the points in σ is $\mathcal{O}(\log(n/2^{2\ell}))$.

If (A, B) is an arbitrary pair in $\mathcal{W}$ which appears at level ℓ of the quadtree then

$$\begin{aligned}\mathbf{EXP}\left[|E(A,B)|\right] &\leq \mathbf{EXP}\left[|\operatorname{CH}(A)| + |\operatorname{CH}(B)|\right]\\ &= \mathbf{EXP}\left[|\operatorname{CH}(A)|\right] + \mathbf{EXP}\left[|\operatorname{CH}(B)|\right]\\ &= \mathcal{O}(\log(n/2^{2\ell})).\end{aligned}$$

Therefore

$$\begin{aligned}\mathbf{EXP}\left[\sum_{(A_i,B_i)\in\mathcal{W}} |E(A_i,B_i)|\right] &= \sum_{(A_i,B_i)\in\mathcal{W}} \mathbf{EXP}\left[|E(A_i,B_i)|\right]\\ &= \sum_{\ell=1}^{\frac{1}{2}\log n} \mathcal{O}\left(n_\ell \log(n/2^{2\ell})\right)\\ &= \sum_{\ell=1}^{\frac{1}{2}\log n} \mathcal{O}\left((2^{2\ell}/\varepsilon^2)\log(n/2^{2\ell})\right).\end{aligned}$$

To bound this summation, we set $m := \frac{1}{2}\log n$ and we get:

$$\begin{aligned}
\sum_{\ell=1}^{\frac{1}{2}\log n} 2^{2\ell}\log(n/2^{2\ell}) &= \sum_{\ell=1}^{m} 2^{2\ell}(2m-2\ell) \\
&= 2\sum_{\ell=1}^{m} 2^{2\ell}(m-\ell) \\
&= 2\sum_{k=0}^{m-1} 2^{2(m-k)}\cdot k \qquad \text{(by setting } k=m-\ell\text{)} \\
&= 2^{2m+1}\sum_{k=0}^{m-1}\frac{k}{2^{2k}} \\
&\leq 2^{2m+1}\sum_{k=0}^{\infty}\frac{k}{2^{2k}} \\
&= \mathcal{O}(n).
\end{aligned}$$

Hence the expected size of the generated $(1+\varepsilon)$-spanner is $\mathcal{O}(n/\varepsilon^2)$. ◻

2.3.3 $\mathcal{C}$-fault tolerant Steiner spanners

Above we showed that the WSPD can be used to construct $\mathcal{C}$-fault tolerant spanners of small size when the points are in convex position or uniformly distributed. For arbitrary point sets, however, the size of the spanner may be $\Omega(n^2)$. In this section we will show that if we are allowed to add Steiner points, we can always use the above method to get a linear-size spanner:

Theorem 2.3.8 *For any set P of n points in the plane and any $\varepsilon > 0$, one can construct a $\mathcal{C}$-fault tolerant Steiner $(1+\varepsilon)$-spanner of size $\mathcal{O}(n/\varepsilon^2)$ by adding at most $4(n-1)$ Steiner points.*

The idea is to add a set Q of Steiner points to P such that $|E(A,B)| = \mathcal{O}(1)$ for any pair (A,B) in the WSPD of $P \cup Q$. Then the theorem immediately follows from Lemma 2.3.4.

Our method is based on the WSPD construction by Fisher and Har-Peled [FHP05]. Their construction uses a compressed quadtree, which is defined as follows.

Let $\mathcal{T}(P)$ be the quadtree on P. We denote the square corresponding to a node $\nu \in \mathcal{T}(P)$ by $\sigma(\nu)$, and the subset of points from P inside $\sigma(\nu)$ by $P(\nu)$. When some of the points are very close together, a quadtree can have superlinear size. A *compressed quadtree* $\mathcal{T}^*(P)$ for P therefore removes internal nodes ν from $\mathcal{T}(P)$ for which all points from P lie in the same quadrant of $\sigma(\nu)$. A compressed quadtree has at most $n-1$ internal nodes. Fisher and Har-Peled [FHP05] show that one can obtain a WSPD of size $\mathcal{O}(s^2 n)$ for P that consists of pairs $(P(\nu_1), P(\nu_2))$ where ν_1 and ν_2 are nodes in $\mathcal{T}^*(P)$.

The set Q of Steiner points that we use is defined as follows. Let $\mathcal{T}^*(P)$ be a compressed quadtree for P. Without loss of generality, we may assume that no point from P lies on any of the splitting lines. For each internal node ν of $\mathcal{T}^*(P)$, we add the four corner points of $\sigma(\nu)$ to Q. To avoid degenerate cases, we slightly move each point into the interior of $\sigma(\nu)$. Note that two (or more) squares $\sigma(\nu_1)$ and $\sigma(\nu_2)$ may share, for instance, their top right corner. In this case we add the (slightly shifted) corner point only once. The resulting set Q has size at most $4(n-1)$. The next lemma finishes the proof of Theorem 2.3.8.

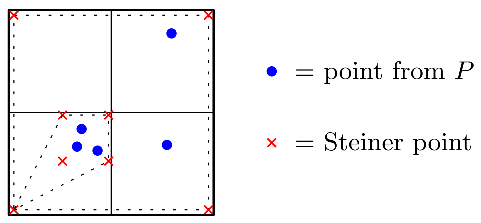


Figure 2.5: Illustration for the proof of Lemma 2.3.9.

Lemma 2.3.9 *Let $\mathcal{T}^*(\overline{P})$ be a compressed quadtree for $\overline{P} := P \cup Q$, where the initial bounding square U is the same as for $\mathcal{T}^*(P)$, and let ν be an internal node of $\mathcal{T}^*(\overline{P})$. Then $\mathrm{CH}(\overline{P}(\nu))$ has at most four vertices.*

Proof. If the square $\sigma(\nu)$ contains zero or one point from P then at most one Steiner point has been added inside $\sigma(\nu)$, and the lemma is true. If $\sigma(\nu)$ contains two or more points then there are two cases, both illustrated in Figure 2.5.

Let μ be the node of $\mathcal{T}^*(P)$ such that $P(\mu) = \overline{P}(\nu) \cap P$. Note that the four shifted corners of $\sigma(\mu)$ were added as Steiner points to Q. If $\sigma(\mu) = \sigma(\nu)$ then $\mathrm{CH}(\overline{P}(\nu))$ is a square. Otherwise, $\sigma(\mu) \subset \sigma(\nu)$. In this case $\mathrm{CH}(\overline{P}(\nu))$ is formed by three of the four corners of $\sigma(\mu)$ together with the unique corner of $\sigma(\nu)$ that generated a Steiner point at some ancestor of ν in $\mathcal{T}^*(P)$, see Figure 2.5. Hence, in this case $\mathrm{CH}(\overline{P}(\nu))$ has four vertices as well. $\square$

2.4 Special cases

In this section we present algorithms for constructing fault tolerant spanners in two special cases. In Section 2.4.1, we give an algorithm that construct a $\mathcal{C}$-fault tolerant spanner for any point set which admits a fat triangulation. Then, in Section 2.4.2, we construct spanners which are fault tolerant under more limited region faults.

2.4.1 $\mathcal{C}$-fault tolerant fat triangulations

We call a triangulation of a point set *α-fat* if all its triangles are α-fat or, in other words, if all angles in the triangulation are at least α. Karavelas and Guibas [KG01] showed that any α-fat triangulation T of a point set P is a 2α-spanner for P. To make the spanner $\mathcal{C}$-fault tolerant, we add some extra edges: we add an edge between every pair of points $u, v \in P$ such that there is a path between u and v in T consisting of two edges.

Theorem 2.4.1 *Let P be a set of n points in the plane and let T be a α-fat triangulation of P. Then we can augment T with a set of $\mathcal{O}(n/\alpha)$ extra edges such that the resulting geometric graph is a $\mathcal{C}$-fault tolerant 2α-spanner.*

Proof. We connect each node v to all other nodes within two steps from v. In other words we add an edge between each pair of points connected by a path of two edges. Let T' be the result. Obviously we add at most $\sum_v D \cdot \deg(v)$ edges, where D is the maximum degree in the triangulation T and $\deg(v)$ is the degree of the node v of T. Since for each triangulation $\sum_v \deg(v) \leq 6n$ we add $\mathcal{O}(D \cdot n)$ edges to the triangulation T. Note that $D = \mathcal{O}(1/\alpha)$ since T is a α-fat triangulation.

Now the claim is that T' is a $\mathcal{C}$-fault tolerant 2α-spanner. Using Proposition 2.2.1, it suffices to show that T' is an $\mathcal{H}$-fault tolerant 2α-spanner. Let h be an arbitrary half-plane and $p, q \in P \setminus h$ be two arbitrary points. Karavelas and Guibas [KG01, Theorem 2.1] proved that there exist a 2α-path $\Pi(p, q)$ between p and q in T zig-zagging above and below the line connecting p to q—see Figure 2.6(a). Note that all the edges in this path intersect the segment between p and q.

If all the nodes on $\Pi(p, q)$ lies outside h we are done. Otherwise assume $p' \in \Pi(p, q)$ lies inside h and let q_1 and q_2 be the other endpoints of the two edges on $\Pi(p, q)$ incident to p'—see Figure 2.6(b). Since the segment pq lies outside h and any edge on $\Pi(p, q)$ intersect pq, the points q_1 and q_2 lie outside h. Because we added edges between pairs within two steps—the dashed edges in Figure 2.6(b)—we can replace the two fat edges (p', q_1) and (p', q_2) with (q_1, q_2). This way we can obtain a path that stays outside h and with length at most the length of $\Pi(p, q)$. $\square$

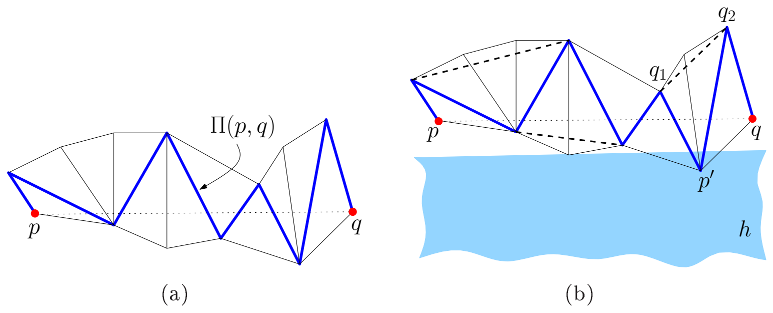


Figure 2.6: Illustration for the proof of Theorem 2.4.1

2.4.2 Limited boundary directions

Now we put some limitation on the region faults. Let $\mathcal{H}'$ be a family of half-planes with at most k boundary directions. By the following procedure, which uses the WSPD, we can make an $\mathcal{H}'$-fault tolerant $(1+\varepsilon)$-spanner of each point set P of n points with $\mathcal{O}(kn/\varepsilon^2)$ size.

Let $\{d_i\}_{i=1}^k$ be the set of directions where the boundary of each half-plane in $\mathcal{H}'$ is parallel to one of d_i's. For each $1 \le i \le k$ assume $d_i^{(1)}$ and $d_i^{(2)}$ are the two directions perpendicular to d_i. To construct a $\mathcal{H}'$-fault tolerant spanner we compute a WSPD of the point set with respect to $s := \frac{4(\varepsilon+2)}{\varepsilon}$. Then for each pair (A,B) in the WSPD and for each direction d_i, we add two edges, one between the extreme point of A and B in direction $d_i^{(1)}$ and the other between the extreme points of A and B in direction $d_i^{(2)}$. See Algorithm 2.4.1 for more details.

Algorithm 2.4.1: BOUNDED-BOUNDARY-DIRECTIONS

```
    **Input**: $P$, $\varepsilon > 0$ and a set of directions $\{d_i\}_{i=1}^k$.
    **Output**: $\mathcal{H}'$-fault tolerant $(1+\varepsilon)$-spanner $\mathcal{G} = (P, \mathcal{E})$.
1   $\mathcal{W} :=$ WSPD of $P$ w.r.t. $s := \frac{4(\varepsilon+2)}{\varepsilon}$;
2   foreach $(A,B) \in \mathcal{W}$ do
3       for $i := 1, 2, \ldots, k$ do
4           Add an edge between extreme points of $A$ and $B$ in direction $d_i^{(1)}$;
5           Add an edge between extreme points of $A$ and $B$ in direction $d_i^{(2)}$;
6       end
7   end
8   return $\mathcal{G}$;
```

Theorem 2.4.2 *Let P be a set of n points the plane and $\mathcal{H}'$ be a family of half-planes with at most k boundary directions. Then for each $\varepsilon > 0$, we can construct an $\mathcal{H}'$-fault tolerant $(1+\varepsilon)$-spanner of size $\mathcal{O}(kn/\varepsilon^2)$ in $\mathcal{O}((n \log n + kn)/\varepsilon^2)$ time.*

Proof. Obviously we add at most $2k$ edges for each pair in WSPD and therefore the size of the graph is $\mathcal{O}(kn/\varepsilon^2)$. Also the time complexity of the algorithm is straight forward. Therefore to complete the proof, we show that the graph generated by Algorithm 2.4.1 is $\mathcal{H}'$-fault tolerant. To show this, it is sufficient to show that for each $h \in \mathcal{H}'$ and any (A, B) in the WSPD which is situated partially outside h, we have an edge outside h which connect A to B.

Since A and B are partially outside h, the extreme points of them in at least one of the directions perpendicular to the boundary of h is outside h. This means that the edges between the extreme points, which are added by the algorithm, lies outside h. ⊔⊓

Remark 2.4.3 At first it may seem that we can generalize the results to any family of convex polygons with bounded number of edge directions (for example axis-parallel polygons). However as you can see in Figure 2.7 this is not the case.

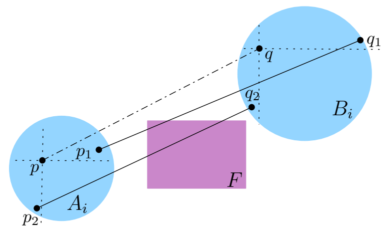


Figure 2.7: Counterexample for axis-parallel polygonal faults.

2.5 $\mathcal{C}$-fault tolerant spanners for arbitrary point sets

In this section we consider the problem of constructing a sparse $\mathcal{C}$-fault tolerant $(1+\varepsilon)$-spanner for an arbitrary set P of n points in the plane without using Steiner points. The method that was described in the previous section does not guarantee a small spanner in general. Here we will describe a method that is guaranteed to result in a spanner of size $\mathcal{O}(n \log n)$.

Throughout this section $d(\cdot,\cdot)$ denotes the (Euclidean) shortest distance between two objects (points, disks, etc.), and **radius**(D) denotes the radius of a disk D.

2.5.1 SSPDs and fault-tolerant spanners

The problem with the WSPD in our application is that, even though the number of pairs in the WSPD is $\mathcal{O}(n)$, the total number of points over all the pairs can be $\Theta(n^2)$. Therefore we will introduce a relaxed version of the WSPD, the SSPD.

Definition 2.5.1 *Let A and B be two sets of points in the plane, and let $s > 0$ be a constant. We say that A and B are* semi-separated with respect to separation constant *s if there are two disjoint disks D_A and D_B, such that*

(i) D_A contains A and D_B contains B,

(ii) $d(D_A, D_B) \geq s \cdot \min(\mathbf{radius}(D_A), \mathbf{radius}(D_B))$.

Thus we allow the balls D_A and D_B to be of different sizes and we only require that the distance between the disks is large relative to the smaller disk. Note that using the same notations we can reformulate the definition of well-separated with respect to s as $d(D_A, D_B) \geq s \cdot \max(\mathbf{radius}(D_A), \mathbf{radius}(D_B))$.

We now define our SSPD.

Definition 2.5.2 *Let P be a set of n points in the plane and let $s > 0$ be a real number. A* semi-separated pair decomposition *(SSPD) for P with respect to s is a collection $\{(A_1, B_1), \ldots, (A_m, B_m)\}$ of pairs of non-empty subsets of P such that*

1. *A_i and B_i are semi-separated with respect to s, for all $i = 1, \ldots, m$.*
2. *for any two distinct points p and q of P, there is exactly one pair (A_i, B_i) in the collection, such that (i) $p \in A_i$ and $q \in B_i$ or (ii) $q \in A_i$ and $p \in B_i$.*

The *weight* of a set A, denoted by $|A|$, is defined as the number of points in A, the weight of a semi-separated pair (A, B) is the sum of the weights of A and B, and the weight of an SSPD is the total weight of all the pairs. Later we will prove that it is possible to compute an SSPD of weight $\mathcal{O}(n \log n)$. First, however, we will show how to use the SSPD to obtain a $\mathcal{C}$-fault tolerant spanner. The idea is to add edges to the spanner for each pair in the SSPD. Because the pairs in an SSPD are only semi-separated, however, adding a single edge for every pair does not necessarily lead to a good spanner. Therefore we use an idea that is also used in the construction of Θ-graphs [Cla87, Kei88].

Consider a pair (A, B) in an SSPD for P. Then there exist two disjoint disks D_A and D_B that contain A and B respectively, and for which

$$d(D_A, D_B) \geq s \cdot \min(\mathbf{radius}(D_A), \mathbf{radius}(D_B)).$$

Assume without loss of generality that $\mathbf{radius}(D_A) \leq \mathbf{radius}(D_B)$, and let o_A denote the center of D_A—see Figure 2.8(a). The set $E(A, B)$ of edges added to the spanner for the pair (A, B) is found as follows.

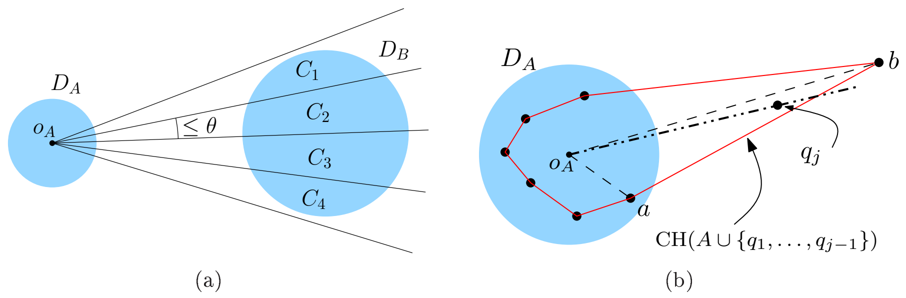


Figure 2.8: (a) The cones of angle at most θ defined with respect to o_A and A. (b) Illustration for the proof that q_j is outside $\mathrm{CH}(A \cup \{q_1, \ldots, q_{j-1}\})$.

1. Partition the plane into $k := \lceil 2\pi/\theta \rceil$ cones $C_1, \ldots, C_k$, all with apex at o_A and with interior angle at most θ, where θ is a suitable constant to be specified later. Let $B^{(i)} := B \cap C_i$ denote the subset of points from B inside the cone C_i; here we assume without loss of generality that no point lies on the boundary between two cones.

2. Let $\mathrm{CH}(A)$ be the convex hull of A. For each $B^{(i)}$, we sort the points in $B^{(i)}$ in order of increasing distance to o_A. Let $q_1, q_2, \ldots$ denote the sorted list of points. We process each point q_j in order as follows. Let $\mathrm{CH}(A')$ be the convex hull of the set $A' = A \cup \{q_1, \ldots, q_{j-1}\}$. We add all edges between q_j and the vertices of A on $\mathrm{CH}(A')$ for which the edge does not intersect $\mathrm{CH}(A')$. Next we update $\mathrm{CH}(A')$ by adding the point q_j. After processing all points $q_i \in B^{(i)}$, we have produced a set $E(A, B^{(i)})$ of edges. The set $E(A, B)$ is simply $\cup_{1 \leq i \leq k} E(A, B^{(i)})$.

Note that in step 2 for each j, the point q_j is outside $\mathrm{CH}(A \cup \{q_1, \ldots, q_{j-1}\})$. To show this assume that q_j is inside $\mathrm{CH}(A')$, for $A' = A \cup \{q_1, \ldots, q_{j-1}\}$. The ray from $o = o_A$ to q_j intersects an edge of $\mathrm{CH}(A')$—see Figure 2.8(b). Let a and b be the endpoints of this edge. Note that q_j lies inside $\triangle oab$. On the other hand, $a, b \in A \cup \{q_1, \ldots, q_{j-1}\}$ and therefore $|oa| \leq |oq_j|$ and $|ob| \leq |oq_j|$. This contradicts the fact that q_j is inside $\triangle oab$.

By construction the edges that we are adding to $E(A, B^{(i)})$ do not cross. Since the number of sets $B^{(i)}$ is $\mathcal{O}(1/\theta)$ and $\sum |B^{(i)}| = |B|$, we have:

Lemma 2.5.3 $|E(A, B)| = \mathcal{O}(|A|/\theta + |B|)$.

To prove that the approach generates a fault-tolerant spanner, we need the following lemma. Consider the ordered set $B^{(i)}$ of the points in B inside the cone C_i.

Lemma 2.5.4 *Let h be a half-plane fault such that both A and $B^{(i)}$ have at least one point outside h. Of all the points in $B^{(i)}$ outside h, let q_j be the one with minimum distance to o_A. There is an edge in $E(A, B^{(i)})$ connecting q_j to a point $p \in A$ outside h.*

Proof. By assumption, q_j is outside h and there is at least one point from A outside h. On the other hand, by the choice of q_j, the points $q_1, \ldots, q_{j-1}$ are all inside h. As mentioned before, q_j is outside $\mathrm{CH}(A \cup \{q_1, \ldots, q_{j-1}\})$ and therefore q_j is a vertex of $\mathrm{CH}(A \cup \{q_1, \ldots, q_j\})$. Let a and b be the neighbors of q_j on $\mathrm{CH}(A \cup \{q_1, \ldots, q_j\})$. We have two cases:

Case 1: a or b lies outside h. In this case we are done because that neighbor belongs to A.

Case 2: a and b are inside h. If we extend the edges (q_j, a) and (q_j, b) then the cone with apex at q_j contains $A \cup \{q_1, \ldots, q_{j-1}\}$. By assumption there exists at least one point of A outside h and therefore there is at least one point $p \in A$ such that $p \notin h$ and p lies on the convex hull $\mathrm{CH}(A \cup \{q_1, \ldots, q_{j-1}\})$—see Figure 2.9. The point p is visible from q_j because all the points on $\mathrm{CH}(A \cup \{q_1, \ldots, q_{j-1}\})$ which are on the side of line (a, b) containing q_j are visible from q_j. Hence there is an edge connecting q_j to p and so we are done. $\boxed{}$

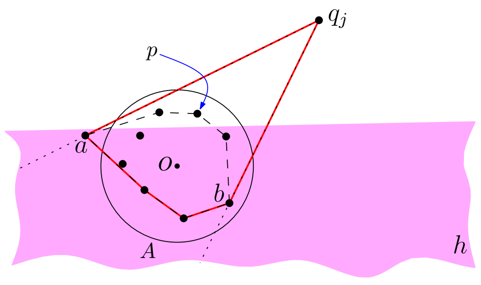


Figure 2.9: Illustrating the proof of Lemma 2.5.4.

Let $\mathcal{G}$ be the graph obtained after applying the above procedure to every pair (A, B) in the SSPD for the point set P. Next we prove that $\mathcal{G}$ is a $(1+\varepsilon)$-spanner if we choose the separation constant s and the angle θ suitably and, moreover, that it is $\mathcal{C}$-fault tolerant. For this we will need the following condition on the structure of the SSPD (which will be satisfied by the SSPD we will construct later).

Monotonicity condition: Suppose p, q are two points that are in the same set X of some pair (A_i, B_i) of the SSPD —thus $X = A_i$ or $X = B_i$—and let (A_j, B_j) be the unique pair in the SSPD such that $p \in A_j$ and $q \in B_j$, or $p \in B_j$ and $q \in A_j$. Then the weights of A_j and B_j are both less than the weight of X.

Lemma 2.5.5 *If θ is chosen such that $\cos\theta - \sin\theta > 1/t$ and the separation constant s of the SSPD is taken as $s := \frac{3t+1}{(\cos\theta - \sin\theta)t-1}$, then the graph $\mathcal{G}$ is an $\mathcal{H}$-fault tolerant t-spanner.*

Proof. Let h be an arbitrary half-plane. To prove the lemma we must show that for each pair of points $p, q \in P$ outside h there is a t-path connecting them in $\mathcal{G}\ominus h$. According to the definition of the SSPD there exists a semi-separated pair (A, B) such that $p \in A$ and $q \in B$ (or vice versa). The proof is done by induction on the maximum weight of A and B.

Base case: If the maximum weight of A and B is 1, then both sets are singletons and therefore we must have an edge between them.

Induction hypothesis: Assume that the lemma holds for all points in pairs whose maximum weight is less than k, for some $k > 1$.

Induction step: Suppose the maximum weight of A and B is k. Let D_A and D_B be two disks containing A resp. B such that

$$d(D_A, D_B) \geq s \cdot \min(\mathbf{radius}(D_A), \mathbf{radius}(D_B))$$

and assume without loss of generality that $\mathbf{radius}(D_A) \leq \mathbf{radius}(D_B)$. Let $o = o_A$ denote the center of D_A.

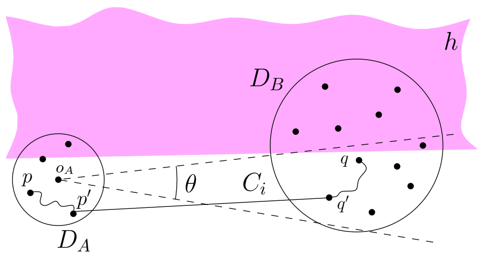


Figure 2.10: Illustrating the proof of Lemma 2.5.5.

Let C_i be the cone with apex o that contains q. Let q' be the point in $B^{(i)} \setminus h$ closest to o. According to Lemma 2.5.4 there is an edge between q' and some point p' in A outside h—see Figure 2.10. By the induction hypothesis, which we may apply because of the monotonicity condition, there are t-paths from p to p' and from q' to q in $\mathcal{G}\ominus h$. By connecting these paths using the edge (p', q') we obtain a path Π in $\mathcal{G}\ominus h$. Next we prove that Π is a t-path between p and q. Set $r := \mathbf{radius}(D_A)$ and $\lambda := \cos\theta - \sin\theta$. Note that $\|xy\|$ denote the (Euclidean) distance between points x and y.

Consider the triangle $\triangle oqq'$. Since $\angle qoq' \leq \theta$, we have

$$\|qq'\| \leq \|oq\| - (\cos\theta - \sin\theta) \cdot \|oq'\|.$$

The total length of Π, denoted $\mathbf{length}(\Pi)$, can now be bounded as follows.

$$\begin{aligned}
\mathbf{length}(\Pi) &\leq t \cdot \|pp'\| + \|p'q'\| + t \cdot \|qq'\| \\
&\leq 2rt + (r + \|oq'\|) + t \cdot (\|oq\| - (\cos\theta - \sin\theta) \cdot \|oq'\|) \\
&= 2rt + (r + \|oq'\|) + t(\|oq\| - r) + tr - t\lambda \cdot \|oq'\| \\
&\leq 3rt + (r + \|oq'\|) + t \cdot \|pq\| - t\lambda \cdot \|oq'\| \\
&= t \cdot \|pq\| + r(3t+1) + (1 - t\lambda) \cdot \|oq'\|.
\end{aligned}$$

Since $d(D_A, D_B) \geq s \cdot r$, we have $\|oq'\| \geq s \cdot r$. Hence, since $\lambda > 1/t$ we get

$$\begin{aligned}
\mathbf{length}(\Pi) &\leq t \cdot \|pq\| + r(3t+1) + sr(1 - t\lambda) \\
&= t \cdot \|pq\|.
\end{aligned}$$

This completes the proof of the lemma. ◻

Now by choosing $\theta = \mathcal{O}(\varepsilon)$ suitably, we can guarantee that $\cos\theta - \sin\theta > 1/(1+\varepsilon)$. This implies $s = \mathcal{O}(1/\varepsilon)$ and leads to the following theorem.

Theorem 2.5.6 *For any set P of n points in the plane and any $\varepsilon > 0$, there exists a $\mathcal{C}$-fault tolerant $(1+\varepsilon)$-spanner of P with $\mathcal{O}((n/\varepsilon^3)\log n)$ edges. The spanner can be constructed in $\mathcal{O}((n/\varepsilon^2)\log^2 n)$ time.*

Proof. By combining Proposition 2.2.1 and Lemmas 2.5.5, the graph constructed by the algorithm is $\mathcal{C}$-fault tolerant. By Lemma 2.5.3 and with the construction algorithm presented below for constructing an SSPD of weight $\mathcal{O}(s^2 n \log n)$, the size of the constructed graph is

$$\begin{aligned}
\sum_{(A,B)\in \text{ SSPD}} |E(A,B)| &= \sum_{(A,B)\in \text{ SSPD}} (|A|/\theta + |B|) \\
&\leq \frac{1}{\theta} \sum_{(A,B)\in \text{ SSPD}} (|A| + |B|) \\
&= \mathcal{O}(\frac{s^2}{\theta} n \log n) \\
&= \mathcal{O}(\frac{1}{\varepsilon^3} n \log n).
\end{aligned}$$

This proves the first part of the theorem. To prove the running time, let (A,B) be an arbitrary pair in the SSPD and assume $\mathbf{radius}(D_A) \leq \mathbf{radius}(D_B)$, where D_A and D_B are two disks containing A and B respectively that satisfy the semi-separated condition.

The first step of the algorithm can be done in $\mathcal{O}(|B|\log|B|)$ time. In the second step, we can compute the convex hull of A in $\mathcal{O}(|A|\log|A|)$ time. For each set $B^{(i)}$, sorting can be done in $\mathcal{O}(|B^{(i)}|\log|B^{(i)}|)$. For every j, to add the non-crossing edges between q_j and the points on CH(A'), it is sufficient to connect q_j to all the points on CH(A') which are between the two tangent lines of CH(A') passing through q_j. Therefore the time we need for adding non-crossing edges is proportional to the number of edges times $\mathcal{O}(\log n)$. Finally we can update CH(A') in $\mathcal{O}(\log m)$ time, where m is the number of points on the convex hull of A', using an online convex hull algorithm—see [PS85, Chapter 3.3.6].

So in total the time for processing the pair (A, B) is bounded by

$$\mathcal{O}(|A|\log|A| + |B|\log|B| + |B|\log(|A|+|B|) + |E(A,B)|\log n).$$

Hence the total running time is

$$\begin{aligned}
&\mathcal{O}\left(\sum_{(A,B)\in\mathcal{W}}\Big(|A|\log|A| + |B|\log|B| + |B|\log(|A|+|B|) + |E(A,B)|\log n\Big)\right)\\
&\leq \mathcal{O}\left(\sum_{(A,B)\in\mathcal{W}}\Big((|A|+|B|)\log n + |E(A,B)|\log n\Big)\right)\\
&= \mathcal{O}(s^2 n\log^2 n + n\log^2 n)\\
&= \mathcal{O}(s^2 n\log^2 n).
\end{aligned}$$

As we will see in the next section, we can compute the SSPD in $\mathcal{O}(s^2 n\log n)$ time, see Lemma 2.5.14, which proves the time complexity of the algorithm. $\square$

2.5.2 Computing an SSPD

To compute an SSPD for a given point set P, we use a BAR-tree, as introduced by Duncan *et al.* [DGK01]. A BAR-tree for a point set P is a BSP-tree with the following properties:

1. each leaf region contains at most one point from P,
2. the tree has size $\mathcal{O}(n)$,
3. if we go down two levels in the tree then the size of the subtree reduces by a factor of β, for some constant $1/2 < \beta < 1$, so its depth is $\mathcal{O}(\log n)$,
4. the region $\mathcal{R}(\nu)$ associated with an (internal or leaf) node ν has aspect ratio at most α for some constant $\alpha > 1$, that is, there are concentric disks $D_I \subset \mathcal{R}(\nu)$ and $D_O \supset \mathcal{R}(\nu)$ with $\mathbf{radius}(D_O) = \alpha \cdot \mathbf{radius}(D_I)$.

Moreover, BAR-trees only use splitting lines that are horizonal, vertical, or diagonal, therefore the complexity of every node (as a polygon) in a BAR-tree is constant.

Let $\mathcal{T}$ be a BAR-tree on the point set P. For a node ν, we use $\text{pa}(\nu)$ to denote the parent of ν, and we use $P(\nu)$ to denote the subset of points from P that are stored in the leaves of the subtree $\mathcal{T}_\nu$ rooted at ν. The *weight* of a node ν is the number of points in $P(\nu)$, and is denoted $|P(\nu)|$. We say that a node ν in $\mathcal{T}$ has *weight class* ℓ, for some integer ℓ, if and only if $|P(\nu)| \leq n/2^\ell$ and $|P(\text{pa}(\nu))| > n/2^\ell$. The weight class of the root is defined to be zero. We denote the collection of nodes of weight class ℓ by $N(\ell)$. Obviously we have $\lfloor \log n \rfloor$ weight classes. Note that some of the nodes in the tree may not be in any weight class; this can happen when the weight of a node ν is almost the same as the weight of its parent. For example, this happens when $|P(\text{pa}(\nu))| = n/2^\ell$ for some ℓ and $|P(\nu)| = n/2^\ell - 1$. It can also happen that a node belongs to more than one weight class, namely when the weight of a node is much smaller than the weight of its parent. The following lemma is straightforward.

Lemma 2.5.7 *Every leaf node is in weight class $\ell_{\max}$, where $\ell_{\max} = \lfloor \log n \rfloor$. Furthermore, on any root-to-leaf-path there is exactly one node with weight class ℓ, for any $0 \leq \ell \leq \ell_{\max}$.*

For a node $\nu \in N(\ell)$, we define its *ℓ-parent* to be the node $\nu' \in N(\ell - 1)$ that is on the path from the root of $\mathcal{T}$ to ν (including ν itself). We denote the ℓ-parent of ν by $\text{pa}(\ell, \nu)$. Observe that ν can be its own ℓ-parent, namely when $\nu \in N(\ell)$ and $\nu \in N(\ell - 1)$. By Lemma 2.5.7, if $\nu \in N(\ell)$ then one of its ancestors (possibly itself) must be in weight class $\ell - 1$, so it must have an ℓ-parent. If μ is the ℓ-parent of ν then we call ν a ℓ-child of μ.

For a node ν in the BAR-tree, the region corresponding to ν is denoted by $\mathcal{R}(\nu)$ and for a region R, we let $\mathbf{diam}(R)$ denote the diameter of the region R. As mentioned before, all nodes in the BAR-tree have bounded aspect ratio, that is, all aspect ratios are bounded by some fixed constant α.

Lemma 2.5.8 *If*

$$d(\mathcal{R}(\nu), \mathcal{R}(\mu)) \geq \frac{(s+1)\alpha}{2} \cdot \min\{\mathbf{diam}(\mathcal{R}(\nu)), \mathbf{diam}(\mathcal{R}(\mu))\}$$

then there are two disks $D_\nu \supset \mathcal{R}(\nu)$ and $D_\mu \supset \mathcal{R}(\mu)$ such that

$$d(D_\nu, D_\mu) \geq s \cdot \min\{\mathbf{radius}(D_\nu), \mathbf{radius}(D_\mu)\}.$$

Proof. Without loss of generality assume $\mathbf{diam}(\mathcal{R}(\nu)) \leq \mathbf{diam}(\mathcal{R}(\mu))$. Let h be a half-plane which contains $\mathcal{R}(\mu)$ such that the distance between h and $\mathcal{R}(\nu)$ is $d(\mathcal{R}(\nu), \mathcal{R}(\mu))$. Note that the half-plane h can be viewed as a disk with infinite

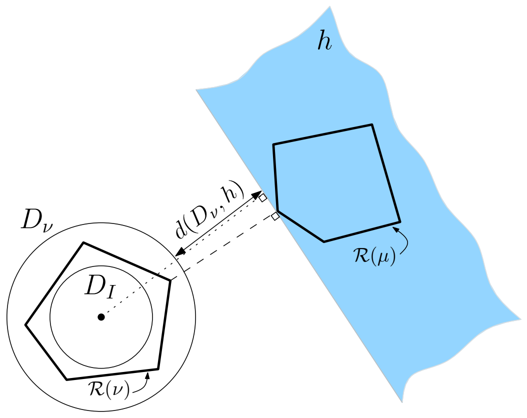


Figure 2.11: Illustrating the proof of Lemma 2.5.8.

radius that contains $\mathcal{R}(\mu)$. Now let D_ν and D_I be two concentric disks such that $D_I \subset \mathcal{R}(\nu) \subset D_\nu$ with $\mathbf{radius}(D_\nu)/\,\mathbf{radius}(D_I) = \alpha$—see Figure 2.11. It is easy to see that

$$2\,\mathbf{radius}(D_I) \le \mathbf{diam}(\mathcal{R}(\nu)) \le 2\,\mathbf{radius}(D_\nu).$$

Then

$$\begin{aligned} d(D_\nu, h) &= d(D_I, h) - (\mathbf{radius}(D_\nu) - \mathbf{radius}(D_I)) \\ &= d(D_I, h) - (\alpha - 1)\,\mathbf{radius}(D_I) \\ &\ge d(D_I, h) - \alpha \cdot \mathbf{radius}(D_I) \\ &\ge d(D_I, h) - \frac{\alpha}{2} \cdot \mathbf{diam}(\mathcal{R}(\nu)) \\ &\ge d(\mathcal{R}(\nu), \mathcal{R}(\mu)) - \frac{\alpha}{2} \cdot \mathbf{diam}(\mathcal{R}(\nu)). \end{aligned}$$

Therefore by setting $D_\mu = h$ we have

$$\begin{aligned} d(D_\nu, D_\mu) &\ge d(\mathcal{R}(\nu), \mathcal{R}(\mu)) - \frac{\alpha}{2} \cdot \mathbf{diam}(\mathcal{R}(\nu)) \\ &\ge \frac{(s+1)\alpha}{2} \cdot \mathbf{diam}(\mathcal{R}(\nu)) - \frac{\alpha}{2} \cdot \mathbf{diam}(\mathcal{R}(\nu)) \\ &\ge \frac{s \cdot \alpha}{2} \cdot \mathbf{diam}(\mathcal{R}(\nu)) \\ &\ge s \cdot \alpha \cdot \mathbf{radius}(D_I) \\ &= s \cdot \mathbf{radius}(D_\nu) \\ &\ge s \cdot \min\{\mathbf{radius}(D_\nu), \mathbf{radius}(D_\mu)\}. \end{aligned}$$

So we are done. ⊔⊓

Now we construct an SSPD $\mathcal{S}$ of the point set P using the following algorithm.

1. Construct a BAR tree $\mathcal{T}$ on P. Let α be the maximum aspect ratio of the region $\mathcal{R}(\nu)$ for any node $\nu \in \mathcal{T}$. Compute the weight classes of all nodes in $\mathcal{T}$.
2. For each weight class ℓ with $0 \leq \ell \leq \ell_{\max}$ do the following: add to $\mathcal{S}$ all pairs $(P(\nu), P(\mu))$ such that
   (i) $\nu, \mu \in N(\ell)$,
   (ii) $d(\mathcal{R}(\nu), \mathcal{R}(\mu)) \geq \frac{(s+1)\alpha}{2} \cdot \min\{\mathbf{diam}(\mathcal{R}(\nu)), \mathbf{diam}(\mathcal{R}(\mu))\}$ and
   (iii) $d(\mathcal{R}(\mathrm{pa}(\ell,\nu)), \mathcal{R}(\mathrm{pa}(\ell,\mu))) < \frac{(s+1)\alpha}{2} \cdot \min\{\mathbf{diam}(\mathcal{R}(\mathrm{pa}(\ell,\nu))), \mathbf{diam}(\mathcal{R}(\mathrm{pa}(\ell,\mu)))\}$.

Lemma 2.5.9 *$\mathcal{S}$ is an SSPD for P with respect to s.*

Proof. By Lemma 2.5.8, all the pairs reported by the algorithm are semi-separated. The only thing that remains to be verified is that for every pair of points p, q there is a unique pair $(P(\nu), P(\mu)) \in \mathcal{S}$ such that $p \in P(\nu)$ and $q \in P(\mu)$, or vice-versa.

For any $0 \leq \ell \leq \ell_{\max}$, define $\nu(p,\ell)$ and $\nu(q,\ell)$ to be the nodes of $N(\ell)$ on the search path to p and q, respectively. Observe that these nodes exist and are uniquely defined by Lemma 2.5.7. We have $\nu(p,0) = \nu(q,0) = \mathrm{root}(\mathcal{T})$, so the sets $P(\nu(p,0))$ and $P(\nu(q,0))$ are the same and therefore not semi-separated. On the other hand, $\nu(p,\ell_{\max})$ and $\nu(q,\ell_{\max})$ are leaves, and so the sets $P(\nu(p,\ell_{\max}))$ and $P(\nu(q,\ell_{\max}))$ are singletons and therefore fulfill condition (ii) of the algorithm. Hence, there must be a value ℓ and two nodes $\nu(p,\ell)$ and $\nu(q,\ell)$ such that they fulfill conditions (ii) and (iii) of the algorithm. The region of any node ν is contained in the region of its parent, which is easily seen to imply that ℓ is unique. $\square$

To bound the weight of the SSPD, we first prove two auxiliary lemmas.

Lemma 2.5.10 *A node ν in $\mathcal{T}$ can be an ℓ-parent of at most a constant number of nodes in $\mathcal{T}$.*

Proof. Consider a node $\nu \in N(\ell-1)$ and let ν' be a node such that $\nu = \mathrm{pa}(\ell, \nu')$. Then ν' is a node in $\mathcal{T}_\nu$ (the subtree of $\mathcal{T}$ rooted at ν) in weight class ℓ. Note that no other node than ν' in $\mathcal{T}_{\nu'}$ can have ν as its ℓ-parent. Recall that the weight of a node reduces by a factor of β when we go down two levels in a BAR-tree. Since $\nu' \in N(\ell)$, its (normal) parent has weight at least $n/2^\ell$. On the other hand $\nu \in N(\ell-1)$, so the weight of ν is at most $n/2^{\ell-1}$. Hence, the path between ν and ν' consists of at most $2k$ links, where $\beta^k = 1/2$. It follows that the total number of nodes in $\mathcal{T}$ that have ν as a ℓ-parent is bounded by 2^{2k}, which is a constant since k is a constant. $\square$

Lemma 2.5.11 *Let $\overline{\mathcal{S}}(\ell)$ be the set of all pairs (ν,μ) such that $\nu,\mu \in N(\ell)$ and $d(\mathcal{R}(\nu),\mathcal{R}(\mu)) < \frac{(s+1)\alpha}{2} \cdot \min\{\mathbf{diam}(\mathcal{R}(\nu)),\mathbf{diam}(\mathcal{R}(\mu))\}$, where $0 \le \ell \le \ell_{\max}$. Then $|\overline{\mathcal{S}}(\ell)| = \mathcal{O}(\alpha^4(s+1)^2 \cdot 2^\ell)$ and*

$$\sum_{(\nu,\mu)\in\overline{\mathcal{S}}(\ell)} (|P(\nu)| + |P(\mu)|) = \mathcal{O}\left(\alpha^4(s+1)^2 \cdot n\right).$$

Proof. We reorder the nodes in the pairs (ν,μ) such that

$$\mathbf{diam}(\mathcal{R}(\nu)) \le \mathbf{diam}(\mathcal{R}(\mu)).$$

We claim that any node ν appears in a constant number of pairs as the first element of the pair. To show this let (ν,μ) be an arbitrary ordered pair. Let $D_{P(\nu)}$ be the smallest enclosing disk of $P(\nu)$ and let o be its center. Consider the annulus A between the disks D_1 and D_2 with center o and radii $r_1 := ((s+1)\alpha+1) \cdot \mathbf{radius}(D_{P(\nu)})$ and $r_2 := r_1 + \mathbf{radius}(D_{P(\nu)})$. Note that

$$\mathbf{diam}(\mathcal{R}(\nu))/2 \le \mathbf{radius}(D_{P(\nu)}) \le \mathbf{diam}(\mathcal{R}(\nu)).$$

Since

$$d(\mathcal{R}(\nu),\mathcal{R}(\mu)) < \frac{(s+1)\alpha}{2} \cdot \mathbf{diam}(\mathcal{R}(\nu))$$

the region $\mathcal{R}(\mu)$ intersect D_1—see Figure 2.12. Now we have two cases:
Case 1: The region $\mathcal{R}(\mu)$ lies partially outside D_2. By the Packing Lemma, [DGK01, Lemma 3.2] this can happen for $\mathcal{O}\left(\alpha^2\left(r_1/(r_2-r_1)\right)\right) = \mathcal{O}(\alpha^3(s+1))$ regions.
Case 2: In this case the region $\mathcal{R}(\mu)$ lies inside D_2. Because the aspect ratio of the region $\mathcal{R}(\mu)$ is at most α, there are two disks D_I and D_O such that $D_I \subset \mathcal{R}(\mu) \subset D_O$ and $\mathbf{area}(D_O) \le \alpha^2 \cdot \mathbf{area}(D_I)$, where $\mathbf{area}(A)$ denotes the area of the region A. Therefore

$$\begin{aligned}
\mathbf{area}(\mathcal{R}(\mu)) &\ge \mathbf{area}(D_I)\\
&\ge \frac{1}{\alpha^2} \cdot \mathbf{area}(D_O)\\
&= \frac{1}{\alpha^2} \cdot \pi\left(\mathbf{radius}(D_O)\right)^2\\
&\ge \frac{1}{4\alpha^2} \cdot \pi\left(\mathbf{diam}(\mathcal{R}(\mu))\right)^2\\
&\ge \frac{1}{4\alpha^2} \cdot \pi\left(\mathbf{diam}(\mathcal{R}(\nu))\right)^2\\
&\ge \frac{\pi r_2^2}{4\alpha^2 \cdot ((s+1)\alpha+2)^2}.
\end{aligned}$$

On the other hand, the area of D_2 is πr_2^2, which means we can have at most $\mathcal{O}(\alpha^4(s+1)^2)$ such regions. Hence in total we can have $\mathcal{O}(\alpha^4(s+1)^2)$ pairs that

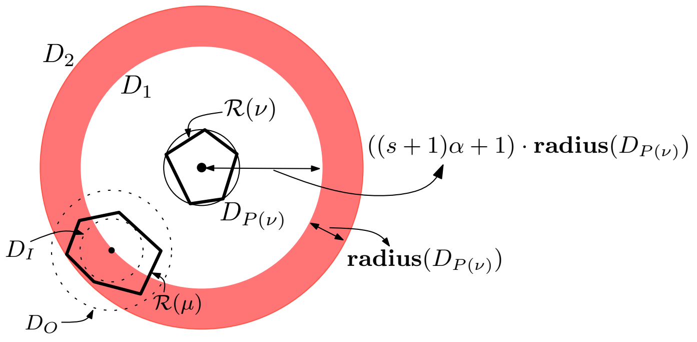

D_2
D_1
$\mathcal{R}(\nu)$
$((s+1)\alpha+1)\cdot \textbf{radius}(D_{P(\nu)})$
$D_{P(\nu)}$
$\textbf{radius}(D_{P(\nu)})$
D_I
$\mathcal{R}(\mu)$
D_O

Figure 2.12: Illustrating the proof of Lemma 2.5.11.

have ν as the first element. Since $|N(\ell)| = \mathcal{O}(2^\ell)$, we can have $\mathcal{O}(2^\ell)$ nodes as the first element of the pair so $|\overline{\mathcal{S}}(\ell)| = \mathcal{O}(\alpha^4(s+1)^2 \cdot 2^\ell)$. The lemma follows since $|P(\nu)| \leq n/2^\ell$ for each $\nu \in N(\ell)$. $\square$

Corollary 2.5.12 *The number of pairs in the SSPD $\mathcal{S}$ generated by the construction algorithm is $\mathcal{O}\left(\alpha^4(s+1)^2 \cdot n\right)$.*

Proof. By the construction algorithm, if $(P(\nu), P(\mu)) \in \mathcal{S}$ and $\nu, \mu \in N(\ell)$ then $(\text{pa}(\ell,\nu), \text{pa}(\ell,\mu)) \in \overline{\mathcal{S}}(\ell-1)$. By combining this with Lemma 2.5.10, we conclude that the number of pairs in $\mathcal{S}$ is bounded by $\mathcal{O}(\sum_{\ell=0}^{\log n} |\overline{\mathcal{S}}(\ell)|)$. Using Lemma 2.5.11 we have

$$\begin{aligned} |\mathcal{S}| &= \mathcal{O}\left(\sum_{\ell=0}^{\log n} |\overline{\mathcal{S}}(\ell)|\right) \\ &= \mathcal{O}\left(\sum_{\ell=0}^{\log n} \alpha^4(s+1)^2 \cdot 2^\ell\right) \\ &= \mathcal{O}\left(\alpha^4(s+1)^2 \cdot n\right). \end{aligned}$$

$\square$

Now we are finally ready to bound the weight of $\mathcal{S}$.

Lemma 2.5.13 *For the SSPD $\mathcal{S}$ generated by the construction algorithm we have*

$$\sum_{(\nu,\mu)\in\mathcal{S}} (|P(\nu)| + |P(\mu)|) = \mathcal{O}\left(\alpha^4(s+1)^2 \cdot n \log n\right).$$

Proof. Since the number of weight classes in $\mathcal{T}$ is $\mathcal{O}(\log n)$ it suffices to prove that for every fixed ℓ it holds that:

$$\sum_{\substack{(\nu,\mu)\in\mathcal{S}\\ \nu,\mu\in N(\ell)}} (|P(\nu)| + |P(\mu)|) = \mathcal{O}(\alpha^4(s+1)^2 n). \tag{2.1}$$

Obviously $|P(\nu)| \leq |P(\text{pa}(\ell,\nu))|$ for each node ν, so we can bound (2.1) by

$$\sum_{\substack{(\nu,\mu)\in\mathcal{S}\\ \nu,\mu\in N(\ell)}} (|P(\text{pa}(\ell,\nu))| + |P(\text{pa}(\ell,\mu))|). \tag{2.2}$$

From the algorithm we know that

$$d(\mathcal{R}(\text{pa}(\ell,\nu)),\mathcal{R}(\text{pa}(\ell,\mu))) < \frac{(s+1)\alpha}{2}\cdot\min\{\mathbf{diam}(\mathcal{R}(\text{pa}(\ell,\nu))),\mathbf{diam}(\mathcal{R}(\text{pa}(\ell,\mu)))\}.$$

Furthermore, by Lemma 2.5.10 each node can be an ℓ-parent of a constant number of nodes. Hence, (2.2) can be bounded by

$$\sum_{(\nu,\mu)\in\overline{\mathcal{S}}(\ell-1)} \mathcal{O}(|P(\nu)| + |P(\mu)|), \tag{2.3}$$

where $\overline{\mathcal{S}}(\ell-1)$ is the set of all pair (ν,μ) such that $\nu,\mu\in N(\ell-1)$ and

$$d(\mathcal{R}(\nu),\mathcal{R}(\mu)) < \frac{(s+1)\alpha}{2}\cdot\min\{\mathbf{diam}(\mathcal{R}(\nu)),\mathbf{diam}(\mathcal{R}(\mu))\}.$$

According to Lemma 2.5.11 summation (2.3) is $\mathcal{O}\left(\alpha^4(s+1)^2\cdot n\right)$, which completes the proof of the lemma. ⊔⊓

Lemma 2.5.14 *The SSPD of a set P of n points with respect to a constant s can be computed in $\mathcal{O}(s^2 n + n\log n)$ time.*

Proof. The BAR tree $\mathcal{T}$ and the weight classes of nodes of $\mathcal{T}$ require $\mathcal{O}(n\log n)$ time to compute [DGK01]. Then we make a tree $\mathcal{T}'$ from $\mathcal{T}$ such that the level of each node in $\mathcal{T}'$ represent its weight class. We do this by making the following changes in $\mathcal{T}$.

1. Remove all the nodes ν with no weight class from the tree and connect the children of ν (if they exist) to the parent of ν.

2. If a node appears in k weight classes ($k > 1$) then repeat the node k times.

By Lemma 2.5.10, each node in the tree $\mathcal{T}'$ has constant degree and also the depth of the tree is $\mathcal{O}(\log n)$.

Now using an algorithm similar to the algorithm for constructing a WSPD, we can construct a SSPD. That is, for each internal node ν of the tree $\mathcal{T}'$, run an algorithm **findpairs**(ν_1, ν_2), where ν_1 and ν_2 are the children of ν. This algorithm tests whether the pair satisfies condition (ii) of the construction described just before Lemma 2.5.9. If they do, it reports the node pair. Otherwise it recurses on the children of ν_1 and ν_2 i.e. for each child μ_1 of ν_1 and each child μ_2 of ν_2 it calls **findpairs**(μ_1, μ_2).

Now the claim is that the running time of the algorithm is $\mathcal{O}(m)$ where m is the size of the SSPD computed by the algorithm. We prove this using a similar argument used for the WSPD [Cal95]. Let the algorithm call **findpairs**(ν_1, ν_2), where ν_1 and ν_2 are the children of a node ν in $\mathcal{T}'$. We define the *computation tree* of (ν_1, ν_2) to be a tree with root at (ν_1, ν_2) which satisfies the following condition. A node (μ_1, μ_2) is a leaf if it satisfies the condition (ii) of the construction described just before Lemma 2.5.9. Otherwise for each child μ_1' of μ_1 and each child μ_2' of μ_2, (μ_1', μ_2') is a child of (μ_1, μ_2). By Lemma 2.5.10, the degree of each node in the computation tree is bounded by a constant. So the time complexity of each call of **findpairs** in the algorithm is linear in the number of nodes in the corresponding computation tree which is linear in the number of leaves in the computation tree. Therefore the total time complexity is linear in the total number of leaves in the computation trees which is the number of pairs that are output.

This proves the lemma since there are $\mathcal{O}(s^2 n)$ pairs in the SSPD according to Corollary 2.5.12. ◻

The following theorem summarizes the results on the SSPD construction.

Theorem 2.5.15 *Given a set P of n points in the plane and $s > 0$ we can compute an SSPD with respect to s of weight $\mathcal{O}(s^2 n \log n)$ in time $\mathcal{O}(s^2 n + n \log n)$.*

2.6 Testing for $\mathcal{C}$-fault tolerance

In the previous sections we constructed fault tolerant spanners. However, an interesting problem is to decide if a given $(1+\varepsilon)$-spanner $\mathcal{G}$ is $\mathcal{C}$-fault tolerant or not. In this section we give an algorithm which in $\mathcal{O}(n^5)$ time using $\mathcal{O}(n^2)$ space can find the answer, where n is the number of vertices of the input graph. Recall that the faults do not contain their boundaries.

Lemma 2.6.1 *Let $\mathcal{G}(P, \mathcal{E})$ be a $(1+\varepsilon)$-spanner on a set P of points in the plane. Let $\mathcal{H}(P)$ be the family of all half-planes in the plane such that the boundary of each half-plane in $\mathcal{H}(P)$ passes through at least two points in P. Then the graph*

$\mathcal{G}$ is a $\mathcal{C}$-fault tolerant $(1+\varepsilon)$-spanner if and only if it is an $\mathcal{H}(P)$-fault tolerant $(1+\varepsilon)$-spanner.

Proof. Obviously a graph is $\mathcal{H}(P)$-fault tolerant if it is $\mathcal{C}$-fault tolerant. To prove the other direction assume $\mathcal{G}$ is $\mathcal{H}(P)$-fault tolerant. By Proposition 2.2.1 it is sufficient to show that the graph $\mathcal{G}$ is an $\mathcal{H}$-fault tolerant $(1+\varepsilon)$-spanner. Let h be an arbitrary half-plane. If the boundary of h passes through at least two vertices of $\mathcal{G}$ then $h \in \mathcal{H}(P)$ and we are done.

Otherwise, if no vertex of $\mathcal{G}$ lies on the boundary of h then we expand h, without changing its boundary direction, until it meets a vertex u of $\mathcal{G}$. It is easy to see that the expansion can affect an edge only after h meets one of the endpoints of the edge and therefore in the expanding process no new edge is affected. If the boundary meets no vertex of $\mathcal{G}$ during the expansion then all the vertices and edges of $\mathcal{G}$ are in h and so we are done.

Let h_u be the new half-plane and assume ℓ_u is the boundary of h_u. Now we rotate the line ℓ_u around u until it meets another vertex of $\mathcal{G}$. Note that by this rotation no new edge is affected. If we h'_u be the new half-plane then $h'_u \in \mathcal{H}(P)$ and by the assumption the graph $\mathcal{G}$ is fault tolerant under h'_u. On the other hand h and h'_u affect the same parts of $\mathcal{G}$ and therefore $\mathcal{G}$ is fault tolerant under h too. ◻

Theorem 2.6.2 *For any $(1+\varepsilon)$-spanner $\mathcal{G}$ on a set P of n points in the plane we can decide whether $\mathcal{G}$ is a $\mathcal{C}$-fault tolerant $(1+\varepsilon)$-spanner or not in $\mathcal{O}(n^5)$ time using $\mathcal{O}(n^2)$ space.*

Proof. Let $\mathcal{H}(P)$ be the family of half-planes such that the boundary of a half-plane $h \in \mathcal{H}(P)$ passes through at least two vertices of $\mathcal{G}$. By Lemma 2.6.1, the graph $\mathcal{G}$ is a $\mathcal{C}$-fault tolerant $(1+\varepsilon)$-spanner if and only if it is an $\mathcal{H}(P)$-fault tolerant $(1+\varepsilon)$-spanner.

Obviously the set $\mathcal{H}(P)$ contains $\mathcal{O}(n^2)$ half-planes. To test a half-plane h, we compute all-pairs shortest paths in $\mathcal{G}\ominus h$ which can be done in $\mathcal{O}(mn + n^2 \log n)$ time using $\mathcal{O}(m)$ space, where m is the number of edges in $\mathcal{G}$. We also can compute all-pairs shortest paths in $\mathcal{G}_{\text{c}}(P)\ominus h$ in $\mathcal{O}(n^3)$ time using $\mathcal{O}(n^2)$ space. Then using a naïve algorithm we can check if $\mathcal{G}\ominus h$ is a $(1+\varepsilon)$-spanner of $\mathcal{G}_{\text{c}}(P)\ominus h$ in $\mathcal{O}(n^2)$ time. So in total the checking procedure takes $\mathcal{O}(n^5)$ time and $\mathcal{O}(n^2)$ space. ◻

2.7 Fault-tolerant geodesic spanners

In this section we consider the problem of constructing fault-tolerant geodesic $(1+\varepsilon)$-spanners for a set P of n points in the plane. Here we require that between any two points $u, v \in P$ outside the region fault F, there is a path in $\mathcal{G}\ominus F$ whose length is at most t times the geodesic distance between u and v in $\mathbb{R}^2 \setminus F$. As remarked in the introduction, finite size fault-tolerant geodesic spanners do not exist unless we are allowed to use Steiner points. As a simple example, consider a set $P = \{p, q\}$ of two points. A spanner $\mathcal{G}$ without Steiner points would have to connect these points by an edge. But this edge can be destroyed by a region fault F, leading to a situation where the distance between p and q in $\mathcal{G} \ominus F$ is non-finite whereas the geodesic distance between p and q in $\mathbb{R}^2 \setminus F$ is finite.

Even if we are allowed to add Steiner points, it is easy to see that finite size fault-tolerant geodesic $(1+\varepsilon)$-spanner do not exist when $\mathcal{F}$ is the family $\mathcal{C}$ of all convex sets. Hence, we restrict the faults to the family $\mathcal{D}$ of disks in the plane.

Our method for constructing a fault-tolerant geodesic spanner works as follows. We first augment P with a set of $4(n-1)$ Steiner points as described in Section 2.2. This way we can get an $\mathcal{O}(n/\varepsilon^2)$ size WSPD consisting of pairs (A, B) where the convex hull of both A and B have at most four vertices. Now fix a pair (A, B). For every pair of points u, v, where u is a vertex of $\mathrm{CH}(A)$ and v is a vertex of $\mathrm{CH}(B)$, we will add a collection of $\mathcal{O}(1/\varepsilon^3)$ Steiner points with $\mathcal{O}(1/\varepsilon^4)$ edges between them, to ensure the following: whenever both u and v are outside the fault disk D, there is a path connecting u and v through those Steiner points and outside D whose length is at most $(1+\varepsilon)$ times the geodesic distance between u and v. This is sufficient because whenever there are points $p \in A$ and $q \in B$ outside D, there are convex hull points $u \in \mathrm{CH}(A)$ and $v \in \mathrm{CH}(B)$ outside D. Hence, we can go from p to u with a short path (by induction), then from u to v (by construction), and then from v to q (by induction).

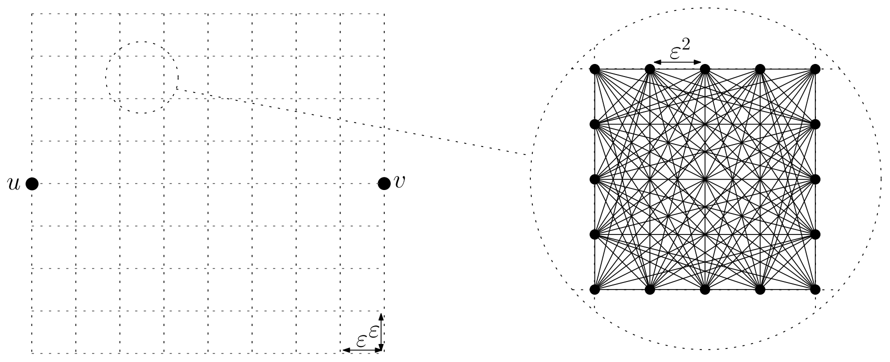


Figure 2.13: The Steiner points added for u, v.

Next we describe how to add Steiner points for the pair u, v. Without loss of generality we assume u and v are on a horizontal line at distance 1. Consider a unit square placed such that uv partitions it into two equal halves. We partition this square into a regular $(1/\varepsilon) \times (1/\varepsilon)$ grid whose cells have size $\varepsilon \times \varepsilon$—we assume for simplicity that $1/\varepsilon$ is an integer—and we put $1/\varepsilon^2$ equally-spaced Steiner points on each grid line, as shown in Figure 2.13. Notice that each grid cell has $1/\varepsilon$ Steiner points on each of its sides, and that we have $\mathcal{O}(1/\varepsilon^3)$ Steiner points in total. For each grid cell we add edges between every pair of Steiner points on its boundary, thus adding $\mathcal{O}(1/\varepsilon^2)$ edges per cell and $\mathcal{O}(1/\varepsilon^4)$ edges in total.

It remains to prove that for every pair of points, say u and v, we can always get a path whose length is close to their geodesic distance when we have a disk fault D. This can be argued as follows.

Assume without loss of generality that the center of D lies below (or on) the line through u and v. Then the geodesic between u and v will go around D on the top side. Let D' be the disk with the same center as D but with radius $r + \sqrt{2}\varepsilon$, where r is the radius of D. Note that D' may contain u or v or both.

Let ∂D and $\partial D'$ denote the boundary of D and D', respectively. Because the diagonal of the grid cells is $\sqrt{2}\varepsilon$ and we have the same distance between D and D' we have:

Observation 2.7.1 *No grid cell can intersect both the interior of D and $\partial D'$.*

The geodesic from u to v consists of a straight line segment connecting u to some point x on ∂D, followed by a circular arc along ∂D from x to some point y, denoted by $\widehat{xy}$, followed by a straight line segment connecting y to v. Draw two rays from o, the common center of D and D', through x and y, and let x' and y' denote the points where these rays intersect $\partial D'$. Next, draw the lines tangent to D' at x' and y'—these are parallel to ux and yv, respectively—and let u' and v' be the intersection of these lines with the vertical lines through u and v—see Figure 2.14. Finally, define Π to be the path consisting of the segments uu' and $u'x'$, followed by the arc along $\partial D'$ from x' to y', followed by the segments $y'v'$ and $v'v$.

From Π we construct a path Π' that uses the edges in our spanner. To this end, let $p_1, p_2, \ldots, p_k$ denote the intersection points of Π with the grid lines, ordered from u to v—see Figure 2.14. Note that $u' = p_1$ and $v' = p_k$. For each intersection point p_i, let p'_i be the closest Steiner point above or on Π on the same grid line. We define Π' to be the path through $p'_1, \ldots, p'_k$. Since each edge in Π' is inside a grid cell intersected by $\partial D'$, by Observation 2.7.1, the path Π' does not intersect D.

It remains to show that Π' approximates the geodesic distance. Firstly we show the length of the path Π, denoted by $\mathbf{length}(\Pi)$, is roughly the same as the geodesic distance between u and v. Recall that we denote the (Euclidean) distance between points x and y by $\|xy\|$ and also we assumed that $\|uv\| = 1$.

Lemma 2.7.2 *If γ is the geodesic distance between u and v then*

$$\mathbf{length}(\Pi) \leq \left(1 + (\pi + 2)\sqrt{2}\varepsilon\right)\gamma.$$

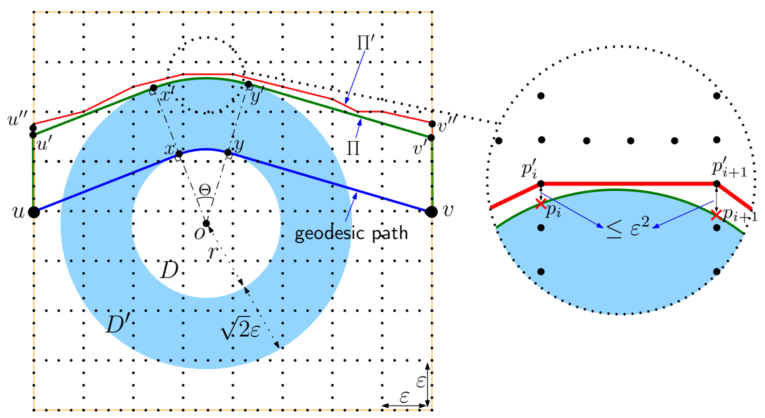


Figure 2.14: The paths Π and Π'.

Proof. From Figure 2.14, it is clear that $\|uu'\| + \|u'x'\| \leq \|ux\| + \sqrt{2}\varepsilon$ and $\|vv'\| + \|v'y'\| \leq \|vy\| + \sqrt{2}\varepsilon$. Let $\Theta = \angle xoy$ be the smaller angle between ox and oy then we have

$$\begin{aligned}
\mathbf{length}(\widehat{x'y'}) &= (r + \sqrt{2}\varepsilon)\Theta \\
&= r\Theta + \sqrt{2}\varepsilon\Theta \\
&= \mathbf{length}(\widehat{xy}) + \sqrt{2}\varepsilon\Theta \\
&\leq \mathbf{length}(\widehat{xy}) + \sqrt{2}\pi\varepsilon.
\end{aligned}$$

Therefore

$$\begin{aligned}
\mathbf{length}(\Pi) &= \|uu'\| + \|u'x'\| + \mathbf{length}(\widehat{x'y'}) + \|y'v'\| + \|v'v\| \\
&\leq \|ux\| + \sqrt{2}\varepsilon + \mathbf{length}(\widehat{xy}) + \sqrt{2}\pi\varepsilon + \|vy\| + \sqrt{2}\varepsilon \\
&\leq \gamma + (2\sqrt{2} + \sqrt{2}\ \pi)\varepsilon \\
&\leq \left(1 + (\pi + 2)\sqrt{2}\varepsilon\right)\gamma. \qquad \text{(since } \gamma \geq 1\text{)}
\end{aligned}$$

Note that in the case when u or v, or both, lie inside D' the same arguments can be used. ⊒

Now we show that Π' approximates Π.

Lemma 2.7.3 *The part of the path Π between u' and v' intersects at most $(2 \cdot \mathbf{length}(\Pi) + 1)/\varepsilon$ grid cells.*

Proof. If the path Π intersects one cell in each column we are done. Otherwise Π intersect at least two cells in the same column. It is easy to see that the length of the part of the path Π which lies inside any two consecutive cells in the column is at least ε. Therefore in total it can intersect at most $(2\cdot\mathbf{length}(\Pi)+1)/\varepsilon$ cells. $\square$

Lemma 2.7.4 $\mathbf{length}(\Pi') \leq (1+6\varepsilon)\,\mathbf{length}(\Pi)$.

Proof. Let ℓ_i be the length of the part of Π inside the ith cell and (p'_i, p'_{i+1}) be the edge on Π' in the same cell—see Figure 2.14. Then obviously $\|p'_i p'_{i+1}\| \leq \ell_i + 2\varepsilon^2$. So

$$\begin{aligned}
\mathbf{length}(\Pi') &= \|up'_1\| + \sum_{i=1}^{k-1} \|p'_i p'_{i+1}\| + \|p'_k v\| \\
&\leq \|uu'\| + \varepsilon^2 + \sum_{i=1}^{k-1} (\ell_i + 2\varepsilon^2) + \|v'v\| + \varepsilon^2 \\
&= \left(\|uu'\| + \sum_{i=1}^{k-1} \ell_i + \|v'v\| \right) + \sum_{i=1}^{k-1} 2\varepsilon^2 + 2\varepsilon^2 \\
&\leq \mathbf{length}(\Pi) + 2k\varepsilon^2 \\
&\leq \mathbf{length}(\Pi) + 2(2\cdot\mathbf{length}(\Pi)+1)\varepsilon \qquad \text{(Lemma 2.7.3)} \\
&\leq (1+6\varepsilon)\,\mathbf{length}(\Pi). \qquad \text{(since } 1 \leq \mathbf{length}(\Pi) \leq 2\text{)}
\end{aligned}$$

$\square$

We obtain the following theorem.

Theorem 2.7.5 *For any set P of n points and any $\varepsilon > 0$, there exists a $\mathcal{D}$-fault tolerant geodesic Steiner $(1+\varepsilon)$-spanner of P with $\mathcal{O}(n/\varepsilon^6)$ edges that uses $\mathcal{O}(n/\varepsilon^5)$ Steiner points.*

Proof. In the procedure of constructing a $\mathcal{D}$-fault tolerant spanner, we added $\mathcal{O}(n)$ Steiner points to have a WSPD of size $\mathcal{O}(n/\varepsilon^2)$ such that the convex hull of each set in the WSPD contains at most four points. Then for each pair (A, B) in the WSPD and for each pair of points on $\mathrm{CH}(A)$ and $\mathrm{CH}(B)$, we added $\mathcal{O}(1/\varepsilon^3)$ Steiner points and $\mathcal{O}(1/\varepsilon^4)$ edges between them. Therefore in total we added $\mathcal{O}(n/\varepsilon^5)$ Steiner points and the graph contains $\mathcal{O}(n/\varepsilon^6)$ edges. $\square$

2.8 Concluding remarks

We introduced the concept of region-fault tolerant spanners for planar point sets, and proved the existence of region-fault tolerant spanners of small size. We showed that for any set of n points in the plane, we can construct a $\mathcal{C}$-fault tolerant spanner of size $\mathcal{O}(n \log n)$ in $\mathcal{O}(n \log^2 n)$ time.

Our spanner construction for arbitrary point sets uses the SSPD, a relaxation of the WSPD. A similar variant with weaker properties, the SSD, was introduced by Varadarajan [Var98] who used it to compute the min-cost perfect matching for points in the plane. His algorithm runs in $\sqrt{n}$ phases where each phase takes time proportional to the weight of the SSD plus the time it takes to compute the SSD. Since our SSPD satisfies the conditions of Varadarajan's SSD, and we can compute it faster, we can improve the running time of the min-cost perfect matching algorithm from $\mathcal{O}(n^{3/2} \log^5 n)$ to $\mathcal{O}(n^{3/2} \log^2 n)$.

The main open problem is to determine whether the spanner size for arbitrary point sets can be improved to $\mathcal{O}(n)$.

Another interesting question is how we can improve the $\mathcal{C}$-fault tolerant checking algorithm. Finally, in our approach, we used straight line segments as edges in our spanners. For geodesic spanners, it is interesting to see what happens if we are allowed to add curved edges.

Chapter 3

Dilation-Optimal Edge Augmentation

3.1 Introduction

After designing a network, in addition to performing network maintenance, it is sometimes desirable to improve its quality. In this chapter we consider the problem when one is given a (geometric) network and the problem at hand is to extend the network with an additional edge while minimizing the dilation of the resulting graph. More formally the problem is as follows. Given an edge weighted graph $G(V, E)$ with n vertices and m edges, construct a graph G' by adding an edge to G such that the dilation of G' is minimized. The problem was first stated by Narasimhan [Nar02] and, surprisingly, it has not been studied earlier, to the best of our knowledge.

The results presented in this chapter are summarized in Table 3.1. Note that some of the presented bounds hold for any graph with positive edge weights (*weighted graphs*) while some only hold for Euclidean graphs. Recall from the introduction that we assume that the vertex set of any graph is a subset of a metric space and for each pair of vertices u and v in the graph, $\mathbf{d}(u, v)$ denotes the distance between u and v in the metric space. Throughout this chapter $\mathcal{G}$ denotes a Euclidean graph while G denotes a graph with positive edge weights.

Throughout this chapter we will use G_{opt} to denote the optimal solution, while t_{opt} and t denote the dilation of G_{opt} and the input graph G respectively. Also for each pair of points $u, v \in V$, $\delta_G(u, v)$ denotes the shortest path between u and v

Input graph	Apx. factor	Time complexity	Space	Section
Weighted graph	1	$\mathcal{O}(n^3m + n^4 \log n)$	$\mathcal{O}(m)$	Section 3.2.1
Weighted graph	1	$\mathcal{O}(n^4)$	$\mathcal{O}(n^2)$	Section 3.2.1
Euclidean graph	$1+\varepsilon$	$\mathcal{O}(n^3/\varepsilon^d)$	$\mathcal{O}(n^2)$	Section 3.2.2
Weighted graph	3	$\mathcal{O}(nm + n^2 \log n)$	$\mathcal{O}(m)$	Section 3.3
Euclidean graph	$2+\varepsilon$	$\mathcal{O}(nm + n^2(\log n + \frac{1}{\varepsilon^{3d}}))$	$\mathcal{O}(n^2)$	Section 3.4
Euclidean t-spanner	$1+\varepsilon$	$\mathcal{O}((\frac{t^7}{\varepsilon^4})^d \cdot n^2)$	$\mathcal{O}(n \log(tn))$	Section 3.5

Table 3.1: Complexity bounds for the algorithms presented in this chapter.

in graph G and $\mathbf{d}_G(u,v)$ denotes the length of $\delta_G(u,v)$.

3.2 Three simple algorithms

We consider the problem of computing an optimal solution G_{opt}. That is, we are given a graph $G = (V, E)$, and the aim is to compute a t_{opt}-spanner $G_{opt} = (V, E \cup \{e\})$.

In this section we present two simple exact algorithms which work for any weighted graph and one fast approximation algorithm for geometric graphs.

3.2.1 Exact algorithms

A naïve approach to decide which edge to add is to test every possible candidate edge. The number of such edges is obviously $\left(\frac{n(n-1)}{2} - m\right) = \mathcal{O}(n^2)$. Testing a candidate edge e entails computing the dilation of the graph $G' = (V, E \cup \{e\})$, denoted the candidate graph or augmented graph. Therefore we briefly consider the problem of computing the dilation of a given graph with positive edge weights. This problem has recently received considerable attention, see for example [EW07, AKK$^+$, NS00].

A trivial upper bound for computing the dilation of G' is obtained by computing the length of the shortest paths between every pair of vertices in G'. This can be done by running Dijkstra's algorithm—implemented using Fibonacci heaps—n times, resulting in an $\mathcal{O}(mn + n^2 \log n)$ time algorithm using $\mathcal{O}(m)$ space. This approach is quite slow and we would like to be able to compute the dilation more efficiently, but no faster algorithm is known for any graphs except planar graphs, paths, cycles, stars and trees [EW07, AKK$^+$, NS00].

Applying the stated bound gives that G_{opt} can be computed in time $\mathcal{O}(n^3(m + n\log n))$ using $\mathcal{O}(m)$ space.

A small improvement can be obtained by observing that when an edge (u, v) is about to be tested, we do not have to do all-pairs-shortest path to compute the dilation of the augmented graph. Instead, for each pair of vertices $x, y \in V$, it suffices to check whether there is a shorter path between x and y using the edge (u, v). That is, we only have to compute $\mathbf{d}_G(x, u) + \mathbf{d}(u, v) + \mathbf{d}_G(v, y)$, $\mathbf{d}_G(x, v) + \mathbf{d}(v, u) + \mathbf{d}_G(u, y)$ and $\mathbf{d}_G(x, y)$, which can be done in constant time since the length of a shortest path between every pair of vertices in G has already been computed (provided that we store this information). So if, as a preprocessing, one computes all-pairs-shortest paths of the input graph G and saves the result in a distance matrix M, then the dilation of the augmented graph can be computed in $\mathcal{O}(n^2)$ time. We use Dilation(G, M, x, y) to denote the procedure that, given the matrix M and an edge (x, y), computes the dilation of the graph $G \cup (x, y)$.

See the details of the exact algorithm in Algorithm 3.2.1.

```
Algorithm 3.2.1: ExpandGraph
  Input: Weighted graph $G = (V, E)$.
  Output: Weighted graph $G' = (V, E \cup \{e\})$.
1 $M$ := distance matrix of $G$ ;
2 $t := \infty$;
3 foreach $(x, y) \in V^2 \setminus E$ do
4     $t' :=$ Dilation$(G, M, x, y)$;
5     if $t' < t$ then
6         $t := t'$ and $e := (x, y)$
7     end
8 end
9 return $G' = (V, E \cup \{e\})$;
```

Hence, we have the following result:

Lemma 3.2.1 *Given a graph G with positive edge weights, an optimal solution G_{opt} can be computed in time $\mathcal{O}(n^4)$ using $\mathcal{O}(n^2)$ space.*

Proof. Solving the all-pair-shortest-path problem requires cubic time and all the distances are stored in an $n{\times}n$ matrix. In total $\mathcal{O}(n^2)$ edges are tested for insertion. For each candidate edge we compute the length of the shortest path between every pair of points in G, which can be done in constant time per candidate as described above. ⊔⊓

3.2.2 A $(1+\varepsilon)$-approximation for Euclidean graphs

In the previous section we showed that an optimal solution can be obtained by testing a quadratic number of candidate edges. Testing each candidate edge entails $\mathcal{O}(n^2)$ distance queries, where a distance query asks for the length of a shortest path in the graph between two query points. One way to speed up the computation is to compute an approximate dilation. The problem of computing an approximate dilation of a geometric graph was considered by Narasimhan and Smid in [NS00]. They showed the following fact, which states that the WSPD of the vertex set of any Euclidean graph $\mathcal{G}$ can be used to approximate the dilation of $\mathcal{G}$—see also Section 13.2 of [NS07].

Fact 3.2.2 *Let V be a set of n points in $\mathbb{R}^d$ and let $\{(A_i, B_i)\}_{i=1}^m$ be a WSPD for V with respect to separation constant $s := 4(2+\varepsilon)/\varepsilon$. For each j with $1 \le j \le m$, let a_j be an arbitrary point in A_j, and let b_j be an arbitrary point in B_j. For any connected Euclidean graph $\mathcal{G}$ with vertex set V, the following holds: For each j with $1 \le j \le m$, let $\mathfrak{D}_{\mathcal{G}}(a_j, b_j)$ be the dilation between a_j and b_j in $\mathcal{G}$, and let*

$$t := \max_{1\le j\le m} \mathfrak{D}_{\mathcal{G}}(a_j, b_j).$$

Then $\mathfrak{D}_{\mathcal{G}}/(1+\varepsilon) \le t \le \mathfrak{D}_{\mathcal{G}}$, or equivalently $t \le \mathfrak{D}_{\mathcal{G}} \le (1+\varepsilon)t$, where $\mathfrak{D}_{\mathcal{G}}$ denotes the dilation of $\mathcal{G}$.

Thus, in order to approximate the dilation of a Euclidean graph, it is sufficient to compute the dilation between $\mathcal{O}(n/\varepsilon^d)$ pairs of vertices. Moreover, the choice of these vertices depends only on the vertex set of the graph, it does not depend on the edges of the graph. As a result the time to compute the dilation decreases from $\mathcal{O}(n^2)$ to $\mathcal{O}(n/\varepsilon^d)$, thus the total running time decreases from $\mathcal{O}(n^4)$ to $\mathcal{O}(n^3/\varepsilon^d)$.

Theorem 3.2.3 *Given a Euclidean graph $\mathcal{G} = (V, E)$ and a real constant $\varepsilon > 0$, one can in $\mathcal{O}(n^3/\varepsilon^d)$ time, using $\mathcal{O}(n^2)$ space, compute a $t^\star$-spanner $\mathcal{G}' = (V, E \cup \{e\})$ such that $t_{opt} \le t^\star \le (1+\varepsilon)\cdot t_{opt}$.*

Proof. The time bound follows from the above discussion. Obviously the dilation of $\mathcal{G}'$, denoted by $t^\star$, reported by the algorithm is at least t_{opt}. It remains to prove that $t^\star$ is bounded by $(1+\varepsilon)\cdot t_{opt}$.

Assume that t is the approximate dilation of $\mathcal{G}'$ and for each candidate graph $\mathcal{G}_i$, let t_i be its approximate dilation as computed by the algorithm and let $\mathfrak{D}_{\mathcal{G}_i}$ be its exact dilation. Based on the algorithm $t = \min_i t_i$ and from Fact 3.2.2 it follows that for each candidate graph $\mathcal{G}_i$, $t_i \le \mathfrak{D}_{\mathcal{G}_i} \le (1+\varepsilon)\cdot t_i$. Assume that $t_{opt} = \mathfrak{D}_{\mathcal{G}_j}$. Therefore

$$t^\star \le (1+\varepsilon)\cdot t \le (1+\varepsilon)\cdot t_j \le (1+\varepsilon)\cdot \mathfrak{D}_{\mathcal{G}_j} = (1+\varepsilon)\cdot t_{opt},$$

which complete the proof. $\square$

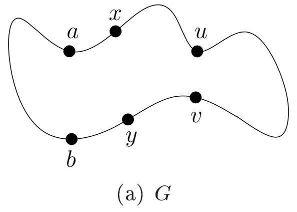
3.3 Adding a bottleneck edge

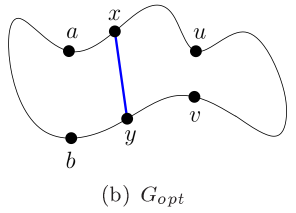
Consider a graph $G = (V, E)$ with positive edge weights and dilation t. In this section we analyze the following simple algorithm: Add an edge between a pair of vertices in G with dilation t. This edge is called a *bottleneck edge* of G. We show that this gives us a 3-approximation of the optimal solution.

Let $G_{\mathcal{B}}$ be a graph obtained from G by adding a bottleneck edge, and let $t_{\mathcal{B}}$ be the dilation of $G_{\mathcal{B}}$. Note that $G_{\mathcal{B}}$ can be computed in the same time as the dilation of G can be decided, i.e., in $\mathcal{O}(mn+n^2 \log n)$ time for graphs with positive edge weights.

Lemma 3.3.1 *Given a graph G with positive edge weights it holds that $t_{\mathcal{B}} < 3t_{opt}$.*

Proof. Recall that t denotes the dilation of G and that G_{opt} denotes the optimal graph. Let (x, y) be the edge added to G to obtain G_{opt}, and let (u, v) be the edge added to G to obtain $G_{\mathcal{B}}$, i.e., (u, v) is a bottleneck edge of G, as illustrated in Figure 3.1.





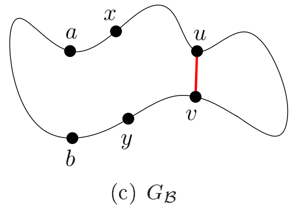


Figure 3.1: (x, y) is the optimal edge added to G and (u, v) is a bottleneck edge of G.

First note that if $t_{opt} > t/3$ then the lemma holds and we are done. Thus we may assume that $t_{opt} \leq t/3$. The proof of the lemma is done by considering a pair of vertices, denoted (a, b), that are endpoints of a bottleneck edge of $G_{\mathcal{B}}$. Fix a path $\delta_{G_{opt}}(a, b)$. If this path does not include the edge (x, y) then

$$\mathbf{d}_{G_{opt}}(a, b) = \mathbf{d}_G(a, b) \geq \mathbf{d}_{G_{\mathcal{B}}}(a, b)$$

and we are done. Therefore, we may assume that the path $\delta_{G_{opt}}(a, b)$ includes (x, y). Also, we will assume without loss of generality that a shortest path in G_{opt} from a to b goes from a to x and then to b via y, otherwise the labels a and b may be switched. Note that $\delta_{G_{opt}}(u, v)$ must pass through (x, y) otherwise we have

$$t_{opt} \geq \mathbf{d}_{G_{opt}}(u, v) / \mathbf{d}(u, v) = \mathbf{d}_G(u, v) / \mathbf{d}(u, v) = t$$

which means that $t = t_{opt}$, which contradicts the assumption that $t_{opt} \leq t/3$. Furthermore, we assume that a shortest path in G_{opt} from u to v goes from u to x and then to v via y, otherwise the labels u and v may be switched.

As a first step we bound the distance between the endpoints of the bottleneck edge u and v. This is done by bounding the length of the path in G between x and y as follows, see Figure 3.1.

$$\begin{aligned}
\mathbf{d}_G(u,v) &\leq \mathbf{d}_{G_{opt}}(u,v) - \mathbf{d}(x,y) + \mathbf{d}_G(x,y) \\
&\leq t_{opt} \cdot \mathbf{d}(u,v) - \mathbf{d}(x,y) + t \cdot \mathbf{d}(x,y) \\
&\leq \frac{t}{3} \cdot \mathbf{d}(u,v) - \mathbf{d}(x,y) + t \cdot \mathbf{d}(x,y) \\
&< \frac{t}{3} \cdot \mathbf{d}(u,v) + t \cdot \mathbf{d}(x,y).
\end{aligned}$$

Since $\mathbf{d}_G(u,v) = t \cdot \mathbf{d}(u,v)$ it follows that

$$\mathbf{d}(u,v) < 3/2 \cdot \mathbf{d}(x,y). \tag{3.1}$$

Also,

$$\begin{aligned}
t \cdot \mathbf{d}(u,v) &= \mathbf{d}_G(u,v) \\
&\leq \mathbf{d}_G(u,a) + \mathbf{d}_G(a,b) + \mathbf{d}_G(b,v) \\
&\leq \mathbf{d}_G(u,a) + t \cdot \mathbf{d}(a,b) + \mathbf{d}_G(b,v)
\end{aligned}$$

which implies that

$$t \cdot (\mathbf{d}(u,v) - \mathbf{d}(a,b)) \leq \mathbf{d}_G(u,a) + \mathbf{d}_G(b,v), \tag{3.2}$$

and

$$\begin{aligned}
\mathbf{d}_G(a,u) + 2\,\mathbf{d}(x,y) + \mathbf{d}_G(v,b) &\leq \mathbf{d}_G(a,x) + \mathbf{d}_G(x,u) + 2\,\mathbf{d}(x,y) \\
&\qquad + \mathbf{d}_G(v,y) + \mathbf{d}_G(y,b) \\
&= \mathbf{d}_{G_{opt}}(a,b) + \mathbf{d}_{G_{opt}}(u,v) \\
&\leq t_{opt}(\mathbf{d}(a,b) + \mathbf{d}(u,v)), \qquad (3.3)
\end{aligned}$$

which gives that

$$\mathbf{d}_G(a,u) + \mathbf{d}_G(v,b) \leq t_{opt}(\mathbf{d}(a,b) + \mathbf{d}(u,v)) - 2\,\mathbf{d}(x,y). \tag{3.4}$$

By putting together (3.2) and (3.4) we have

$$\begin{aligned}
t(\mathbf{d}(u,v) - \mathbf{d}(a,b)) &\leq \mathbf{d}_G(a,u) + \mathbf{d}_G(v,b) \\
&\leq t_{opt}(\mathbf{d}(a,b) + \mathbf{d}(u,v)) - 2\,\mathbf{d}(x,y) \\
&< t_{opt}(\mathbf{d}(a,b) + \mathbf{d}(u,v)),
\end{aligned}$$

which implies that

$$\mathbf{d}(a,b)(t_{opt} + t) > \mathbf{d}(u,v)(t - t_{opt})$$

and

$$\mathbf{d}(a,b) > \frac{t - t_{opt}}{t_{opt} + t} \cdot \mathbf{d}(u,v) > \frac{t - \frac{t}{3}}{\frac{t}{3} + t} \cdot \mathbf{d}(u,v) = \frac{1}{2} \cdot \mathbf{d}(u,v). \tag{3.5}$$

Now we are ready to put together the results:

$$\begin{aligned}
t_{\mathcal{B}} \cdot \mathbf{d}(a,b) &= \mathbf{d}_{G_{\mathcal{B}}}(a,b) \\
&\leq \mathbf{d}_G(a,u) + \mathbf{d}(u,v) + \mathbf{d}_G(v,b) \\
&< \mathbf{d}_G(a,u) + \frac{3}{2}\mathbf{d}(x,y) + \mathbf{d}_G(v,b) && \text{(from (3.1))} \\
&< \mathbf{d}_G(a,u) + 2\,\mathbf{d}(x,y) + \mathbf{d}_G(v,b) \\
&\leq t_{opt}\,(\mathbf{d}(a,b) + \mathbf{d}(u,v)) && \text{(from (3.3))} \\
&< 3t_{opt} \cdot \mathbf{d}(a,b). && \text{(from (3.5))}
\end{aligned}$$

This completes the proof of the lemma since $t_{\mathcal{B}} < 3t_{opt}$. ⊒

We conclude by stating the main result of this section followed by a lower bound for the bottleneck approach.

Theorem 3.3.2 *Given a graph $G = (V, E)$ with positive edge weights, a $t_{\mathcal{B}}$-spanner $G' = (V, E \cup \{e\})$ with $t_{\mathcal{B}} < 3t_{opt}$ can be computed in $\mathcal{O}(mn + n^2 \log n)$ time using $\mathcal{O}(m)$ space.*

The following observation by Grüne [Grü05] shows that the upper bound stated in Lemma 3.3.1 is tight, even for Euclidean graphs.

Observation 3.3.3 *([Grü05]) For for any $0 < \varepsilon < 2$, there exists a Euclidean graph $\mathcal{G}$ such that $(3 - \varepsilon) \cdot t_{opt} \leq t_{\mathcal{B}}$.*

Proof. Choose the value of α such that $\sin(\frac{\alpha}{2}) = \varepsilon' = \varepsilon/(4 - \varepsilon)$. We construct a graph $\mathcal{G}$ which includes seven points $p_1, \ldots, p_7$ along two line segments (p_1, p_4) and (p_4, p_7), where the angle between the two segments is α. Put two point p_2 and p_3 on segment (p_1, p_4) such that $\|p_1p_2\| = d$, $\|p_2p_3\| = \ell$ and $\|p_3p_4\| = 1$, where

$$d := \frac{2(1 - \varepsilon')}{(1 + \varepsilon')(3 - \varepsilon')} \quad \text{and} \quad \ell := \frac{(1 - \varepsilon')^2}{(1 + \varepsilon')(3 - \varepsilon')}.$$

Also we put points p_5 and p_6 on segment (p_4, p_7) with similar interdistances—see Figure 3.2(a).

Now consider the graph $\mathcal{G}$ on $V := \{p_1, \ldots, p_7\}$ with edge set

$$E := \{(p_i, p_{i+1}), i = 1, \ldots, 6\}.$$

We have the following observations.

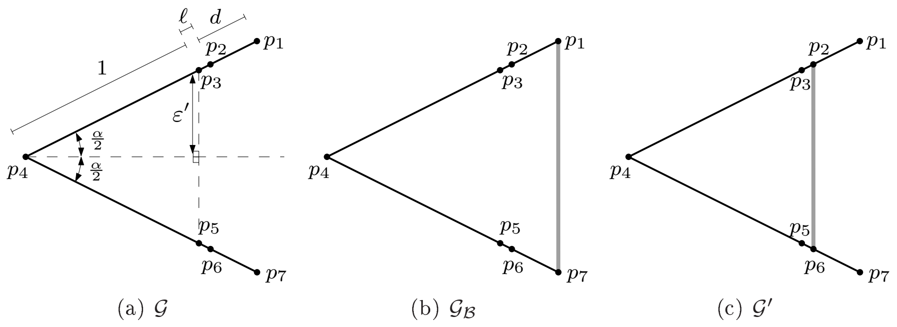


Figure 3.2: The graph $\mathcal{G}$, where $t_{\mathcal{B}}/t_{\mathcal{P}}$ gets arbitrarily close to 3.

1. The dilation of $\mathcal{G}$ is $1/\varepsilon'$ which is attained by pairs (p_1, p_7), (p_2, p_6) and (p_3, p_5). So we can choose (p_1, p_7) as a bottleneck edge. Note that by moving p_2 and p_3 a little bit the dilation between p_2 and p_6 and the dilation between p_3 and p_5 will decrease and (p_1, p_7) will be the only bottleneck edge.

2. If we insert the bottleneck edge (p_1, p_7) in $\mathcal{G}$, see Figure 3.2(b), the dilation of $\mathcal{G}_{\mathcal{B}}$ does not change because the length of the new path connecting p_3 and p_5 passing through p_1 and p_7 is equal to

$$\begin{aligned} 2\left(\ell + d + (1+\ell+d)\sin(\alpha/2)\right) &= 2\left(\frac{1-\varepsilon'}{1+\varepsilon'} + \left(1 + \frac{1-\varepsilon'}{1+\varepsilon'}\right)\cdot\varepsilon'\right) \\ &= 2 \end{aligned}$$

which is the same as $\mathbf{d}_{\mathcal{G}}(p_3, p_5)$. So $t_{\mathcal{G}} = t_{\mathcal{B}} = 1/\varepsilon'$.

3. If $\mathcal{G}'$ attained by adding the edge (p_2, p_6) to the graph $\mathcal{G}$, see Figure 3.2(c), then the dilation of $\mathcal{G}'$ is taken by the pairs (p_1, p_7) and (p_3, p_5) which is equal to

$$t_{\mathcal{G}'} = \frac{\ell + (1+\ell)\varepsilon'}{\varepsilon'} = \frac{d + (1+\ell)\varepsilon'}{(1+\ell+d)\varepsilon'} = \frac{\varepsilon' + 1}{\varepsilon'(3-\varepsilon')}.$$

Since $t_{opt} \leq t_{\mathcal{G}'}$ we have

$$\begin{aligned} \frac{t_{\mathcal{B}}}{t_{opt}} &\geq \frac{t_{\mathcal{B}}}{t_{\mathcal{G}'}} \\ &= \frac{1/\varepsilon'}{(\varepsilon'+1)/(\varepsilon'(3-\varepsilon'))} \\ &= \frac{3-\varepsilon'}{\varepsilon'+1} \\ &= 3 - \varepsilon \qquad\qquad (\text{since } \varepsilon' = \frac{\varepsilon}{4-\varepsilon}) \end{aligned}$$

which implies the observation. ⊔⊓

3.4 A $(2 + \varepsilon)$-approximation for Euclidean graphs

In the remainder of the chapter we will develop approximation algorithms for Euclidean graphs. In this section we present a fast approximation algorithm which guarantees an approximation factor of $(2+\varepsilon)$. The algorithm is similar to the algorithms presented in Section 3.2 in the sense that it tests candidate edges. Testing a candidate edge entails computing the dilation of the input graph augmented with the candidate edge. The main difference is that we will show, in Section 3.4.1, that only a linear number of candidate edges need to be tested to obtain a solution that gives a $(2 + \varepsilon)$-approximation, instead of a quadratic number of edges.

Moreover, in Section 3.4.2 we show that the same approximation bound can be achieved by performing only a linear number of shortest path queries for each candidate edge. The candidate edges are selected by using the well-separated pair decomposition—see Section 2.3.

3.4.1 Linear number of candidate edges

In this section we show how to obtain a $(2 + \varepsilon)$-approximation in cubic time. As mentioned above, the algorithm is similar to the algorithm presented in Section 3.2 in the sense that it tests candidate edges. Here we will show that only a linear number of candidate edges is needed to be tested to obtain a solution that gives a $(2 + \varepsilon)$-approximation.

The approach is straight-forward, see Algorithm 3.4.1. First the algorithm computes the length of the shortest path in $\mathcal{G}$ between every pair of points in V. The distances are saved in a matrix M. Next, the well-separated pair decomposition of V is computed. Note that, in line 5, the candidate edges will be chosen using the well-separated pair decomposition. In line 6, the function Dilation returns the dilation of the graph $\mathcal{G}_i(V, E \cup \{(a_i, b_i)\})$, i.e., in lines 5–8, a candidate edge is tested by computing the dilation of $\mathcal{G}$ with the candidate edge (a_i, b_i) added to $\mathcal{G}$. Recall that Dilation uses the distance matric M to answer such a query in quadratic time. Next, we bound the running time of the approximation algorithm and then prove the approximation bound.

Lemma 3.4.1 *Algorithm 3.4.1 requires $\mathcal{O}(n^3/\varepsilon^{2d})$ time and $\mathcal{O}(n^2)$ space.*

Proof. The complexity of all lines of the algorithm is straight-forward to calculate. Recall that line 1 requires $\mathcal{O}(mn + n^2 \log n)$ time and quadratic space, and line 2 requires $\mathcal{O}(n/\varepsilon^{2d} + n \log n)$ time according to Section 2.3. Because $k = \mathcal{O}(n/\varepsilon^{2d})$, the time needed for lines 4–10 is $\mathcal{O}(n/\varepsilon^{2d})$ times the complexity of Dilation which is $\mathcal{O}(n^2)$. Thus summing up we get $\mathcal{O}(\frac{n}{\varepsilon^{2d}} \cdot n^2)$, as stated in the lemma. ⊔⊓

Algorithm 3.4.1: Approximate ExpandGraph

```
Input: Euclidean graph $\mathcal{G} = (V, E)$ and a real constant $\varepsilon > 0$.
Output: Euclidean graph $\mathcal{G}' = (V, E \cup \{e\})$.
1  $M :=$ distance matrix of $\mathcal{G}$ ;
2  $\{(A_i, B_i)\}_{i=1}^{k} :=$ WSPD of the point set $V$ with respect to $s = 256/\varepsilon^2$;
3  $t := \infty$;
4  for $i := 1, 2, \ldots, k$ do
5      Select arbitrary points $a_i \in A_i$ and $b_i \in B_i$;
6      $t_i :=$ Dilation$(\mathcal{G}_i, M, a_i, b_i)$;
7      if $t_i < t$ then
8          $t := t_i$ and $e := (a_i, b_i)$
9      end
10 end
11 return $\mathcal{G}' = (V, E \cup \{e\})$;
```

It remains to analyze the quality of the solution obtained from algorithm 3.4.1. We need to compare the graph resulting from adding an optimal edge to $\mathcal{G}$ and the graph $\mathcal{G}'$ resulting from Approximate ExpandGraph. Let $e = (a, b)$ be an optimal edge and let (A_i, B_i) be the well-separated pair such that $a \in A_i$ and $b \in B_i$. At first sight, it seems that the edge (a_i, b_i) tested by the algorithm should be a good candidate. However, the separation constant of our well-separated pair decomposition only depends on ε which implies that the shortest path between a and a_i, and between b and b_i could be very long compared to the distance between a and b. In Lemma 3.4.2, we show the existence of a "short" edge e' that is a good approximation of the optimal edge and then, in Lemma 3.4.3, we show that Approximate ExpandGraph computes a good approximation of e'.

Let $\Delta_{\mathcal{G}}(p, q)$ denote the set of point pairs (u, v) in V such that

$$(p, q) \in \delta_{\mathcal{G} \cup \{(p,q)\}}(u, v).$$

That is, $\Delta_{\mathcal{G}}(p, q)$ is the set of point pairs for which a shortest path between them in $\mathcal{G} \cup \{(p, q)\}$ passes through (p, q). When it is clear which graph we are talking about, we use $\Delta(p, q)$ instead of $\Delta_{\mathcal{G}}(p, q)$.

Lemma 3.4.2 *For any given constant $0 < \lambda \leq 1$, there exists a pair $p, q \in V$ of the graph $\mathcal{G}$ such that for every pair $(u, v) \in \Delta(p, q)$ it holds that $\|uv\| \geq \frac{\lambda}{2}\|pq\|$, and the dilation of $\mathcal{G} \cup \{(p, q)\}$ is bounded by $(2 + \lambda) \cdot t_{opt}$.*

Proof. The proof is done in two steps. First a point pair $p_j, q_j \in V$ is selected that fulfills the first requirement of the lemma. Then, we prove the second requirement, i.e., the dilation of $\mathcal{G} \cup \{(p_j, q_j)\}$ is bounded by $(2 + \lambda) \cdot t_{opt}$.

Consider an optimal solution $\mathcal{G}_1 = \mathcal{G} \cup \{(p_1, q_1)\}$, with dilation $t_1 = t_{opt}$. If $\|uv\| \geq \frac{\lambda}{2} \cdot \|p_1 q_1\|$ for every point pair $(u, v) \in \Delta(p_1, q_1)$ then $j = 1$, i.e., we have found the point pair we are searching for. Otherwise, let $e_2 = (p_2, q_2)$ denote the closest pair in $\Delta(p_1, q_1)$, and continue the search for (p_j, q_j). See Figure 3.3 for an illustration. Note that there exists a point pair $(u, v) \in \Delta(p_1, q_1)$ such that $\|uv\| < \frac{\lambda}{2} \cdot \|p_1q_1\|$, otherwise $j = 1$. And since $\|p_2q_2\| \leq \|uv\|$ for every $(u, v) \in \Delta(p_1, q_1)$ we have $\|p_2q_2\| < \frac{\lambda}{2} \cdot \|p_1q_1\|$.

We define e_3 in a similar way, that is, if for each point pair $(u, v) \in \Delta(p_2, q_2)$ it holds that $\|uv\| \geq \frac{\lambda}{2} \cdot \|p_2q_2\|$ then we have found the point pair $(p_j = p_2, q_j = q_2)$ that we are searching for. Otherwise, let $e_3 = (p_3, q_3)$ denote the closest pair in $\Delta(p_2, q_2)$.

We continue to define the vertex pairs $e_4, \ldots, e_j$ and the corresponding graphs $\mathcal{G}_4, \ldots, \mathcal{G}_j$ until we find a point pair (p_j, q_j) for which there is no vertex pair e_{j+1} such that $(p_{j+1}, q_{j+1}) \in \Delta(p_j, q_j)$ and $\|p_{j+1}q_{j+1}\| < \frac{\lambda}{2} \cdot \|p_jq_j\|$. Based on the construction we have one basic property: for each i,

$$\|p_{i+1}q_{i+1}\| < \frac{\lambda}{2} \cdot \|p_iq_i\|.$$

This complete the first part of the proof.

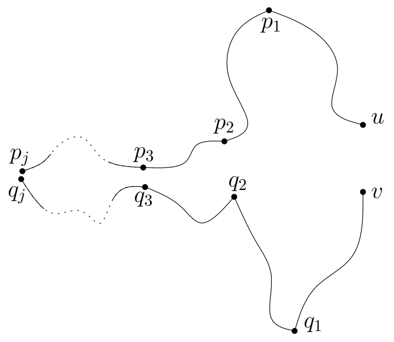


Figure 3.3: Illustrating the proof of Lemma 3.4.2.

For the second part, we claim that $\mathcal{G}_j$ has dilation at most $(2+\lambda) \cdot t_{opt}$. Before we continue we need to prove:

$$\mathbf{d}_{\mathcal{G}_i}(p_{i+1}, q_{i+1}) \leq t_{opt} \cdot \|p_{i+1}q_{i+1}\|. \tag{3.6}$$

The inequality is obviously true for $i = 1$. For $i > 1$ it holds that

$$\|p_{i+1}q_{i+1}\| < \|p_2q_2\|$$

which implies that $(p_{i+1}, q_{i+1}) \notin \Delta(p_1, q_1)$ since (p_2, q_2) is the closest pair in $\Delta(p_1, q_1)$. This, in turn, implies that

$$\mathbf{d}_{\mathcal{G}}(p_{i+1}, q_{i+1}) = \mathbf{d}_{\mathcal{G}_1}(p_{i+1}, q_{i+1}) \leq t_{opt} \cdot \|p_{i+1}q_{i+1}\|.$$

Since $\mathcal{G}$ is a subgraph of $\mathcal{G}_i$, the length of the shortest path in $\mathcal{G}_i$ between p_{i+1} and q_{i+1} must be bounded by the length of the shortest path in $\mathcal{G}$ between p_{i+1} and q_{i+1}, which is bounded by $t_{opt} \cdot \|p_{i+1}q_{i+1}\|$. Thus, inequality (3.6) holds.

We continue with the second part of the proof. It will be shown that for every pair $u, v \in V$ it holds that

$$\frac{\mathbf{d}_{\mathcal{G}_j}(u,v)}{\|uv\|} < (2+\lambda) \cdot t_{opt}.$$

Note that $\mathcal{G}_1$ is the graph $\mathcal{G}$ augmented by the optimal edge.

If $(u,v) \notin \Delta(p_1,q_1)$ then we are done since $\mathbf{d}_{\mathcal{G}_j}(u,v) \le \mathbf{d}_{\mathcal{G}}(u,v) = \mathbf{d}_{\mathcal{G}_1}(u,v)$. Otherwise, if $(u,v) \in \Delta(p_1,q_1)$, the following holds (see Figure 3.3 for an illustration):

$$\begin{aligned}
\mathbf{d}_{\mathcal{G}_j}(u,v) &\le \mathbf{d}_{\mathcal{G}_1}(u,v) - \|p_1q_1\| + (\mathbf{d}_{\mathcal{G}_1}(p_2,q_2) - \|p_1q_1\|) + \cdots \\
&\qquad + (\mathbf{d}_{\mathcal{G}_{j-1}}(p_j,q_j) - \|p_{j-1}q_{j-1}\|) + \|p_jq_j\| \\
&< t_{opt} \cdot \|uv\| - \|p_1q_1\| + (t_{opt} \cdot \|p_2q_2\| - \|p_1q_1\|) + \cdots \\
&\qquad + (t_{opt} \cdot \|p_jq_j\| - \|p_{j-1}q_{j-1}\|) + \|p_jq_j\| \qquad \text{(cf. (3.6))} \\
&= t_{opt} \cdot \|uv\| - 2\|p_1q_1\| + ((t_{opt}-1) \cdot \|p_2q_2\|) + \cdots \\
&\qquad + ((t_{opt}-1) \cdot \|p_jq_j\|) + 2\|p_jq_j\| \\
&< t_{opt} \cdot \|uv\| + (t_{opt}-1)(\|p_2q_2\| + \ldots + \|p_jq_j\|) \\
&\qquad\qquad \text{(since } \|p_jq_j\| < \|p_1q_1\|) \\
&< t_{opt} \cdot \|uv\| + t_{opt} \cdot \sum_{i=2}^{j} \left(\frac{\lambda}{2}\right)^{i-2} \|p_2q_2\| \\
&\qquad\qquad \text{(since } \|p_{i+1}q_{i+1}\| \le (\tfrac{\lambda}{2}) \cdot \|p_iq_i\|) \\
&\le t_{opt} \cdot \|uv\| + t_{opt} \cdot \|uv\| \cdot \sum_{i=0}^{j-2} \left(\frac{\lambda}{2}\right)^{i} \qquad \text{(since } \|p_2q_2\| \le \|uv\|) \\
&= 2t_{opt} \cdot \|uv\| + t_{opt} \cdot \|uv\| \cdot \sum_{i=1}^{j-2} \left(\frac{\lambda}{2}\right)^{i} \\
&\le 2t_{opt} \cdot \|uv\| + t_{opt} \cdot \|uv\| \cdot \lambda \cdot \sum_{i=1}^{j-2} \frac{1}{2^i} \qquad \text{(since } \lambda \le 1) \\
&< (2+\lambda) \cdot t_{opt} \cdot \|uv\|
\end{aligned}$$

This concludes the lemma. ⊒

In the previous lemma we showed the existence of a "short" candidate edge (p,q) for which the resulting graph has small dilation. Note that algorithm Approximate ExpandGraph might not test (p,q). However, in the following lemma it will be shown that algorithm Approximate ExpandGraph will test an edge (a,b) that is almost as good as (p,q).

Lemma 3.4.3 *For any given constant $0<\varepsilon\le 1$ it holds that the graph $\mathcal{G}'$ returned by algorithm* Approximate ExpandGraph *has dilation at most $(2+\varepsilon)\cdot t_{opt}$.*

Proof. According to Lemma 3.4.2, for each $0<\lambda\le 1$ there exists an edge (p,q) such that for every pair $(u,v)\in\Delta(p,q)$ it holds that $\|uv\|\ge\frac{\lambda}{2}\|pq\|$, and the dilation t_H of $H=\mathcal{G}\cup\{(p,q)\}$ is bounded by $(2+\lambda)\cdot t_{opt}$. Let (A_i,B_i) be the well-separated pair computed in line 2 of the algorithm such that $p\in A_i$ and $q\in B_i$. According to Definition 2.3.3 such a well-separated pair must exist. Next, consider the candidate edge (a_i,b_i) tested by the algorithm, such that $a_i,p\in A_i$ and $b_i,q\in B_i$. For simplicity of writing we will use a and b to denote a_i and b_i respectively.

Our claim is that the dilation of $\mathcal{G}':=\mathcal{G}\cup\{(a,b)\}$ is bounded by $(1+\varepsilon/4)\cdot t_H$. Before we continue to prove the claim, we show that

$$\mathbf{d}_{\mathcal{G}}(a,p)=\mathbf{d}_H(a,p)\quad\text{and}\quad\mathbf{d}_{\mathcal{G}}(b,q)=\mathbf{d}_H(b,q). \tag{3.7}$$

Lemma 3.4.2 states that if $(x',y')\in\Delta(p,q)$ then

$$\|x'y'\|\ge(\varepsilon/8)\|pq\|.$$

But by Lemma 2.3.2 the distances $\|ap\|$ and $\|bq\|$ are less than

$$\frac{2}{s}\|pq\|=\frac{\varepsilon^2}{128}\|pq\|,$$

which is less than $(\varepsilon/8)\|pq\|$ since $\varepsilon\le 1$. As a consequence $(a,p)\notin\Delta(p,q)$ and $(b,q)\notin\Delta(p,q)$, thus $(p,q)\notin\delta_H(a,p)$ and $(p,q)\notin\delta_H(b,q)$. Hence, the equation (3.7) holds, which we will need below.

To prove the claim, we show that the dilation between any pair of points in $\mathcal{G}'$ is bounded by $(1+\varepsilon/4)\cdot t_H$. Let x and y be arbitrary points of V. We have two cases:

Case 1: $(x,y)\notin\Delta(p,q)$.

In this case we are done, since

$$\mathbf{d}_{\mathcal{G}'}(x,y)\le\mathbf{d}_{\mathcal{G}}(x,y)=\mathbf{d}_H(x,y).$$

Case 2: $(x,y)\in\Delta(p,q)$.

Consider the length of the path in $\mathcal{G}'$ between x and y as illustrated in Fig-

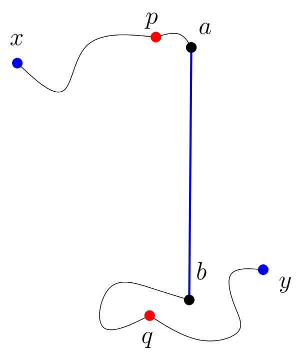


Figure 3.4: Illustrating the proof of Lemma 3.4.3.

ure 3.4. Without loss of generality we have:

$$\begin{aligned}
\mathbf{d}_{\mathcal{G}'}(x,y) &\le \mathbf{d}_{\mathcal{G}}(x,p) + \mathbf{d}_{\mathcal{G}}(p,a) + \|ab\| + \mathbf{d}_{\mathcal{G}}(b,q) + \mathbf{d}_{\mathcal{G}}(q,y) \\
&= \mathbf{d}_{\mathcal{G}}(x,p) + \mathbf{d}_{H}(p,a) + \|ab\| + \mathbf{d}_{H}(b,q) + \mathbf{d}_{\mathcal{G}}(q,y) \quad \text{(cf. (3.7))} \\
&\le \mathbf{d}_{\mathcal{G}}(x,p) + \|ab\| + \mathbf{d}_{\mathcal{G}}(q,y) + t_H \cdot (\|pa\| + \|bq\|) \\
&\le \mathbf{d}_{\mathcal{G}}(x,p) + (1+\frac{4}{s}) \cdot \|pq\| + \mathbf{d}_{\mathcal{G}}(q,y) + \frac{4t_H}{s} \cdot \|pq\| \quad \text{(Lemma 2.3.2)} \\
&\le \mathbf{d}_{H}(x,y) + \frac{8t_H}{s} \cdot \|pq\| \\
&\le \mathbf{d}_{H}(x,y) + \frac{64t_H}{\varepsilon s} \cdot \|xy\| \quad \text{(Lemma 3.4.2)} \\
&= \mathbf{d}_{H}(x,y) + \frac{\varepsilon}{4} \cdot t_H \cdot \|xy\|.
\end{aligned}$$

Therefore the dilation between x and y in $\mathcal{G}'$ is:

$$\frac{\mathbf{d}_{\mathcal{G}'}(x,y)}{\|xy\|} \le \frac{\mathbf{d}_{H}(x,y)}{\|xy\|} + \frac{(\varepsilon/4)t_H\|xy\|}{\|xy\|} \le \left(1+\frac{\varepsilon}{4}\right) \cdot t_H.$$

Finally, according to Lemma 3.4.2 and by setting $\lambda := \varepsilon/4$ it holds that

$$t_H \le (2+\varepsilon/4) \cdot t_{opt}.$$

This completes the lemma since $(2+\varepsilon/4)(1+\varepsilon/4) < (2+\varepsilon)$. □

We may now conclude this section with the following theorem.

Theorem 3.4.4 *Given a Euclidean graph $\mathcal{G} = (V,E)$ in $\mathbb{R}^d$ one can in $\mathcal{O}(n^3/\varepsilon^{2d})$ time, using $\mathcal{O}(n^2)$ space, compute a t'-spanner $\mathcal{G}' = (V, E \cup \{e\})$, where $t' \le (2+\varepsilon) \cdot t_{opt}$.*

3.4.2 Speeding up algorithm 3.4.1

In the previous section we showed that a $(2+\varepsilon)$-approximate solution can be obtained by testing a linear number of candidate edges. Testing each candidate edge entails $\mathcal{O}(n^2)$ shortest path queries for computing the dilation of the candidate graph. One way to speed up the computation is to compute an approximate dilation. As in Section 3.2.2 we will use Fact 3.2.2 by Narasimhan and Smid [NS00].

We will use their idea to speed up line 6 of Approximate Expandgraph from $\mathcal{O}(n^2)$ to $\mathcal{O}(n/\varepsilon^d)$, i.e., we check a linear number of pairs in order to compute an approximate dilation using Fact 3.2.2. There will be two main changes in the Approximate ExpandGraph algorithm; two well-separated pair decompositions will be computed and the computation of the dilation will be different, see Algorithm 3.4.2. Instead of computing the exact dilation of $\mathcal{G}$ with the candidate edge (a_i, b_i) added to $\mathcal{G}$, we compute the approximate dilation. This is done by a call to Approximate Dilation, or ApproxDilation for short, with parameters M, $\mathcal{S}$ and (a_i, b_i). The ApproxDilation algorithm is stated in more detail below. Note that the number of point pairs in $\mathcal{S}$ is bounded by $\mathcal{O}(n/\varepsilon^d)$.

Algorithm 3.4.2: Approximate ExpandGraph2$(\mathcal{G}, \varepsilon)$

```
   Input: Euclidean graph $\mathcal{G} = (V, E)$ and a real constant $\varepsilon > 0$.
   Output: Euclidean graph $\mathcal{G}' = (V, E \cup \{e\})$.
1  $M$ := distance matrix of $\mathcal{G}$;
2  $\{(C_j, D_j)\}_{j=1}^{\ell}$ := WSPD of the set $V$ with respect to $s' = 4(1+\varepsilon)/\varepsilon$;
3  $\mathcal{S} := \emptyset$;
4  for $j := 1, 2, \ldots, \ell$ do
5      Select an arbitrary point $c_j$ of $C_j$ and an arbitrary point $d_j$ of $D_j$;
6      Add $(c_j, d_j)$ to $\mathcal{S}$;
7  end
8  $t := \infty$;
9  $\{(A_i, B_i)\}_{i=1}^{k}$ := WSPD of the set $V$ with respect to $s = 256/\varepsilon^2$;
10 for $i := 1, 2, \ldots, k$ do
11     Select an arbitrary point $a_i$ of $A_i$ and an arbitrary point $b_i$ of $B_i$;
12     $t_i$ := ApproxDilation$(M, \mathcal{S}, a_i, b_i)$;
13     if $t_i < t$ then
14         $t := t_i$ and $e := (a_i, b_i)$;
15     end
16 end
17 return $\mathcal{G}' = (V, E \cup \{e\})$;
```

Theorem 3.4.5 *Given a Euclidean graph $\mathcal{G} = (V, E)$ and a real constant $\varepsilon > 0$ one can in $\mathcal{O}(nm + n^2(\log n + 1/\varepsilon^{3d}))$ time, using $\mathcal{O}(n^2)$ space, compute a t'-spanner $\mathcal{G}' = (V, E \cup \{e\})$ such that $t' \leq (2+\varepsilon) \cdot t_{opt}$.*

```
Algorithm 3.4.3: ApproxDilation
   Input: Distance matrix $M$, a set $\mathcal{S}$ of point pairs and point pair $(a, b)$.
   Output: Approximate dilation.
1  foreach $(x, y) \in \mathcal{S}$ do
2      $d_{x,y} := \min\{M[x,y], M[x,a] + \|ab\| + M[b,y], M[x,b] + \|ba\| + M[a,y]\}$;
3      $t_{x,y} := d_{x,y}/\|xy\|$;
4  end
5  return $\max_{x,y} t_{x,y}$;
```

Proof. The complexity of all lines of the algorithm 3.4.2, except line 12, is as in Lemma 3.4.1. Lines 1–9 require $\mathcal{O}(mn + n^2 \log n + n/\varepsilon^{2d})$ time. It remains to consider line 12 of the algorithm. Note that the number of times line 12 is executed is $\mathcal{O}(n/\varepsilon^{2d})$. Procedure ApproxDilation performs $\mathcal{O}(n/\varepsilon^d)$ shortest-path queries, instead of $\mathcal{O}(n^2)$, thus the total time needed by line 12 is $\mathcal{O}(\frac{n}{\varepsilon^{2d}} \cdot \frac{n}{\varepsilon^d})$. Summing up the running times gives the stated time complexity.

In Lemma 3.4.3 it was proven that the solution returned by algorithm Approximate ExpandGraph had a dilation that was at most a factor $(2+\varepsilon)$ worse than the dilation of an optimal solution. Since the modified algorithm does not compute the exact dilation of a candidate graph, but instead computes a $(1+\varepsilon)^2$-approximate dilation it is not hard to verify that the same arguments as in Lemma 3.4.3 can be applied to prove that the algorithm Approximate ExpandGraph2 returns a graph with dilation at most $(1+\varepsilon)^2 \cdot (2+\varepsilon) \cdot t_{opt}$. Setting $\varepsilon = \min\{\varepsilon/10, 1\}$, concludes the proof of the theorem. ◻

3.5 A special case: $\mathcal{G}$ has constant dilation

In the special case when the dilation of the graph $\mathcal{G}$ is known to be constant there are well-known tools that we can use to decrease both the time complexity and the space complexity of the algorithms and improve the approximation factor. The main idea is to use the following fact by Gudmundsson *et al.* [GLNS02b] which enable us to compute an approximate dilation of a candidate graph in linear time.

Fact 3.5.1 *([GLNS02b]) Let V be a set of n points in $\mathbb{R}^d$, let $t > 1$ and $0 < \varepsilon \leq 1$ be real numbers, and let $\mathcal{G} = (V, E)$ be a t-spanner for V. In*

$$\mathcal{O}\left(m + \frac{nt^{5d}}{\varepsilon^{2d}} (\log n + (t/\varepsilon)^d)\right)$$

time, we can preprocess $\mathcal{G}$ into a data structure of size $\mathcal{O}((t^{3d}/\varepsilon^{2d})\, n \log(tn))$ such that for any two distinct points p and q in V, a $(1+\varepsilon)$-approximation to the *shortest-path distance between p and q in $\mathcal{G}$ can be computed in time $\mathcal{O}\big((t^5/\varepsilon^2)^d\big)$.*

The query structure in Fact 3.5.1 is denoted M' and is constructed by algorithm QueryStructure. We have to use a modified version of ApproxDilation, denoted ApproxDilation2, that takes the query structure M' as input instead of the matrix M. The (exact) shortest path distance queries using M in ApproxDilation are replaced in ApproxDilation2 by performing approximate shortest path distance queries using M'.

Next we state the main algorithm. Recall that the parameter t is a constant and an upper bound on the dilation of the input graph $\mathcal{G}$. Also note that this algorithm only needs one well-separated pair decomposition.

Algorithm 3.5.1: Approximate ExpandGraph3($\mathcal{G}, t, \varepsilon$)

```
Input: Euclidean $t$-spanner $\mathcal{G} = (V, E)$ and two real constants $t > 1$ and
       $\varepsilon > 0$.
Output: Euclidean graph $\mathcal{G}' = (V, E \cup \{e\})$.
1  $M' :=$ QueryStructure($\mathcal{G}, t, \varepsilon$) using Fact 3.5.1;
2  $\{(A_i, B_i)\}_{i=1}^k :=$ WSPD of $V$ with respect to $s = 8(t+1)/\varepsilon$;
3  $\mathcal{S} := \emptyset$;
4  for $j := 1, 2, \ldots, k$ do
5      Select an arbitrary point $a_j$ of $A_j$ and an arbitrary point $b_j$ of $B_j$;
6      Add $(a_j, b_j)$ to $\mathcal{S}$;
7  end
8  $t := \infty$;
9  for $i := 1, 2, \ldots, k$ do
10     $t_i :=$ ApproxDilation2($M', \mathcal{S}, a_i, b_i$);
11     if $t_i < t$ then
12         $t := t_i$ and $e := (a_i, b_i)$;
13     end
14 end
15 return $\mathcal{G}' = (V, E \cup \{e\})$;
```

Lemma 3.5.2 Approximate ExpandGraph3 *runs in $\mathcal{O}((t^7/\varepsilon^4)^d \cdot n^2)$ time and uses $\mathcal{O}((t^3/\varepsilon^2)^d\, n \log(tn))$ space.*

Proof. The time complexity of lines 1–3 is dominated by line 1, thus $\mathcal{O}(m + n(t^5/\varepsilon^2)^d(\log n + (t/\varepsilon)^d))$ time. Lines 10–13 is executed $\mathcal{O}((t/\varepsilon)^d n)$ times, and each iteration requires $\mathcal{O}((t/\varepsilon)^d n \cdot (t^{5d}/\varepsilon^{2d}))$ time according to Facts 3.2.2 and 3.5.1. Summing up the time bounds gives the time bound stated in the lemma.

The space bound follows since the approximate distance oracle stated in Fact 3.5.1 only uses $\mathcal{O}((t^3/\varepsilon^2)^d\, n \log(tn))$ space, instead of the quadratic space needed earlier. ◻

Now, we show that this algorithm computes a $(1+\varepsilon)$-approximation of the optimal solution. Note that in Approximate ExpandGraph3 the separation constant depends both on ε and t which is the main difference compared to the previous algorithms. This allows us to improve the approximation factor.

Lemma 3.5.3 *Let $\mathcal{G} = (V, E)$ be a Euclidean graph with constant dilation t and a positive real constant ε, and let $\{(A_i, B_i)\}_{i=1}^{k}$ be a well-separated pair decomposition of V with respect to $s = 8(t+1)/\varepsilon$. For every pair (A_i, B_i) and any elements $a_1, a_2 \in A_i$ and $b_1, b_2 \in B_i$, let $\mathcal{G}_1 = (V, E \cup \{(a_1, b_1)\})$ and $\mathcal{G}_2 = (V, E \cup \{(a_2, b_2)\})$, and let t_1 and t_2 denote the dilation of $\mathcal{G}_1$ and $\mathcal{G}_2$, respectively. It holds that $t_1 \leq (1+\varepsilon)t_2$.*

Proof. It suffices to prove that for every pair of points $(u, v) \in \Delta(a_2, b_2)$ there exists a path in $\mathcal{G}_1$ of length at most $(1+\varepsilon) \cdot \mathbf{d}_{\mathcal{G}_2}(u, v)$. Recall that $\Delta(p, q)$ is the set of point pairs for which a shortest path between them in $\mathcal{G} \cup \{(p, q)\}$ passes through (p, q). Without loss of generality we may assume that the shortest path between u and v in $\mathcal{G}_2$, goes from u to a_2 and to v via b_2. We have:

$$\begin{aligned}
\mathbf{d}_{\mathcal{G}_1}(u, v) &\leq \mathbf{d}_{\mathcal{G}}(u, a_2) + \mathbf{d}_{\mathcal{G}}(a_2, a_1) + \|a_1 b_1\| + \mathbf{d}_{\mathcal{G}}(b_1, b_2) + \mathbf{d}_{\mathcal{G}}(b_2, v) \\
&\leq \mathbf{d}_{\mathcal{G}}(u, a_2) + t\|a_2 a_1\| + \|a_1 b_1\| + t\|b_1 b_2\| + \mathbf{d}_{\mathcal{G}}(b_2, v) \\
&\leq \mathbf{d}_{\mathcal{G}}(u, a_2) + \frac{4t}{s}\|a_2 b_2\| + (1 + 4/s) \cdot \|a_2 b_2\| + \mathbf{d}_{\mathcal{G}}(b_2, v) \\
&< \mathbf{d}_{\mathcal{G}}(u, a_2) + \|a_2 b_2\| + \mathbf{d}_{\mathcal{G}}(b_2, v) + \frac{8t}{s}\|a_2 b_2\| \\
&= \mathbf{d}_{\mathcal{G}_2}(u, v) + \frac{t\varepsilon}{t+1}\|a_2 b_2\| \\
&< (1+\varepsilon) \cdot \mathbf{d}_{\mathcal{G}_2}(u, v).
\end{aligned}$$

In the second inequality we used Lemma 2.3.2, in the fifth inequality we used the fact that $s = 8(t+1)/\varepsilon$ and in the final step we used that $\mathbf{d}_{\mathcal{G}_2}(u, v) \geq \|a_2 b_2\|$ since $(u, v) \in \Delta(a_2, b_2)$. The lemma follows. ◻

Lemma 3.5.4 *Algorithm* Approximate ExpandGraph3 *returns a graph with dilation at most $(1+\varepsilon)^3 \cdot t_{opt}$.*

Proof. Assume that t_{opt} is the dilation of an optimal solution $\mathcal{G} \cup \{(p, q)\}$, and let $\mathcal{G}'$ with dilation $t_{\mathcal{C}}$ be the output of the algorithm 3.5.1.

We will use the same notations as in the algorithm. For each i let t_i^* be the (exact) dilation of $\mathcal{G}_i = \mathcal{G} \cup \{(a_i, b_i)\}$. According to Fact 3.2.2, for each i,

$$t_i^* \leq t_i \leq (1+\varepsilon)^2 \cdot t_i^*.$$

Let (A_j, B_j) be the pair in the well-separated pair decomposition such that $p \in A_j$ and $q \in B_j$, or $p \in B_j$ and $q \in A_j$. From Lemma 3.5.3 it follows that

$t_j^* \leq (1+\varepsilon) \cdot t_{opt}$. As a result it follows that

$$t_{\mathcal{C}} \leq t_j \leq (1+\varepsilon)^2 \cdot t_j^* \leq (1+\varepsilon)^3 \cdot t_{opt}.$$

Therefore $t_{opt} \leq t_{\mathcal{C}} \leq (1+\varepsilon)^3 \cdot t_{opt}$ which completes the lemma. $\sqcup\!\!\!\!\sqcap$

The following theorem follows by setting $\varepsilon = \min\{\varphi/15, 1\}$ and combining Lemmas 3.5.2 and 3.5.4.

Theorem 3.5.5 *Let V be a set of n points in $\mathbb{R}^d$, let $t > 1$ and $\varphi > 0$ be real numbers, and let $\mathcal{G} = (V, E)$ be a geometric t-spanner of V. One can in $\mathcal{O}((t^7/\varphi^4)^d \cdot n^2)$ time, using $\mathcal{O}((t^3/\varphi)^d n \log(tn))$ space, compute a t'-spanner $\mathcal{G}' = (V, E \cup \{e\})$ such that $t' \leq (1+\varphi) \cdot t_{opt}$.*

3.6 Concluding remarks

We considered the problem of adding an edge to a Euclidean graph such that the dilation of the resulting graph is minimized, and gave several algorithms. Our main result is a $(2+\varepsilon)$-approximation algorithm with running time

$$\mathcal{O}\left(nm + n^2(\log n + 1/\varepsilon^{3d})\right)$$

using $\mathcal{O}(n^2)$ space.

Several problems remain open.

1. Is there an exact algorithm with running time $o(n^4)$ using linear space?
2. Can we achieve a $(1+\varepsilon)$-approximation within the same time bound as in Theorem 3.4.5?
3. A natural extension is to allow more than one edge to be added. Can we generalize our results to this case?

Acknowledgements

We would like to thank René van Oostrum for fruitful discussions during the early stages of this work, Mohammad Ali Abam for discussions about Section 3.2.2, Sergio Cabello for simplifying the algorithm in Section 3.5 and Ansgar Grüne for Observation 3.3.3.

Finally, we thank the anonymous referees for many insightful comments and suggestions on how to improve the journal version of this chapter.

Chapter 4

Dilation-Optimal Edge Deletion

4.1 Introduction

Given a (geometric) network, a natural question to ask is what happens to the quality of the network when some connections are removed. In case some links in a traffic network have to be shut down (e.g., due to budget considerations), we may want to know which edges of the network should be removed so as to not decrease the quality of the new network too much. Alternatively, we may want to know the most critical edge in the network, i.e., the edge whose removal causes the largest possible decrease in the quality of the new network.

In this chapter, we consider a simple variant of this problem: The initial network is a polygonal cycle $\mathfrak{C}$ in the plane, and we have to remove one single edge from $\mathfrak{C}$. We measure the quality of the resulting polygonal path Π by its *dilation* (or stretch factor) $\mathfrak{D}_\Pi$.

Recall that the dilation between two distinct *vertices* x and y of the path Π is defined as

$$\mathfrak{D}_\Pi(x, y) := \frac{\mathbf{d}_\Pi(x, y)}{\|xy\|},$$

where $\mathbf{d}_\Pi(x, y)$ denotes the Euclidean length of the subpath of Π connecting x and y, and $\|xy\|$ denotes the Euclidean distance between x and y. For convenience we define $\mathfrak{D}_\Pi(x, x) := 1$. The dilation between two *sets* X and Y of *vertices* of Π is defined as

$$\mathfrak{D}_\Pi(X,Y) := \max\{\mathfrak{D}_\Pi(x,y) \mid x \text{ is a vertex of } X,\, y \text{ is a vertex of } Y\},$$

the dilation of a *set* X of *vertices* of Π is defined as

$$\mathfrak{D}_\Pi(X) := \mathfrak{D}_\Pi(X,X),$$

and the *dilation* of the path Π is defined as

$$\mathfrak{D}_\Pi := \mathfrak{D}_\Pi(\Pi) = \max\{\mathfrak{D}_\Pi(x,y) \mid x \text{ and } y \text{ are vertices of } \Pi\}.$$

Recently the problem of constructing minimum-dilation networks on a given point set has attracted a lot of attention. Klein and Kutz [KK07] showed that constructing a geometric network on a point set in the plane using a given number of edges such that the network has minimum dilation is NP-hard. The problem is NP-hard even for more specific cases like minimum dilation spanning tree or minimum dilation path, see [CHL07] and [GKM07]. Minimum dilation stars is the case which we can construct in polynomial time, see [EW07].

All the above problems want to construct a network from scratch but in many applications we already have a network and the problem at hand is to extend/prune the network such that the dilation of the resulting network is minimized. In Chapter 3 we saw that in the case of adding an edge, the optimal edge, i.e. the new edge which minimize the dilation of the network, can be computed in $\mathcal{O}(n^4)$ time, where n is the number of nodes in the network. There are also several approximation algorithms which run faster. However, the problem of computing the edge in the network whose removal minimizes/maximizes the dilation of the resulting network was not studied before, to the best of our knowledge.

The problem we consider is the following: We are given a polygonal cycle $\mathfrak{C} = (p_0, \dots, p_{n-1}, p_0)$ whose n vertices $p_0, \dots, p_{n-1}$ are points in the plane. We want to determine the edge e of $\mathfrak{C}$ for which the dilation of the polygonal path $\mathfrak{C} \setminus \{e\}$ is minimized or maximized. In other words, if we denote by Π_i (for $0 \le i < n$) the polygonal path obtained by removing the edge (p_i, p_{i+1}) from $\mathfrak{C}$ (where indices are to be read modulo n), then our goal is to compute

$$\mathfrak{D}^{\min}_{\mathfrak{C}} := \min_{0 \le i < n} \mathfrak{D}_{\Pi_i}$$

and

$$\mathfrak{D}^{\max}_{\mathfrak{C}} := \max_{0 \le i < n} \mathfrak{D}_{\Pi_i}.$$

It is known that computing the dilation of a path, or even approximating it, need $\Omega(n \log n)$ time, see [NS07, Theorem 13.1.3]. Therefore using a naïve algorithm which checks all the edges of the cycle takes $\Omega(n^2 \log n)$ time to compute the optimal edge in the cycle.

A summary of our results and the organization of the rest of this chapter are in the following.

In Section 4.2, we consider the dilation-minimal edge deletion problem and present a randomized algorithm for the problem. We start in Section 4.2.1 by describing an approach of [AKK$^+$] to estimate the dilation of a polygonal path. These ideas will play a central role in the algorithm we give in Section 4.2.2 for solving a decision problem associated with the problem of computing $\mathfrak{D}_{\mathfrak{C}}^{\min}$; the algorithm solving this decision problem runs in $\mathcal{O}(n \log^2 n)$ expected time. In Section 4.2.3, we give a simple randomized approach which reduces the problem of computing $\mathfrak{D}_{\mathfrak{C}}^{\min}$ to an expected number of $\mathcal{O}(\log n)$ decision problems of Section 4.2.2. Thus, this reduction incurs a logarithmic slowdown of the decision procedure.

In Section 4.3, we consider the dilation-maximal edge deletion problem and present a $\mathcal{O}(n \log n)$ time algorithm for the problem. We first show that for two fixed vertices x and y of $\mathfrak{C}$, it is easy to determine the largest possible dilation between them if one edge is removed from $\mathfrak{C}$. We then show that, in order to compute $\mathfrak{D}_{\mathfrak{C}}^{\max}$, it suffices to consider pairs (x, y) of vertices whose distance is at most twice the closest-pair distance in the vertex set of $\mathfrak{C}$. Since there are only $\mathcal{O}(n)$ such pairs (x, y), this leads to an efficient algorithm for computing $\mathfrak{D}_{\mathfrak{C}}^{\max}$.

Finally, we present an algorithm that computes in $\mathcal{O}(n \log n)$ total time an approximation to the dilation of each path P_i, as well as an approximation of $\mathfrak{D}_{\mathfrak{C}}^{\min}$ in Section 4.4. The algorithm uses Fact 3.2.2 which states that the well-separated pair decomposition can be used to reduce the problem of approximating the dilation of a Euclidean graph to the problem of computing the shortest-path distances between $\mathcal{O}(n)$ pairs of vertices. This result, together with the observation that for any two vertices x and y of $\mathfrak{C}$, the sequence $\mathfrak{D}_{\Pi_0}(x, y), \dots, \mathfrak{D}_{\Pi_{n-1}}(x, y)$ contains only two distinct values, leads to an $\mathcal{O}(n \log n)$–time algorithm that approximates the dilation of each path Π_i as well as the minimum dilation $\mathfrak{D}_{\mathfrak{C}}^{\min}$.

4.2 Dilation-minimal edge deletion in a cycle

In this section, we give an algorithm which given a polygonal cycle $\mathfrak{C}$, computes the edge whose removal generates a path which has minimum dilation among all the paths generated by removing an edge from $\mathfrak{C}$. The algorithm starts with removing a random edge from $\mathfrak{C}$ and computes the dilation κ of the generated path. Then it uses a decision algorithm which for each edge e in the cycle $\mathfrak{C}$, decides whether the dilation of the path $\mathfrak{C} \setminus \{e\}$ is less than κ. Then it picks one of the edges whose removal generates a path with dilation less than κ and assigns the dilation of the path to κ. It repeats the procedure until no edge in the cycle generates a path with dilation less than κ.

4.2.1 Estimating the dilation of a polygonal path

Our algorithm for computing the edge of a polygonal cycle whose removal minimizes the dilation of the resulting path uses as a subroutine parts of the algorithm of [AKK$^+$] that decides if the dilation of a polygonal path is less than some given threshold $\kappa > 1$. We describe those parts of this algorithm which are relevant for us.

Let $\Pi = (p_0, \ldots, p_{n-1})$ be a polygonal path whose n vertices are points in the plane and let $\kappa \geq 1$ be a real number. Without loss of generality we assume p_0 is the origin. The idea is to use a lifting transformation that rephrases the decision problem, i.e., the problem of deciding if $\mathfrak{D}_\Pi < \kappa$, into a point-cone incidence-problem in $\mathbb{R}^3$.

We denote the first and last vertices of a polygonal path Π by $f(\Pi)$ and $l(\Pi)$, respectively. Thus, $f(\Pi) = p_0$. For each vertex p of Π, we define the *weight* of p to be

$$\omega_\Pi(p) := \mathbf{d}_\Pi(p, f(\Pi))/\kappa.$$

We map each vertex $p = (x_p, y_p)$ of Π to the point

$$h_\Pi(p) := (x_p, y_p, \omega_\Pi(p)) \in \mathbb{R}^3.$$

Let C denote the three-dimensional cone

$$C := \{(x, y, z) \in \mathbb{R}^3 \mid z = \sqrt{x^2 + y^2}\}.$$

We map each vertex p of Π to the cone

$$C_\Pi(p) := C \oplus h_\Pi(p) = \{c + h_\Pi(p) \mid c \in C\}.$$

Note that we can reformulate the definition of $C_\Pi(p)$ as

$$C_\Pi(p) := \left\{(x, y, z) \mid z - \omega_\Pi(p) = \sqrt{(x - x_p)^2 + (y - y_p)^2}\right\}, \tag{4.1}$$

and therefore $C_\Pi(p)$ is the graph of the bivariate function $f_p(q) = \|pq\| + \omega_\Pi(p)$ for $q \in \mathbb{R}^2$. If p and q are vertices of Π, then we say that p is *before* q on Π, if

$$\mathbf{d}_\Pi(p, f(\Pi)) < \mathbf{d}_\Pi(q, f(\Pi));$$

this will be denoted as $p <_\Pi q$. We then get the following lemma.

Lemma 4.2.1 *For any two vertices p and q of Π with $p <_\Pi q$, we have*

$$\mathfrak{D}_\Pi(p, q) < \kappa \textit{ if and only if } h_\Pi(q) \textit{ lies below } C_\Pi(p).$$

Proof. By straightforward algebraic manipulation, see Figure 4.1 for an illustration, we have

$$\begin{aligned}
\mathfrak{D}_\Pi(p,q) < \kappa \quad &\iff \quad \frac{\mathbf{d}_\Pi(q,p)}{\|qp\|} < \kappa \\
&\iff \quad \frac{\mathbf{d}_\Pi(f(\Pi),q) - \mathbf{d}_\Pi(f(\Pi),p)}{\|qp\|} < \kappa \\
&\iff \quad \frac{\mathbf{d}_\Pi(f(\Pi),q)}{\kappa} - \frac{\mathbf{d}_\Pi(f(\Pi),p)}{\kappa} < \|qp\| \\
&\iff \quad \omega_\Pi(q) - \omega_\Pi(p) < \|qp\| \\
&\iff \quad \omega_\Pi(q) - \omega_\Pi(p) < \sqrt{(x_q - x_p)^2 + (y_q - y_p)^2}. \qquad (4.2)
\end{aligned}$$

By combining Equations 4.1 and 4.2 complete proof of the lemma. ◻

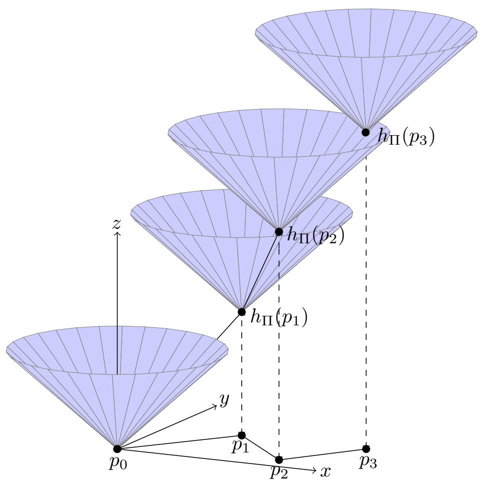


Figure 4.1: Illustration of the Lemma 4.2.1.

If X and Y are subsets of the vertex set of Π, then we say that X is *before* Y on Π, if $\mathbf{d}_\Pi(x, f(\Pi)) < \mathbf{d}_\Pi(y, f(\Pi))$ for all $x \in X$ and all $y \in Y$; this will be denoted as $X <_\Pi Y$. For any subset X of the vertex set of Π, we define $C_\Pi(X) := \{C_\Pi(p) \mid p \in X\}$ and $h_\Pi(X) := \{h_\Pi(p) \mid p \in X\}$.

The lower envelope of a set S of bi-variate functions is defined as the point-wise minimum of all functions that belong to S. Namely, the lower envelope of the set $S = \{f_1, \ldots, f_n\}$ can be defined as the following function:

$$\mathcal{L}_S(x,y) := \min_{1 \le k \le n} \overline{f_k}(x,y),$$

where

$$\overline{f_k}(x,y) := \begin{cases} f_k(x,y) & \text{if } (x,y) \text{ belongs to the domain of } f_k \\ +\infty & \text{otherwise.} \end{cases}$$

Lemma 4.2.1 immediately gives the following result.

Lemma 4.2.2 *For any two subsets X and Y of the vertex set of Π with $X <_\Pi Y$, we have*

$$\mathfrak{D}_\Pi(X,Y) < \kappa \text{ if and only if } h_\Pi(Y) \text{ lies below } \mathcal{L}_{C_\Pi(X)}.$$

The minimization diagram of $C_\Pi(X)$, i.e., the projection of the lower envelope $\mathcal{L}_{C_\Pi(X)}$ onto the xy-plane, is the additively weighted Voronoi diagram[1] $V_\Pi(X)$ of X with respect to the weight function ω_Π. If the point y of Y is located in the Voronoi region of the point x of X, then $h_\Pi(y)$ is below $\mathcal{L}_{C_\Pi(X)}$ if and only if $h_\Pi(y)$ is below $C_\Pi(x)$.

This yields an efficient algorithm to verify if $\mathfrak{D}_\Pi(X,Y) < \kappa$ for two subsets X and Y of the vertex set of Π having the property that $X <_\Pi Y$: The Voronoi diagram $V_\Pi(X)$ can be computed in $\mathcal{O}(|X| \log |X|)$ time, c.f. [For87]. Within the same time bound, this diagram can be preprocessed into a linear size data structure that supports $\mathcal{O}(\log |X|)$-time point-location queries, c.f. [Kir83]. This structure can now be queried with each point y of Y to determine which point x of X contains y in its Voronoi cell. Once this is known, the check if $h_\Pi(y)$ is below $C_\Pi(x)$ can be performed in $\mathcal{O}(1)$ time. The total running time of this algorithm is $\mathcal{O}((|X| + |Y|) \log |X|)$.

4.2.2 The decision problem

Let $\mathfrak{C}$ be a polygonal cycle on a set of n vertices in the plane and let $\kappa > 1$ be a real number. In this section, we present an algorithm that decides for each edge e of $\mathfrak{C}$, whether or not the dilation of the polygonal path $\mathfrak{C} \setminus \{e\}$ is less than κ. We first describe the overall approach. Then, we give two implementations that yield running times of $\mathcal{O}(n \log^3 n)$ and $\mathcal{O}(n \log^2 n)$, respectively.

For a cycle $\mathfrak{C} = (p_0, p_1, \ldots, p_{n-1}, p_0)$, we call p_{i+1} the successor of p_i in $\mathfrak{C}$ (where indices are to be read modulo n) and denote it by $\mathbf{succ}(p_i)$. If $R = (r_1, \ldots r_m)$ and $Q = (q_1, \ldots, q_n)$ are two polygonal paths having the property that $q_1 = f(Q)$ is the successor of $l(R) = r_m$ in $\mathfrak{C}$, then we denote the concatenation of R and Q by $R \oplus Q$. Thus, $R \oplus Q$ is the polygonal path $(r_1, \ldots, r_m, q_1, \ldots, q_n)$.

In order to facilitate a recursive approach, we will consider the following more general problem: Assume that (the vertex set of) $\mathfrak{C}$ is partitioned into two polygonal paths T (the *top*) and B (the *bottom*) such that $\mathfrak{D}_T < \kappa$. We want to decide

[1] It's defined just like the usual Voronoi diagram, but each site has a weight, and the distance to a site is the usual Euclidean distance plus the site weight.

for each edge e of B, i.e., any edge e on the cycle $\mathfrak{C}$ with both endpoints in B, whether or not the dilation of $\mathfrak{C} \setminus \{e\}$ is less than κ. If we take $T = \emptyset$ then we obtain the original problem.

The details of the decision algorithm are presented in Algorithm 4.2.1.

Algorithm 4.2.1: Decision-Algorithm

```
   Input: Paths $T$ and $B$ and $\kappa > 1$.
   Output: yes or no for every edge of $B$.
 1 if $|B| = 2$ then
 2     return yes for the edge in $B$;
 3 else
 4     $l$ := the last vertex of $B$;      /* in counterclockwise order along $\mathfrak{C}$ */
 5     $r$ := the first vertex of $B$;     /* in counterclockwise order along $\mathfrak{C}$ */
 6     $m$ := the middle vertex of $B$;
 7     $B^r$ := the part of the path $B$ between $r$ and $m$;
 8     $B^l$ := the part of the path $B$ between succ($m$) and $l$;
 9     if $\mathfrak{D}_{T \oplus B^r} < \kappa$ then  Decision-Algorithm($T \oplus B^r, B^l, \kappa$);
10     else return no for each edge $e$ of $B^l$;
11     if $\mathfrak{D}_{B^l \oplus T} < \kappa$ then Decision-Algorithm($B^l \oplus T, B^r, \kappa$);
12     else return no for each edge $e$ of $B^r$;
13 end
```

The correctness of the algorithm is obvious. We will show below that after a preprocessing step taking $\mathcal{O}(n \log^2 n)$ expected time, we can decide in $\mathcal{O}(|B| \log n)$ time if $\mathfrak{D}_{T \oplus B^r} < \kappa$ and $\mathfrak{D}_{B^l \oplus T} < \kappa$, where $|B|$ denotes the number vertices on B. The expected running time $t(n)$ of the algorithm can therefore be written as $t(n) = \mathcal{O}(n \log^2 n) + r(n)$ where the function r satisfies the recurrence

$$r(b) \leq 2 \cdot r(b/2) + \mathcal{O}(b \log n).$$

This implies that $t(n) = \mathcal{O}(n \log^2 n)$.

Figure 4.2 illustrates the recursion tree of the algorithm. The nodes of the tree are labeled according to a breadth-first search (BFS) numbering where the first (left) child of a node corresponds to the recursive call in line 11 and the second (right) child corresponds to the recursive call in line 9. Later, we will refer to a recursive call corresponding to the node with BFS-number i as the i-th step of the recursion. For each node i, the current top and bottom paths are denoted by T_i and B_i, respectively. These paths can be computed as follows. Assume that the polygonal cycle $\mathfrak{C}$ is given by the array $\mathfrak{C}[0, \ldots, n]$ and that B consists of b vertices. Then $B_1 = \mathfrak{C}[0, \ldots, b-1]$ and $T_1 = T$. For $i \geq 1$, if $B_i = \mathfrak{C}[l, r]$, then $B_{2i} := \mathfrak{C}[l, l + \lfloor \frac{r-l}{2} \rfloor]$, $B_{2i+1} := \mathfrak{C}[l + \lfloor \frac{r-l}{2} \rfloor + 1, r]$, $T_{2i} := B_{2i+1} \oplus T_i$, and

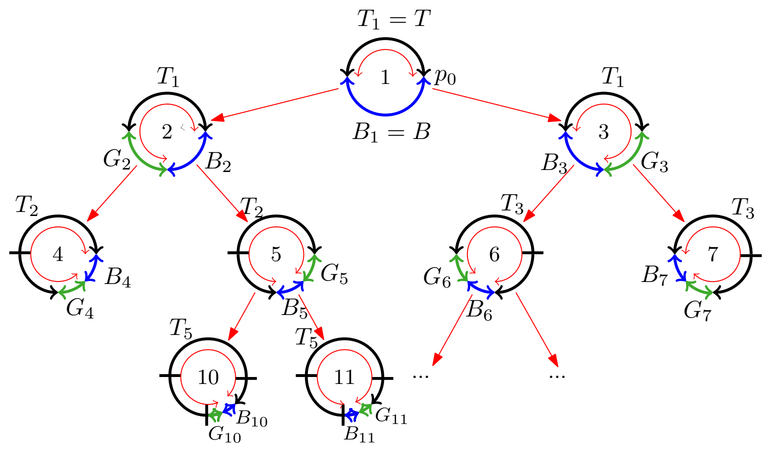


Figure 4.2: The recursion tree. Note that $G_{2i} := B_{2i+1}$ and $G_{2i+1} := B_{2i}$.

$T_{2i+1} := T_i \oplus B_{2i}$. Observe that each top chain T_j is the concatenation of $\mathcal{O}(\log n)$ many bottom chains.

A first implementation

We will show that after an $\mathcal{O}(n \log^2 n)$–time preprocessing, we can decide if (i) $\mathfrak{D}_{T \oplus B^r} < \kappa$ and (ii) $\mathfrak{D}_{B^l \oplus T} < \kappa$ in $\mathcal{O}(|B| \log^2 n)$ time. Later, we will give a faster implementation. Since (ii) is symmetric to (i), we only show how to decide whether or not (i) holds.

Suppose we have a polygonal path T' with $\mathfrak{D}_{T'} < \kappa$ that is given as a list of k polygonal paths $(B'_1, \ldots, B'_k)$ such that $f(B'_{i+1})$ is the successor of $l(B'_i)$ in the cycle $\mathfrak{C}$, for $1 \leq i < k$. Thus, we have $T' = B'_1 \oplus \ldots \oplus B'_k$. Given a new polygonal path B' with $f(B') = \mathbf{succ}(l(T'))$ in $\mathfrak{C}$, we want to decide if $\mathfrak{D}_{T' \oplus B'} < \kappa$.

Observe that $\mathfrak{D}_{T' \oplus B'} < \kappa$ if and only if

(a) $\mathfrak{D}_{T'} < \kappa$,

(b) $\mathfrak{D}_{B'} < \kappa$, and

(c) $\mathfrak{D}_{T' \oplus B'}(T', B') < \kappa$.

We are given that (a) holds. Using the algorithm of [AKK$^+$] we can decide in $\mathcal{O}(|B'| \log |B'|)$ expected time whether or not (b) holds. Thus, it remains to show how to verify whether or not (c) holds.

Obviously, $\mathfrak{D}_{T' \oplus B'}(T', B') < \kappa$ if and only if $\mathfrak{D}_{T' \oplus B'}(B'_i, B') < \kappa$ for each i

with $1 \leq i \leq k$. Since $B'_i <_{T' \oplus B'} B'$, we know from Lemma 4.2.2 that

$$\mathfrak{D}_{T' \oplus B'}(B'_i, B') < \kappa \text{ if and only if } h_{T' \oplus B'}(B') \text{ lies below } \mathcal{L}_{C_{T' \oplus B'}(B'_i)}.$$

Assume that for each path B'_i, we have the total accumulated scaled length

$$\ell_i := \sum_{j=1}^{i} \mathbf{d}_{B'_j}(l(B'_j), f(B'_j))/\kappa$$

and the additively weighted Voronoi diagram $V_{B'_i}(B'_i)$ that has been augmented with a data structure to support point-location queries in t_{loc} time per query. Recall that $V_{B'_i}(B'_i)$ is the projection of the lower envelope $\mathcal{L}_{C_{B'_i}(B'_i)}$ onto the xy-plane. It is defined with respect to the weights

$$\omega_{B'_i}(p) = \mathbf{d}_{B'_i}(p, f(B'_i))/\kappa.$$

Since

$$\omega_{T' \oplus B'}(p) = \omega_{B'_i}(p) + \ell_{i-1}$$

for all $p \in B'_i$, we have

$$\omega_{T' \oplus B'}(p) - \omega_{T' \oplus B'}(q) = \omega_{B'_i}(p) - \omega_{B'_i}(q)$$

for all $p, q \in B'_i$. It follows that the diagram $V_{B'_i}(B'_i)$ is also the projection of the lower envelope $\mathcal{L}_{C_{T' \oplus B'}(B'_i)}$.

The associated point-location structure of B'_i can therefore be used to determine for each point b' in B', the point t in B'_i that contains b' in its Voronoi cell in $V_{T' \oplus B'}(B'_i)$. Once this is known for each point b' in B', we can check if $h_{T' \oplus B'}(b')$ is below $C_{T' \oplus B'}(t)$. To this end, we compute the weights $\omega_{T' \oplus B'}(t) = \omega_{B'_i}(t) + \ell_{i-1}$ and $\omega_{T' \oplus B'}(b') = \omega_{B'}(b') + \ell_k$.

The overall running time of this approach (excluding the preprocessing time) is $\mathcal{O}(k|B'|t_{loc})$.

In our application, the relevant paths B'_i are the bottom paths B_i that appear in the recursive calls (see Figure 4.2 and the discussion after that). As a consequence, $k = \mathcal{O}(\log n)$, $|T' \oplus B'| \leq n$, and we can precompute the required information in $\mathcal{O}(n \log^2 n)$ time by computing all the diagrams $V_{B_i}(B_i)$ along with the point-location data structures. Since $t_{loc} = \mathcal{O}(\log n)$, it follows that the overall running time of this approach (after $\mathcal{O}(n \log^2 n)$ preprocessing time) is $\mathcal{O}(|B'| \log^2 n)$. With this implementation, Algorithm 4.2.1 runs in $\mathcal{O}(n \log^3 n)$ expected time.

A faster implementation

In the i-th step of the recursion, we split B_i (almost evenly) into B_{2i} and B_{2i+1}, and compute the diagrams $V_{B_{2i}}(B_{2i})$ and $V_{B_{2i+1}}(B_{2i+1})$. We then locate each point b of B_{2i} in $V_{B_{2i+1}}(B_{2i+1})$ to determine which point t of B_{2i+1} contains b in its Voronoi cell in $V_{B_{2i+1}}(B_{2i+1})$. We store t in a table $\mathcal{T}_b$ associated with b under the key $2i+1$ that identifies the set B_{2i+1}. In the same way, we locate each point b of B_{2i+1} in $V_{B_{2i}}(B_{2i})$ and store the corresponding point t of B_{2i} in a table $\mathcal{T}_b$ associated with b under the key $2i$.

Since we perform exactly one point-location query for each point b of B_1 on each level of the recursion tree, the table $\mathcal{T}_b$ has $\mathcal{O}(\log n)$ entries. We can therefore use the construction of [FKS84] to store $\mathcal{T}_b$ in a perfect-hash table of size $\mathcal{O}(\log n)$ that supports $\mathcal{O}(1)$ access time. Note that in the complexity model of [FKS84], it is assumed that the entries come from a finite universe and the algorithm is able to randomly access each memory location in constant time. Recall that the construction of [FKS84] is randomized and builds the hash table in $\mathcal{O}(\log n)$ expected time.

The total time we spend on each level of the recursion tree is $\mathcal{O}(n \log n)$, so the total expected preprocessing time is $\mathcal{O}(n \log^2 n)$ and the total time we spend for answering point-location queries is $\mathcal{O}(n \log^2 n)$.

In order to determine for a point b' of B', where $B' \subseteq B_1$, which point t of B'_i contains b' in its Voronoi cell, we find the index j for which $B'_i = B_j$. Then we retrieve the entry with the key j from $\mathcal{T}_{b'}$. This is exactly the point t of B_j that contains b in its Voronoi cell in $V_{B_j}(B_j)$.

It follows that $t_{loc} = \mathcal{O}(1)$, so that the overall running time of this approach (after $\mathcal{O}(n \log^2 n)$ preprocessing time) is $\mathcal{O}(|B'| \log n)$. With this implementation, Algorithm 4.2.1 runs in $\mathcal{O}(n \log^2 n)$ expected time.

4.2.3 The optimization algorithm

We now present our algorithm that computes, for a given polygonal cycle $\mathfrak{C}$ on a set of n points in the plane, the value of $\mathfrak{D}_{\mathfrak{C}}^{\min}$ in $\mathcal{O}(n \log^3 n)$ expected time. Clarkson and Shor [CS88] used a similar randomized approach to compute the diameter of a point set. The main idea is to check all the edges in the cycle in a random order. We start with a random edge $e_1 \in \mathfrak{C}$ and compute the dilation of the path $\mathfrak{C} \setminus \{e_1\}$, using the algorithm of [AKK$^+$], and assign this value to κ. Then we run Algorithm 4.2.1 to see for each edge e of the cycle $\mathfrak{C}$ whether the dilation of the path $\mathfrak{C} \setminus \{e\}$ is less than κ. We store with each edge e of $\mathfrak{C}$ a Boolean flag, denoted by $flag(e)$, which is **true** if and only if the dilation of the path $\mathfrak{C} \setminus \{e\}$ is less than κ. For any edge e with $flag(e) =$ **true**, we compute and assign the dilation of $\mathfrak{C} \setminus \{e\}$ to κ and update flags same as before. For more details see algorithm 4.2.2.

The correctness of the algorithm follows from the fact that it returns

$$\kappa = \min_{1 \le i \le n} \mathfrak{D}_{\Pi_i} = \mathfrak{D}_{\mathfrak{C}}^{\min},$$

Algorithm 4.2.2: OPTIMIZATION ALGORITHM

```
Input: A cycle $\mathfrak{C}$ on a set of $n$ points.
Output: $\mathfrak{D}_{\mathfrak{C}}^{\min}$.
1  $\{e_i\}_i$ := random permutation of the edges of $\mathfrak{C}$;
2  $\kappa$ := the dilation of the path $\mathfrak{C} \setminus \{e_1\}$;
3  Run Algorithm 4.2.1 and for each edge $e$ of $\mathfrak{C}$, update $flag(e)$;
4  for $i = 2, 3, \dots, n$ do
5      if $flag(e_i) =$ true then
6          $\kappa$ := the dilation of the path $\mathfrak{C} \setminus \{e_i\}$;
7          Run Algorithm 4.2.1 and for each edge $e$ of $\mathfrak{C}$, update $flag(e)$;
8      end
9  end
10 return $\kappa$;
```

where Π_i is the polygonal path obtained by removing e_i from $\mathfrak{C}$.

Clearly, line 1 takes $\mathcal{O}(n)$ time. The algorithm of [AKK+] and, therefore, line 2, takes $\mathcal{O}(n \log n)$ expected time. Each time we run Algorithm 4.2.1, we spend $\mathcal{O}(n \log^2 n)$ expected time. Observe that we run this algorithm once in line 3 and, moreover, in lines 4–9 each time the dilation of Π_i is less than the current value of κ. In the latter case, we also spend $\mathcal{O}(n \log n)$ expected time to compute the dilation of Π_i. Since the edges of $\mathfrak{C}$ are in random order, the values $\mathfrak{D}_{\Pi_1}, \dots, \mathfrak{D}_{\Pi_n}$ are in random order as well. At the start of the i-th iteration of lines 4–9, the value of κ is equal to $\min_{1 \le j < i} \mathfrak{D}_{\Pi_j}$. Thus, $\mathfrak{D}_{\Pi_i} < \kappa$ if and only if $\mathfrak{D}_{\Pi_i}$ is the minimum of the set $\{\mathfrak{D}_{\Pi_j} \mid 1 \le j \le i\}$. It follows that $\mathfrak{D}_{\Pi_i} < \kappa$ with probability $1/i$. Using the linearity of expectation, it follows that the expected number of times that lines 6 and 7 are performed is equal to $\sum_{i=2}^{n} 1/i = \mathcal{O}(\log n)$. Thus, the overall expected running time of our algorithm is $\mathcal{O}(n \log^3 n)$.

Theorem 4.2.3 *Given a polygonal cycle $\mathfrak{C}$ on n vertices in the plane, we can compute $\mathfrak{D}_{\mathfrak{C}}^{\min}$ in $\mathcal{O}(n \log^3 n)$ expected time.*

4.3 Dilation-maximal edge deletion in a cycle

Let $\mathfrak{C} = (p_0, \dots, p_{n-1}, p_0)$ be a polygonal cycle whose vertices $p_0, \dots, p_{n-1}$ are points in the plane. Recall from Section 4.1 that Π_i (for $0 \le i < n$) denotes the polygonal path obtained by removing the edge (p_i, p_{i+1}) from $\mathfrak{C}$, $\mathbf{d}_{\Pi_i}(x, y)$ denotes the length of the subpath of Π_i between x and y, $\mathfrak{D}_{\Pi_i}(x, y) = \mathbf{d}_{\Pi_i}(x, y)/\|xy\|$ denotes the dilation between x and y in Π_i, and $\mathfrak{D}_{\Pi_i}$ denotes the dilation of Π_i. In this section, we will give an algorithm that computes

$$\mathfrak{D}_{\mathfrak{C}}^{\max} = \max_{0 \le i < n} \mathfrak{D}_{\Pi_i}.$$

Let L be the total length of the edges of $\mathfrak{C}$. We define $\Delta(p_0) := 0$ and

$$\Delta(p_i) := \Delta(p_{i-1}) + \|p_{i-1}p_i\| \text{ for } 1 \le i < n.$$

Thus, $\Delta(p_i)$ is the length of the path $(p_0, \dots, p_i)$ and the shortest-path distance $\mathbf{d}_{\mathfrak{C}}(p_i, p_j)$ between p_i and p_j in the cycle $\mathfrak{C}$ is given by

$$\mathbf{d}_{\mathfrak{C}}(p_i, p_j) = \min(|\Delta(p_i) - \Delta(p_j)|, L - |\Delta(p_i) - \Delta(p_j)|).$$

Consider two distinct vertices x and y of $\mathfrak{C}$. We obtain the largest dilation between x and y in any path Π_i, by deleting an arbitrary edge on the shorter of the two paths in $\mathfrak{C}$ between x and y. Thus, the following lemma holds.

Lemma 4.3.1 *Let x and y be two distinct vertices of $\mathfrak{C}$. Then*

$$\max_{0 \le i < n} \mathfrak{D}_{\Pi_i}(x, y) = \frac{\max(|\Delta(x) - \Delta(y)|, L - |\Delta(x) - \Delta(y)|)}{\|xy\|} \ge \frac{L}{2\|xy\|}.$$

The next lemma states that the closest pair in the vertex set of $\mathfrak{C}$ can be used to obtain a 2-approximation to $\mathfrak{D}_{\mathfrak{C}}^{\max}$.

Lemma 4.3.2 *Let (p, q) be a closest pair in the vertex set of $\mathfrak{C}$. Then*

$$\mathfrak{D}_{\mathfrak{C}}^{\max} \le 2 \cdot \max_{0 \le i < n} \mathfrak{D}_{\Pi_i}(p, q).$$

Proof. Let j be an index such that $\mathfrak{D}_{\mathfrak{C}}^{\max} = \mathfrak{D}_{\Pi_j}$ and let x and y be two vertices of $\mathfrak{C}$ such that $\mathfrak{D}_{\Pi_j} = \mathfrak{D}_{\Pi_j}(x, y)$. Then

$$\mathfrak{D}_{\mathfrak{C}}^{\max} = \frac{\mathbf{d}_{\Pi_j}(x, y)}{\|xy\|} \le \frac{L}{\|pq\|}.$$

By Lemma 4.3.1, we have

$$\frac{L}{\|pq\|} \le 2 \cdot \max_{0 \le i < n} \mathfrak{D}_{\Pi_i}(p, q).$$

$\square$

Thus, by computing the closest pair (p, q) in the vertex set of $\mathfrak{C}$ and then using Lemma 4.3.1 to compute $\max_{0 \le i < n} \mathfrak{D}_{\Pi_i}(p, q)$, we obtain a 2-approximation to $\mathfrak{D}_{\mathfrak{C}}^{\max}$. We now show that a simple extension leads to an algorithm that computes the exact value of $\mathfrak{D}_{\mathfrak{C}}^{\max}$.

Let S be the set of all pairs (x, y) in the vertex set of $\mathfrak{C}$ for which $x \ne y$ and $\|xy\| \le 2\|pq\|$. The following lemma states that it suffices to consider the elements of S to compute $\mathfrak{D}_{\mathfrak{C}}^{\max}$.

Lemma 4.3.3 *We have*

$$\mathfrak{D}_{\mathfrak{C}}^{\max} = \max_{(x,y)\in S} \max_{0\le i<n} \mathfrak{D}_{\Pi_i}(x,y).$$

Proof. It is clear that

$$\mathfrak{D}_{\mathfrak{C}}^{\max} = \max_{x,\, y \text{ vertices of } \mathfrak{C}} \max_{0\le i<n} \mathfrak{D}_{\Pi_i}(x,y) \ge \max_{(x,y)\in S} \max_{0\le i<n} \mathfrak{D}_{\Pi_i}(x,y).$$

Let j be an index such that $\mathfrak{D}_{\mathfrak{C}}^{\max} = \mathfrak{D}_{\Pi_j}$ and let x and y be two vertices of $\mathfrak{C}$ such that $\mathfrak{D}_{\Pi_j} = \mathfrak{D}_{\Pi_j}(x,y)$. If we can show that $(x,y) \in S$ (i.e., $\|xy\| \le 2\|pq\|$), then the proof is complete. By Lemma 4.3.1, we have

$$\frac{L}{2\|pq\|} \le \max_{0\le i<n} \mathfrak{D}_{\Pi_i}(p,q).$$

It follows that

$$\begin{aligned}
\frac{L}{2\|pq\|} &\le \max_{0\le i<n} \mathfrak{D}_{\Pi_i}(p,q) \\
&\le \max_{x,\, y \text{ vertices of } \mathfrak{C}} \max_{0\le i<n} \mathfrak{D}_{\Pi_i}(x,y) \\
&= \mathfrak{D}_{\mathfrak{C}}^{\max} \\
&= \mathfrak{D}_{\Pi_j}(x,y) \\
&= \frac{\mathbf{d}_{\Pi_j}(x,y)}{\|xy\|} \\
&\le \frac{L}{\|xy\|}.
\end{aligned}$$

This implies that $\|xy\| \le 2\|pq\|$. ⊔⊓

The discussion above leads to Algorithm 4.3.1 for computing the value of $\mathfrak{D}_{\mathfrak{C}}^{\max}$.

By Lemma 4.3.1, each value computed in line 7 of Algorithm 4.3.1 is equal to $\max_{0\le i<n} \mathfrak{D}_{\Pi_i}(x,y)$. By Lemma 4.3.3, the largest of the values computed in line 9 is equal to $\mathfrak{D}_{\mathfrak{C}}^{\max}$. This proves the correctness of the algorithm. To analyze the running time of the algorithm, it is clear that lines 1–3 takes $\mathcal{O}(n)$ time. The closest-pair computation in line 4 takes $\mathcal{O}(n \log n)$ time; see [Smi00]. In [LS95], it is shown that the size of the set S is $\mathcal{O}(n)$. It is also shown there that if the points in the vertex set of $\mathfrak{C}$ are stored in two lists X and Y, where the points in X are sorted by x-coordinates and the points in Y are sorted by y-coordinates, and if there are cross-pointers between these two lists, then the set S can be computed in $\mathcal{O}(n)$ time. Therefore, line 5 takes $\mathcal{O}(n \log n)$ time. In lines 6–8, the algorithm spends $\mathcal{O}(1)$ time for each element of S. Since the size of S is $\mathcal{O}(n)$, the total time for lines 6–8 is $\mathcal{O}(n)$. Thus, the total time of the algorithm is $\mathcal{O}(n \log n)$.

Algorithm 4.3.1: Critical Edge

```
Input: A cycle $\mathfrak{C} = p_0, p_1, \ldots, p_{n-1}, p_0$.
Output: $\mathfrak{D}_{\mathfrak{C}}^{\max}$.
1 $\Delta(p_0) := 0$;
2 for $i := 1, 2, \ldots, n-1$ do  $\Delta(p_i) := \Delta(p_{i-1}) + \|p_{i-1}p_i\|$;
3 $L := \Delta(p_{n-1}) + \|p_{n-1}p_0\|$;
4 $(p, q) :=$ the closest pair in the vertex set of $\mathfrak{C}$;
5 $S :=$ the set of all pairs $(x, y)$ in the vertex set of $\mathfrak{C}$ for which $x \neq y$ and
  $\|xy\| \leq 2\|pq\|$;
6 foreach  $(x, y) \in S$ do
7     $t_{x,y} := \max(|\Delta(x) - \Delta(y)|, L - |\Delta(x) - \Delta(y)|)/\|xy\|$;
8 end
9 return $\max_{x,y} t_{x,y}$;
```

If the cycle $\mathfrak{C}$ is a convex polygon, then we can improve the running time: In [LP78], it is shown that the closest pair can be computed in $\mathcal{O}(n)$ time. Since $\mathfrak{C}$ is a convex polygon, we can obtain the lists X and Y in $\mathcal{O}(n)$ time. It follows that the entire algorithm runs in $\mathcal{O}(n)$ time.

The following observations lead to an alternative $\mathcal{O}(n)$–time algorithm for the case when $\mathfrak{C}$ is a convex polygon. The Delaunay triangulation DT of the vertex set of $\mathfrak{C}$ can be computed in $\mathcal{O}(n)$ time, see [AGSS89]. This implies that the closest pair can be computed in the same time bound. We show that DT can be used to compute the set S in $\mathcal{O}(n)$ time. A proof of the following lemma can be found in [DDS92].

Lemma 4.3.4 *Consider the Delaunay triangulation DT of the vertex set of $\mathfrak{C}$, and let x and y be two distinct vertices that are not connected by an edge in DT. Then there exists a path $(x = x_0, x_1, x_2, \ldots, x_k = y)$ in DT, such that*

1. *for each i with $0 \leq i < k$, $\|x_i x_{i+1}\| < \|xy\|$ and*
2. *for each i with $0 \leq i \leq k$, $\|xx_i\| < \|xy\|$.*

Let (p, q) be the closest pair in the vertex set of $\mathfrak{C}$ and let $d := 2\|pq\|$. Let DT' be the subgraph of DT consisting of all edges having length at most d. Lemma 4.3.4 implies that we obtain the set S (i.e., all pairs of points whose distance is at most d) by performing a BFS from each vertex x of DT until we reach a vertex y such that $\|xy\| > d$. The total time for this is proportional to the size of S, which we know to be $\mathcal{O}(n)$.

Theorem 4.3.5 *Given a polygonal cycle $\mathfrak{C}$ on n vertices in the plane, we can compute $\mathfrak{D}_{\mathfrak{C}}^{\max}$ in $\mathcal{O}(n \log n)$ time. If $\mathfrak{C}$ is a convex polygon, $\mathfrak{D}_{\mathfrak{C}}^{\max}$ can be computed in $\mathcal{O}(n)$ time.*

4.4 $(1+\varepsilon)$-Approximation algorithm

Consider again the polygonal cycle $\mathfrak{C} = (p_0, \ldots, p_{n-1}, p_0)$ whose vertices are points in the plane. Let $\varepsilon > 0$ be a constant. In this section, we show that an approximation to the dilation of *each* path Π_i $(0 \le i < n)$, as well as an approximation to $\mathfrak{D}_{\mathfrak{C}}^{\min}$, can be computed in $\mathcal{O}(n \log n)$ total time. We use Fact 3.2.2 to approximate the dilation of the paths. By Fact 3.2.2, in order to approximate the dilation of a Euclidean graph, it is sufficient to compute the dilation between $\mathcal{O}(n)$ pairs of vertices. Moreover, the choice of these vertices depends only on the vertex set of the graph, it does not depend on the edges of the graph.

In a preprocessing step, we compute a WSPD $\{(A_j, B_j)\}_{j=1}^m$, where $m = \mathcal{O}(n)$, for the vertex set $\{p_0, \ldots, p_{n-1}\}$ of the cycle $\mathfrak{C}$, with respect to $s := 4(2+\varepsilon)/\varepsilon$. We can do this in $\mathcal{O}(n \log n)$ time. For each j with $1 \le j \le m$, we pick an arbitrary point a_j in A_j, and an arbitrary point b_j in B_j. Our algorithm will compute, for each i with $0 \le i < n$, the value

$$t_i := \max_{1 \le j \le m} \mathfrak{D}_{\Pi_i}(a_j, b_j).$$

Recall that, by Fact 3.2.2, $\mathfrak{D}_{\Pi_i}/(1+\varepsilon) \le t_i \le \mathfrak{D}_{\Pi_i}$, which means that t_i is a $(1+\varepsilon)$-approximation of $\mathfrak{D}_{\Pi_i}$.

Lemma 4.4.1 *Let* $t^* := \min(t_0, t_1, \ldots, t_{n-1})$. *Then*

$$\frac{\mathfrak{D}_{\mathfrak{C}}^{\min}}{(1+\varepsilon)} \le t^* \le \mathfrak{D}_{\mathfrak{C}}^{\min}.$$

Proof. Let i be an index such that $t^* = t_i$ and let j be an index such that $\mathfrak{D}_{\mathfrak{C}}^{\min} = \mathfrak{D}_{\Pi_j}$. Then

$$t^* = t_i \le t_j \le \mathfrak{D}_{\Pi_j} = \mathfrak{D}_{\mathfrak{C}}^{\min}$$

and

$$\mathfrak{D}_{\mathfrak{C}}^{\min} = \mathfrak{D}_{\Pi_j} \le \mathfrak{D}_{\Pi_i} \le (1+\varepsilon) t_i = (1+\varepsilon) t^*.$$

$\square$

We remark that, by a similar argument, the value $t^{**} := \max(t_0, t_1, \ldots, t_{n-1})$ can be shown to satisfy

$$\frac{\mathfrak{D}_{\mathfrak{C}}^{\max}}{(1+\varepsilon)} \le t^{**} \le \mathfrak{D}_{\mathfrak{C}}^{\max}.$$

In other words, the algorithm that will be presented below can also be used to compute an approximation to $\mathfrak{D}_{\mathfrak{C}}^{\max}$ in $\mathcal{O}(n \log n)$ time. We have seen in Section 4.3, however, that the exact value of $\mathfrak{D}_{\mathfrak{C}}^{\max}$ can be computed within the same time bound.

As mentioned above, our algorithm computes t_i for $i = 0, 1, \ldots, n-1$. The naïve algorithm takes $\mathcal{O}(n^2)$ time to compute the approximation because we test $\mathcal{O}(n)$ candidate edges and for each one we spend $\mathcal{O}(n)$ time to approximate the dilation of the path. But we can improve the running time using the following algorithm.

The main idea is to maintain, for the current value of i, the m dilations $\mathfrak{D}_{\Pi_i}(a_j, b_j)$ $(1 \leq j \leq m)$ in a balanced binary search tree $\mathcal{T}$. Observe that, for any fixed index j, the value of $\mathfrak{D}_{\Pi_i}(a_j, b_j)$ changes at most twice when i is increased from 0 to $n-1$. More precisely, when we move from testing the edge (p_{i-1}, p_i) to testing the edge (p_i, p_{i+1}), the value of $\mathfrak{D}_{\Pi_i}(a_j, b_j)$ is different from $\mathfrak{D}_{\Pi_{i-1}}(a_j, b_j)$ only if $a_j = p_i$ or $b_j = p_i$. In other words, when we test (p_i, p_{i+1}), the dilation of all pairs remains the same except the pairs for which one of the points in the pair is the same as p_i. As a result, the total number of updates in $\mathcal{T}$ will be at most $2m$. So to maintain the search tree $\mathcal{T}$, in total, we need $\mathcal{O}(m \log m)$ time. For the details of the algorithm see Algorithm 4.4.1.

Now we show the correctness and compute the time complexity of the algorithm. Let Π denote the path $(p_0, p_1, \ldots, p_{n-1})$. Recall the relation $<_\Pi$ of Section 4.2.1. We may assume without loss of generality that $a_j <_\Pi b_j$ for each j with $1 \leq j \leq m$.

In the preprocessing steps, we compute, in $\mathcal{O}(n)$ time, the values

$$\Delta(p_i) = \mathbf{d}_\Pi(p_0, p_i) \qquad (0 \leq i < n)$$

and the total length L of the cycle $\mathfrak{C}$. Observe that, for $0 \leq i < n$, the distance $\mathbf{d}_{\Pi_i}(a_j, b_j)$ between a_j and b_j in the path Π_i satisfies

$$\mathbf{d}_{\Pi_i}(a_j, b_j) = \begin{cases} \Delta(b_j) - \Delta(a_j) & \text{if } i = n-1 \text{ or } p_i <_\Pi a_j \text{ or } b_j <_\Pi p_{i+1}, \\ L - (\Delta(b_j) - \Delta(a_j)) & \text{otherwise.} \end{cases} \tag{4.3}$$

Using (4.3) we can compute the dilation between each pair of points in constant time. Then, in $\mathcal{O}(n \log n)$ time, we can compute the WSPD and select the representatives for each pair in the WSPD. In the final part of the preprocessing step, we compute, for each i with $0 < i < n$, the set

$$S_i := \{j \mid 1 \leq j \leq m, a_j = p_i \text{ or } b_j = p_i\}.$$

Obviously, all these sets can be computed in $\mathcal{O}(m) = \mathcal{O}(n)$ time. Note that S_i contains all indices j such that the pairs (a_j, b_j) should be updated in step i.

The correctness of the algorithm follows from Lemma 4.4.1 and the discussion above. We have seen already that the preprocessing step, i.e. lines 1–10, takes $\mathcal{O}(n \log n)$ time. Lines 11–16 take $\mathcal{O}(m \log m) = \mathcal{O}(n \log n)$ time. The total time for lines 17–25 is proportional to $\sum_{i=1}^{n-1} (|S_i| + 1) \log m \leq (2m + n) \log m = \mathcal{O}(n \log n)$.

Algorithm 4.4.1: Approximate Minimal Deletion

```
Input: Cycle $\mathfrak{C} = p_0, p_1, \ldots, p_{n-1}, p_0$ and a real constant $\varepsilon > 0$.
Output: Path $\Pi = \mathfrak{C} \setminus \{e\}$.
$\Delta(p_0) := 0$;
for $i := 1, 2, \ldots, n-1$ do  $\Delta(p_i) := \Delta(p_{i-1}) + \|p_{i-1}p_i\|$;
$L := \Delta(p_{n-1}) + \|p_{n-1}p_0\|$;
$\{(A_i, B_i)\}_{i=1}^m :=$ WSPD of the point set $P := \{p_0, \ldots, p_{n-1}\}$ with respect to
    $s = 4(2+\varepsilon)/\varepsilon$;
for $i := 1, \ldots, m$ do
    Select arbitrary points $a_i \in A_i$ and $b_i \in B_i$;
    if $a_i \not<_\Pi b_i$ then switch $a_i$ and $b_i$;
end
for $i := 1, 2, \ldots, n-1$ do  $S_i := \{j \mid 1 \le j \le m, a_j = p_i$ or $b_j = p_i\}$;
Initialize an empty balanced binary search tree $\mathcal{T}$ (e.g., a red-black tree);
for $j = 1, 2, \ldots, m$ do
    $D_j := \mathfrak{D}_{\Pi_0}(a_j, b_j)$;                         /* Using (4.3) */
    Insert $D_j$ into $\mathcal{T}$;
end
$t_0 :=$ the maximum element in the tree $\mathcal{T}$;
$t = t_0$; $e = (p_0, p_1)$;
for $i = 1, 2, \ldots, n-1$ do
    foreach $j \in S_i$ do
        Delete $D_j$ from the tree $\mathcal{T}$;
        $D_j := \mathfrak{D}_{\Pi_i}(a_j, b_j)$;                     /* Using (4.3) */
        Insert $D_j$ into $\mathcal{T}$;
    end
    $t_i :=$ the maximum element in the tree $\mathcal{T}$;
    if $t_i < t$ then $t := t_i$; $e := (p_i, p_{i+1})$;
end
return $\mathfrak{C} \setminus \{e\}$;
```

Theorem 4.4.2 *Given a polygonal cycle $\mathfrak{C} = (p_0, \ldots, p_{n-1}, p_0)$ on n vertices in the plane and a constant $\varepsilon > 0$, in $\mathcal{O}(n \log n)$ time, we can compute a sequence $t_0, \ldots, t_{n-1}, t^*$ of real numbers, such that*

$$\frac{\mathfrak{D}_{P_i}}{(1+\varepsilon)} \le t_i \le \mathfrak{D}_{P_i}$$

for each $i = 0, 1, \ldots, n-1$ and

$$\frac{\mathfrak{D}_{\mathfrak{C}}^{\min}}{(1+\varepsilon)} \le t^* \le \mathfrak{D}_{\mathfrak{C}}^{\min}.$$

4.5 Concluding remarks

Recently, there has been a fair amount of work on the problem of computing the optimal dilation of a given (geometric) graph [CHL07, EW07, GKM07, KK07]. In this chapter we considered a variation of the problem where we are given a polygonal cycle and are supposed to choose one edge to remove such that the resulting polygonal path gives the smallest (or the largest) possible dilation.

A straightforward and challenging open problem is to generalize the results to more general Euclidean graphs. In that case it is also interesting to see how we can generalize the results when we want to remove more than one edge.

Acknowledgments

We would like to thank Hee-Kap Ahn, Christian Knauer, Michiel Smid and Yajun Wang, the co-authors on this chapter and Jan Vahrenhold for fruitful discussions on the subject.

Chapter 5

Computing Spanner Diameter

5.1 Introduction

The diameter of an unweighted undirected graph is the length of the longest shortest path. When studying the diameter of a graph, the length of the path is usually the number of edges/links along the path. More precisely, for a graph $G(V, E)$, the diameter of G, denoted by $\mathbf{diam}(G)$, is equal to $\max_{x,y \in V} d_G(x, y)$ where $d_G(x, y)$ is the shortest path length between x and y in G. To compute the diameter of a graph G, it is sufficient to compute all-pairs-shortest paths which can be done in $\mathcal{O}(mn + n^2 \log n)$ time using $\mathcal{O}(m + n)$ space where n is the number of vertices and m is the number of edges of the input graph.

Now suppose we have a geometric t-spanner. Then normally we would measure the length of a path as the sum of its edges lengths. We would then define the diameter of the spanner as the maximum distance between any two points. This, however, is not very useful: in a geometric t-spanner, the diameter is at most t times the geometric diameter and since t is small (say $(1 + \varepsilon)$ for some ε close to 0) this is not very informative. Sometimes, however, the number of links on a path also plays a role. Consider for instance the network of some airline. We would like to have a short connection between every pair of airports, but we would also like to have as few stopovers as possible. Also in applications involving wireless ad hoc networks it is often desirable to have small spanner diameter since it determines the maximum number of times a message has to be transmitted in a network. This leads us to the following definition of diameter where we consider the number of links on a path as its length, but we restrict our attention to paths that are relatively short in the weighted sense. More formally, the diameter is defined as follows:

Definition 5.1.1 *The diameter (or spanner diameter) of a t-spanner $G(V, E)$ is the smallest integer $\mathbb{D}$ such that for every pair of vertices u and v in V, there exists a t-path between u and v in G which contains at most $\mathbb{D}$ edges.*

In other words, we have both an upper bound on the path length and an upper bound on the number of links on the path.

As far as we know there is no known algorithm to compute the spanner diameter of a t-spanner. In this chapter we present a dynamic programming approach. In Chapter 6 we will study this quality criterion when building t-spanners.

In this chapter we assume that $G(V, E)$ is a t-spanner of V where $(V, \mathbf{d})$ is a metric space. Recall that a t-path between two vertices u and v in G is a path of length at most $t \cdot \mathbf{d}(u, v)$.

5.2 Dynamic programming approach

Let $G(V, E)$ be a t-spanner of V. For each p and q in V, let $\delta_G^{\leq k}(p, q)$, or $\delta^{\leq k}(p, q)$ if the graph is clear from the context, denote the shortest path between p and q in G with at most k edges, if such a path exists. In the case when we have more than one such path we consider the path with the smallest number of edges. Also assume that $\mathbf{d}_G^{\leq k}(p, q)$, or for short $\mathbf{d}^{\leq k}(p, q)$, is the length of $\delta^{\leq k}(p, q)$. If there is no path between p and q which contains at most k links then we set $\mathbf{d}^{\leq k}(p, q)$ to ∞. For simplicity we set $\mathbf{d}^{\leq k}(p, p) = 0$ for each vertex $p \in V$ and every integer k. Note that if $\mathbf{d}^{\leq k}(p, q) \leq t \cdot \mathbf{d}(p, q)$ then there exists a t-path between p and q which contains at most k links. So by Definition 5.1.1, the diameter of G is the smallest integer k such that $\mathbf{d}^{\leq k}(p, q) \leq t \cdot \mathbf{d}(p, q)$, for every pair of vertices p and q in V.

To compute the diameter of a t-spanner it is sufficient to compute $\mathbf{d}^{\leq k}$ for every k and for every pair of vertices in the graph. Obviously for each pair p and q in V we have

$$\mathbf{d}^{\leq k}(p, q) = \min_{\binom{i+j=k}{r \in V}} \left\{ \mathbf{d}^{\leq i}(p, r) + \mathbf{d}^{\leq j}(r, q) \right\}.$$

In the next lemma we will show how to decrease the number of paths we need to examine.

Lemma 5.2.1 *Let $G(V, E)$ be a graph, let p and q be any two vertices of G and let $k \geq 2$ be an integer. If i' and j' are two arbitrary positive integers such that*

$i' + j' = k$ then

$$\min_{\binom{i+j=k}{r\in V}} \left\{\mathbf{d}^{\leq i}(p,r) + \mathbf{d}^{\leq j}(r,q)\right\} = \min_{r\in V} \left\{\mathbf{d}^{\leq i'}(p,r) + \mathbf{d}^{\leq j'}(r,q)\right\}.$$

Proof. Obviously the left-hand side of the equality is at most equal to the right-hand side. To prove the opposite inequality, assume i and j are two arbitrary positive integers such that $i + j = k$. The claim is that for each summation $\mathbf{d}^{\leq i}(p,r) + \mathbf{d}^{\leq j}(r,q)$ there exist $r' \in V$ such that

$$\mathbf{d}^{\leq i}(p,r) + \mathbf{d}^{\leq j}(r,q) \geq \mathbf{d}^{\leq i'}(p,r') + \mathbf{d}^{\leq j'}(r',q).$$

Without loss of generality we assume $i' < i$. So $j' > j$ and for each pair of vertices (x,y) we have $\mathbf{d}^{\leq j'}(x,y) \leq \mathbf{d}^{\leq j}(x,y)$. Now there are two cases:

Case 1: $\delta^{\leq i}(p,r)$ contains at most i' edges. In this case we are done because $\mathbf{d}^{\leq i}(p,r) = \mathbf{d}^{\leq i'}(p,r)$ and $\mathbf{d}^{\leq j}(r,q) \geq \mathbf{d}^{\leq j'}(r,q)$.

Case 2: $\delta^{\leq i}(p,r)$ contains more than i' edges. Let r' be the i'th node in $\delta^{\leq i}(p,r)$ starting from p and let Π be the sub-path of $\delta^{\leq i}(p,r)$ between r' and r, see Figure 5.1 for an illustration. Note that Π is the shortest path between r' and r with at most $i - i'$ edges and

$$\mathbf{d}^{\leq i}(p,r) = \mathbf{d}^{\leq i'}(p,r') + \mathbf{d}^{\leq i-i'}(r',r).$$

Now let Π' be the path generated by concatenating Π and $\delta^{\leq j}(r,q)$. Clearly Π'

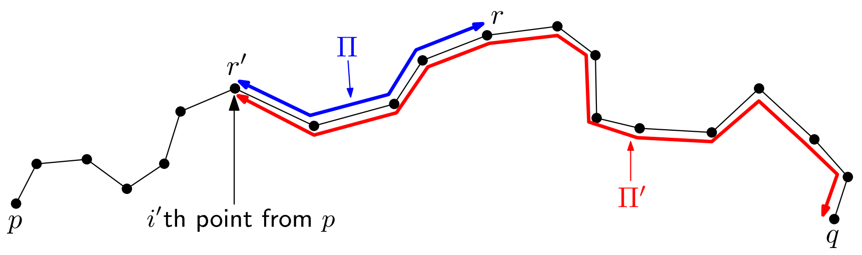


Figure 5.1: Illustrating the proof of Lemma 5.2.1.

is a path between r' and q which contains at most $i - i' + j$ edges. Therefore

$$\begin{aligned}\mathbf{d}^{\leq i}(p,r) + \mathbf{d}^{\leq j}(r,q) &= \mathbf{d}^{\leq i'}(p,r') + \mathbf{d}^{\leq i-i'}(r',r) + \mathbf{d}^{\leq j}(r,q)\\ &\geq \mathbf{d}^{\leq i'}(p,r') + \mathbf{d}^{\leq i-i'+j}(r',q)\\ &= \mathbf{d}^{\leq i'}(p,r') + \mathbf{d}^{\leq j'}(r',q).\end{aligned}$$

The claim and, hence, the lemma follows. ⊔⊓

By replacing i' by $k-1$ and j' by 1 we have the following corollary.

Corollary 5.2.2 *Let $G(V,E)$ be a graph and let p and q be two vertices of G. For every integer $k \geq 2$ we have*

$$\mathbf{d}^{\leq k}(p,q) = \min_{r \in V} \left\{ \mathbf{d}^{\leq k-1}(p,r) + \mathbf{d}^{\leq 1}(r,q) \right\}.$$

To compute the diameter of a t-spanner G, we compute $\mathbf{d}^{\leq k}(p,q)$ for each k using Corollary 5.2.2. We start with $k=1$. Obviously for all edges (p,q) in the graph G, $\mathbf{d}^{\leq 1}(p,q) = \mathbf{d}(p,q)$ and $\mathbf{d}^{\leq 1}(p,q) = \infty$ otherwise. Then we can compute $\mathbf{d}^{\leq k}$ using $\mathbf{d}^{\leq 1}$ and $\mathbf{d}^{\leq k-1}$. We continue constructing $\mathbf{d}^{\leq k}$, for $k = 2, 3, \ldots$, until for every pair of vertices (p,q), the value of $\mathbf{d}^{\leq k}(p,q)$ is at most $t \cdot \mathbf{d}(p,q)$. For details see Algorithm 5.2.1.

```
Algorithm 5.2.1: DIAMETER
   Input: $G(V,E)$ and $t > 1$.
   Output: Spanner diameter of $G(V,E)$.
 1 foreach $(p,q) \in V^2$ do
 2     if $(p,q) \in E$ then $\mathbf{d}^{\leq 1}(p,q) := \mathbf{d}(p,q)$;
 3     else $\mathbf{d}^{\leq 1}(p,q) := \infty$;
 4 end
 5 $k := 1$;
 6 flag:=true;
 7 while flag =true do
 8     $k := k+1$;
 9     flag:=false;
10     foreach $(p,q) \in V^2$ do
11         $\mathbf{d}^{\leq k}(p,q) := \min_{r \in V} \left\{ \mathbf{d}^{\leq k-1}(p,r) + \mathbf{d}^{\leq 1}(r,q) \right\}$;
12         if $\mathbf{d}^{\leq k}(p,q) > t \cdot \mathbf{d}(p,q)$ then flag:=true;
13     end
14 end
15 return $k$;
```

The following corollary follows immediately from Algorithm 5.2.1.

Corollary 5.2.3 *The diameter of a t-spanner G with n vertices can be computed in $\mathcal{O}(\mathbb{D} \cdot n^3)$ time using $\mathcal{O}(n^2)$ space, where $\mathbb{D}$ is the diameter of G.*

We can improve the running time to $\mathcal{O}(\log(\mathbb{D}) \cdot n^3)$ using the following trick. In Algorithm 5.2.1 we compute $\mathbf{d}^{\leq k+1}$ using $\mathbf{d}^{\leq k}$ and $\mathbf{d}^{\leq 1}$. By Lemma 5.2.1 we can compute $\mathbf{d}^{\leq 2k}$ using $\mathbf{d}^{\leq k}$. So instead of computing $\mathbf{d}^{\leq k}$, for every k between 1 and

$\mathbb{D}$, we compute $\mathbf{d}^{\leq 2^i}$ for $0 < i \leq \lceil \log \mathbb{D} \rceil$. Clearly for $i = \lceil \log(\mathbb{D}) \rceil$, we have a t-path with at most i links between every pair of vertices. To find the exact diameter, we perform a binary search on the interval $(2^{i-1}, 2^i]$. More precisely, we compute $\mathbf{d}^{\leq k}$ for $k = \frac{2^i + 2^{i-1}}{2}$, which is the middle point of the interval $(2^{i-1}, 2^i]$, using $\mathbf{d}^{\leq 2^{i-1}}$ and $\mathbf{d}^{\leq 2^{i-2}}$ which have already been computed. If for all pairs (p, q) the value of $\mathbf{d}^{\leq k}(p, q)$ is at most t times $\mathbf{d}(p, q)$ then we limit the new interval to $(2^{i-1}, k]$ and continue the search. Otherwise, we continue the search on $(k, 2^i]$. We have the diameter when the length of the searching interval is 1. See Algorithm 5.2.2 for more details.

Algorithm 5.2.2: DIAMETER2

```
Input: $G(V, E)$ and $t > 1$.
Output: Spanner diameter of $G(V, E)$.
foreach $(p, q) \in V^2$ do
    if $(p, q) \in E$ then $\mathbf{d}^{\leq 1}(p, q) := \mathbf{d}(p, q)$;
    else $\mathbf{d}^{\leq 1}(p, q) := \infty$;
end
$k := 0$;
flag:=true;
while flag =true do
    flag:=false;
    $k := k + 1$;
    foreach $(p, q) \in V^2$ do
        $\mathbf{d}^{\leq 2^k}(p, q) := \min_{r \in V} \left\{ \mathbf{d}^{\leq 2^{k-1}}(p, r) + \mathbf{d}^{\leq 2^{k-1}}(r, q) \right\}$;
        if $\mathbf{d}^{\leq 2^k}(p, q) > t \cdot \mathbf{d}(p, q)$ then flag:=true;
    end
end
$I_b := 2^{i-1}$; $I_e := 2^i$;
while $I_e \neq (I_b + 1)$ do
    $I_m := (I_b + I_e)/2$; $j := I_b - I_m$;
    foreach $(p, q) \in V^2$ do
        $\mathbf{d}^{\leq I_m}(p, q) := \min_{r \in V} \left\{ \mathbf{d}^{\leq I_b}(p, r) + \mathbf{d}^{\leq j}(r, q) \right\}$;
        if $\mathbf{d}^{\leq I_m}(p, q) > t \cdot \mathbf{d}(p, q)$ then flag:=true;
    end
    if flag = true then $I_b := I_m$ else $I_e := I_m$;
end
return $I_e$;
```

It is easy to see that Algorithm 5.2.2 needs to compute $\mathbf{d}^{\leq k}$ for $\mathcal{O}(\log(\mathbb{D}))$ values of k which means the time complexity of the algorithm is $\mathcal{O}(\log(\mathbb{D}) \cdot n^3)$. Note that the space complexity of the algorithm increases to $\mathcal{O}(\log(\mathbb{D}) \cdot n^2)$ since we need to save all the distances $\mathbf{d}^{\leq 2^i}$ generated in the first loop of the algorithm. The following corollary immediately follows from Algorithm 5.2.2.

Corollary 5.2.4 *The diameter of a t-spanner G with n vertices can be computed in $\mathcal{O}(\log(\mathbb{D}) \cdot n^3)$ time using $\mathcal{O}(\log(\mathbb{D}) \cdot n^2)$ space, where $\mathbb{D}$ is the diameter of G.*

5.3 Improving the complexity bounds

With some modifications we can reduce the running time to $\mathcal{O}(\mathbb{D} \cdot mn)$ which is much faster for t-spanners with a small number of edges. On line 11 of Algorithm 5.2.1, we compute the minimum over all $r \in V$, but if r is not adjacent to q, then $\mathbf{d}^{\leq 1}(r, q) = \infty$ and obviously we do not need to check r. In the modification, to compute $\mathbf{d}^{\leq k}(p, q)$, we compute the minimum over the vertices which are adjacent to q. Therefore for a pair (p, q) we need $\mathcal{O}(\deg(q))$ time to compute the minimum, where $\deg(q)$ is the degree of q. Therefore the total running time is

$$\begin{aligned}
\mathcal{O}\left(\mathbb{D} \cdot \sum_{(p,q)\in V^2} \deg(q)\right) &= \mathcal{O}\left(\mathbb{D} \cdot \sum_{p\in V} \sum_{q\in V} \deg(q)\right) \\
&= \mathcal{O}\left(\mathbb{D} \cdot \sum_{p\in V} 2m\right) \\
&= \mathcal{O}(\mathbb{D} \cdot mn).
\end{aligned}$$

Algorithm 5.3.1 uses $\mathcal{O}(n^2)$ space since the path lengths are stored in a matrix. However, by processing one node at a time, we can replace the matrix by an array. More precisely, for each vertex $p \in V$, we compute the *radius* of G from p. The radius of G from a node p is defined as the minimum integer $\mathbb{D}_p$ such that for any node $q \in V$, there is a t-path between p and q in G which contains at most $\mathbb{D}_p$ edges. After all the vertices have been processed, the largest radius will be the diameter of G.

Theorem 5.3.1 *The diameter of a t-spanner G with n nodes and m edges can be computed in $\mathcal{O}(\mathbb{D} \cdot mn)$ time using $\mathcal{O}(n)$ space, where $\mathbb{D}$ is the diameter of G.*

Algorithm 5.3.1: DIAMETER3

```
Input: $G(V,E)$ and $t > 1$.
Output: Spanner diameter of $G(V,E)$.
foreach $(p,q) \in V^2$ do
    if $(p,q) \in E$ then $\mathbf{d}^{\leq 1}(p,q) := \mathbf{d}(p,q)$;
    else $\mathbf{d}^{\leq 1}(p,q) := \infty$;
end
$k := 1$;
flag:=true;
while flag =true do
    $k := k+1$;
    flag:=false;
    foreach $(p,q) \in V^2$ do
        $\mathbf{d}^{\leq k}(p,q) := \min_{r \text{ is adjacent to } q} \left\{\mathbf{d}^{\leq k-1}(p,r) + \mathbf{d}(r,q)\right\}$;
        if $\mathbf{d}^{\leq k}(p,q) > t \cdot \mathbf{d}(p,q)$ then flag:=true;
    end
end
return $k$;
```

5.4 A final approach

In Algorithm 5.3.1, we updated $\mathbf{d}^{\leq k}$ for every pair of vertices p and q in V. This was done by computing $\mathbf{d}^{\leq k-1}(p,r)+\mathbf{d}^{\leq 1}(r,q)$ for every vertex $r \in V$ adjacent to q. However, the shortest path between p and r of *at most* $k-1$ edges, $\delta^{\leq k-1}(p,r)$, may contain less than $k-1$ edges. We observe that we do not need to check the path in this case since this will give us the shortest path between p and q containing at most $k-1$ edges – which already has been computed.

In this section we process all pairs (p,q) such that $\delta^{\leq k-1}(p,q)$ contains exactly $k-1$ edges instead of checking all pairs. It is obvious that the rest of the entries of $\mathbf{d}^{\leq k}$ are the same as the corresponding entries in $\mathbf{d}^{\leq k-1}$.

To do this, let S_{k-1} be the set of all pairs $(p,q) \in V^2$ such that $\delta^{\leq k-1}(p,q)$ contains exactly $k-1$ links. For each pair $(p,q) \in S_{k-1}$ we check whether we can add an edge to $\delta^{\leq k-1}(p,q)$, from one of its endpoints, such that the resulting path forms a shorter path between its endpoints. More precisely, for each pair $(p,q) \in S_{k-1}$ and for every adjacent node of p, say r, we check if there is a shorter path between r and q via p. In the case that $\mathbf{d}^{\leq k}(q,r) > \mathbf{d}^{\leq k-1}(p,q) + \mathbf{d}(p,r)$, we update $\mathbf{d}^{\leq k}(q,r)$ and add (q,r) to S_k. We do the same for all adjacent nodes of q. For more details see Algorithm 5.4.1.

In the worst case the set S_k may contain all the pairs of vertices which means we need $\mathcal{O}(mn)$ time to compute $\mathbf{d}^{\leq k}$. This gives us an algorithm, called DIAMETER4, with running time $\mathcal{O}(\mathbb{D}mn)$, and space complexity $\mathcal{O}(n^2)$. Note that the space complexity can be improved to $\mathcal{O}(n)$, since the vertices can be processed iteratively. We denote this space improved version of DIAMETER4 by DIAMETER4i. Even thought the complexity bounds of DIAMETER4 and DIAMETER4i are identical to the bounds of the algorithm in Section 5.3 we believe that this approach will be more efficient in practice. This is also supported by the experiments performed in the next section.

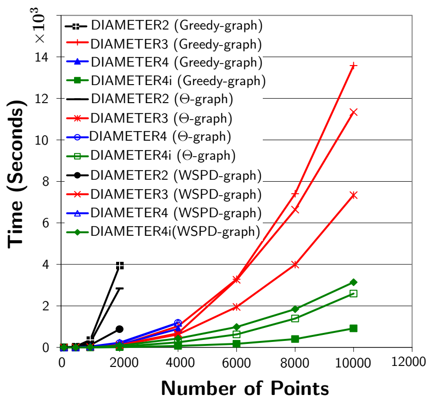
Algorithm 5.4.1: DIAMETER4

```
Input: $G(V,E)$ and $t > 1$.
Output: Spanner diameter of $G(V,E)$.
foreach $(p,q) \in V^2$ do
    if $(p,q) \in E$ then
        $\mathbf{d}^{\leq 1}(p,q) := \mathbf{d}(p,q)$;
        Add $(p,q)$ to $S_1$;
    else $\mathbf{d}^{\leq 1}(p,q) := \infty$;
end
$k := 1$; flag:=true;
while flag =true do
    $k := k+1$;
    foreach $(p,q) \in V^2$ do $\mathbf{d}^{\leq k}(p,q) := \mathbf{d}^{\leq k-1}(p,q)$;
    flag:=false;
    foreach $(p,q) \in S_{k-1}$ do
        foreach vertex $r$ adjacent to $p$ do
            if $\mathbf{d}^{\leq k}(r,q) > \mathbf{d}^{\leq k-1}(p,q) + \mathbf{d}(p,r)$ then
                $\mathbf{d}^{\leq k}(r,q) := \mathbf{d}^{\leq k-1}(p,q) + \mathbf{d}(p,r)$;
                Add $(r,q)$ to $S_k$;
                if $\mathbf{d}^{\leq k}(r,q) > t \cdot \mathbf{d}(r,q)$ then flag:=true;
            end
        end
        foreach vertex $r$ adjacent to $q$ do
            if $\mathbf{d}^{\leq k}(r,p) > \mathbf{d}^{\leq k-1}(p,q) + \mathbf{d}(q,r)$ then
                $\mathbf{d}^{\leq k}(r,p) := \mathbf{d}^{\leq k-1}(p,q) + \mathbf{d}(q,r)$;
                Add $(r,p)$ to $S_k$;
                if $\mathbf{d}^{\leq k}(r,p) > t \cdot \mathbf{d}(r,p)$ then flag:=true;
            end
        end
    end
end
return $k$;
```

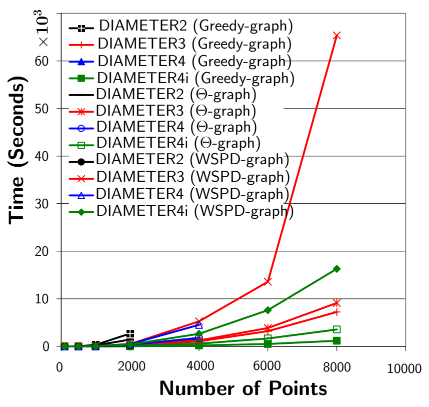
5.5 Experimental results

We implemented DIAMETER2, DIAMETER3, DIAMETER4 and DIAMETER4i—the version of DIAMETER4 which use linear space—and compared their running times on three different t-spanners on planar point sets: greedy-graphs, Θ-graphs and WSPD-graphs. These spanners range from very sparse graphs with constant degree to much denser graphs with linear degree. The theoretical diameter of the spanners are also different. The properties of these spanners are discussed in detail in Chapter 6. The experiments were done on uniformly distributed point sets range from 100 to 10,000 points.

Algorithm DIAMETER2 (Algorithm 5.2.2) is the slowest algorithm as its $\mathcal{O}(\log(\mathbb{D}) \cdot n^3)$ theoretical bound shows. Since its running time depends on $\mathbb{D}$ it performs better for graphs with small diameter, see Figure 5.2. For example, for a t-spanner with 2K vertices, DIAMETER2 needs about 4K seconds to compute the diameter of a greedy-graph. This time decreases to 3K seconds for Θ-graphs and 1K seconds for WSPD-graphs. Note that diameter of greedy-graphs, Θ-graphs and WSPD-graphs are 102, 26 and 3 respectively. The experiments also show that the running time of DIAMETER2 is independent of the number of edges of the input graph.



(a) Uniform distribution, $t = 2$.



(b) Uniform distribution, $t = 1.1$.

Figure 5.2: Comparing the running times of the diameter algorithms for different graphs.

For DIAMETER3 the experiments show that the number of edges is the most important variable in the running time. For example for a 2-spanner on a set of 4K points, DIAMETER3 needs 1000 seconds to compute the diameter of greedy-graphs and the time decreases to roughly 600 seconds for Θ-graphs and WSPD-graphs. For smaller values of t, this order changes: for $t = 1.1$, the algorithm needs 5K seconds to compute the diameter of WSPD-graphs and this decreases to 1000 seconds for greedy-graphs and Θ-graphs. Note that, in general, when we

	Diameter2			Diameter3			Diameter4			Diameter4i		
	Greedy-graph	Θ-graph	WSPD-graph	Greedy-graph	Θ-graph	WSPD-graph	Greedy-graph	Θ-graph	WSPD-graph	Greedy-graph	Θ-graph	WSPD-graph
100	0.2	0.1	0.03	0.02	0.03	0.03	0.04	0.06	0.07	0.01	0.02	0.04
500	31	9.9	4	1.5	3.1	7.5	4	6.8	10.8	0.48	2.1	10
1000	327	195	59.3	12.3	29.6	69.7	31.4	52.2	83.3	3.1	15.3	67
2000	2702	1445	254	118	196	471	226	301	484	23	95	413
4000				1057	1256	5265	1544	1822	4548	200	576	2635
6000				3196.5	3848	13571				510.5	1654	7603
8000				7228	9131	65360				1192	3552	16301

Table 5.1: The running of diameter algorithms (in seconds) on 1.1-spanners on uniformly distributed point sets.

decrease t, the diameter decreases and the size of the graph increases. As one can see in Figure 5.2, by decreasing t, the running time of Diameter3 improved slightly on WSPD-graphs but the improvement is considerable for greedy-graphs and Θ-graphs.

Algorithm Diameter4 has the same behavior as Diameter3 and shows a small improvement when we have very dense graphs. However the improvement on the $\mathcal{O}(n)$-space version of the algorithm, or Diameter4i, is considerable. It uses roughly one-third of the time needed for Diameter4 and, for example, to compute the diameter of a 2-spanner with 4K vertices, it needs 75, 250 and 430 seconds for greedy-graphs, Θ-graphs and WSPD-graphs, respectively. Note that the algorithm Diameter4 needs 900 seconds to compute the diameter of greedy-graphs and this time increases to 1200 seconds for Θ-graphs and WSPD-graphs.

The experiments indicate that the improvements to Algorithm Diameter4 make a considerable difference, see Table 5.1. However, to verify this we would need to perform tests on much larger point sets.

5.6 Concluding remarks

In this chapter we gave several algorithms to compute the spanner diameter of a t-spanner. The fastest algorithm runs in $\mathcal{O}(\mathbb{D} \cdot mn)$ time using $\mathcal{O}(m+n)$ space, where $\mathbb{D}$ is the diameter, m is the size and n is the number of vertices of the input graph. All the algorithms work for general, not necessarily geometric, t-spanners. An obvious open question is if there are faster algorithms for geometric t-spanners. Also in the analysis of the last algorithm we used the obvious $\mathcal{O}(n^2)$ bound for the size of S_k, the set of point pairs which the shortest path between them with at most k links contains exactly k edges. It would be nice if a more intelligent analysis would show a better bound on the number of such pairs in the graph.

Acknowledgements

We would like to thank Shripad Thite for his comments on a talk on this subject which caused the improvement in Algorithm 5.2.2.

Chapter 6

Experimental Study of Geometric Spanners

6.1 Introduction

The problem of constructing spanners has received considerable attention from a theoretical perspective, see [ADD$^+$93, ADM$^+$95, AMS99, BGM04, CDNS95, DHN93, DN97, DNS95, GLN02, Kei88, KG92, LL92, LNS02, Sal91, Vai91], the surveys [Epp00, GK07, Smi00], and the recent book by Narasimhan and Smid [NS07], but almost no attention from a practical or experimental perspective [NP03, SZ04].

In this chapter we consider the most well-known algorithms for the construction of t-spanners in the plane: variants of greedy spanners, variants of Θ-graphs, spanners constructed from the well-separated pair decomposition (WSPD), skip-list spanners, sink spanners and some hybrid algorithms. The quality measurements used in the literature is the number of edges, the weight, the maximum degree, the spanner diameter and the number of crossings. We study each of the algorithms independently, but also in combination with each other. To the best of our knowledge, this is the first time an extensive experimental study has been performed on the construction of t-spanners. Navarro and Paredes [NP03] presented four heuristics for point sets in high-dimensional metric space ($d = 20$) and showed by empirical methods that the running time was $\mathcal{O}(n^{2.24})$ and the number of edges in the produced graphs was $\mathcal{O}(n^{1.13})$. In [SZ04] Sigurd and Zachariasen considered the problem of constructing a minimum weight t-spanner of a given graph, but they only considered sparse graphs of small size, i.e., graphs with at most 64 vertices and with average vertex degree 4 or 8.

The chapter is organized as follows. We first briefly go through the desirable properties for t-spanners. In Section 6.2 we describe the implemented algorithms together with the theoretical bounds and implementation details. Finally, we discuss possible improvements and future research.

Throughout the chapter t will be assumed to be a small constant. In the experiments we used values of t between 1.05 and 2.

6.1.1 Spanner properties

As input we are given a set P of n points in the plane and a real value $t > 1$. The aim is to compute a t-spanner for P with some good properties. Recall from the introduction that the properties that we consider are:

Size: The number of edges in the graph. This is the most important measurement and all the implemented algorithms produce spanners with $\mathcal{O}(n)$ edges.

Degree: The maximum number of edges incident to a vertex.

Weight: The weight of a Euclidean network $\mathcal{G}$ is the sum of the edge weights. The best that can be achieved is a constant times the weight of the minimum spanning tree, denoted $wt(MST(P))$.

Spanner Diameter: Defined as the smallest integer $\mathbb{D}$ such that for any pair of vertices u and v in P, there is a path of length at most $t \cdot \|uv\|$ between u and v containing at most $\mathbb{D}$ edges. We implemented the algorithm in Chapter 5 to compute the spanner diameter of the t-spanners. Throughout the chapter we will simply refer to the spanner diameter, by the diameter.

Recall that some of the above properties are competing, e.g., a graph with constant degree cannot have constant spanner diameter, and a graph with small spanner diameter cannot have a linear number of edges [ADM$^+$95].

6.2 Spanner construction algorithms

Here we give a short description of each of the implemented algorithms together with their theoretical bounds. For a detailed description of each of the algorithms considered in this section please refer to the recent book by Narasimhan and Smid [NS07]. A summary of the theoretical bounds can be found in Table 6.1.

6.2.1 The original greedy algorithm and an improvement

The greedy algorithm was discovered independently by Bern in 1989 and Althöfer *et al.* [ADD$^+$93]. Since then the greedy algorithm has been subject to considerable research [Cha94, CDNS95, DHN93, DN97, GLN02, Soa94]. The original algorithm starts with the complete graph $\mathcal{G}$ while maintaining a partial spanner graph $\mathcal{G}'$ of $\mathcal{G}$. All the edges of $\mathcal{G}$ are sorted with respect to their length in increasing order. Next the edges are processed in sorted order. Processing an edge (u, v) entails a shortest path query in $\mathcal{G}'$ between u and v. If there is no t-path between u and v (a path of length at most $t \cdot \|uv\|$) in $\mathcal{G}'$ then (u, v) is added to $\mathcal{G}'$ otherwise it is discarded. The time complexity of the original greedy algorithm is $\mathcal{O}(n^3 \log n)$ and it uses $\mathcal{O}(n^2)$ space. The graph constructed using the greedy algorithm will be called a greedy graph.

```
Algorithm 6.2.1: Org. Greedy
  Input: $P$ and $t > 1$.
  Output: $t$-spanner $\mathcal{G}' = (P, \mathcal{E}')$.

1 Construct the complete graph of $P$, denoted $\mathcal{G} = (P, \mathcal{E})$;
2 Sort the edges in $\mathcal{E}$ by increasing weight; /* ties are broken arbitrarily */
3 $\mathcal{E}' := \emptyset$;
4 $\mathcal{G}' := (P, \mathcal{E}')$;
5 foreach $(u, v) \in \mathcal{E}$ do                         /* in sorted order */
6     if ShortestPath($\mathcal{G}', u, v$) $> t \cdot \|uv\|$ then $\mathcal{E}' := \mathcal{E}' \cup \{(u, v)\}$;
7 end
8 return $\mathcal{G}' = (P, \mathcal{E}')$;
```

Theorem 6.2.1 *The greedy graph is a t-spanner of P with $\mathcal{O}(\frac{n}{t-1})$ edges, $\mathcal{O}(\frac{1}{t-1})$ maximum degree and total weight $\mathcal{O}(\frac{1}{(t-1)^4} \cdot wt(MST(P)))$, and can be computed in time $\mathcal{O}(n^3 \log n)$.*

Note that a trivial $\Omega(n)$ lower bound on the diameter of a greedy graph is obtained by placing n points on a line.

The greedy approach can also be used to prune a given t-spanner $\mathcal{G} = (P, \mathcal{E})$, that is, instead of considering the edges in the complete graph (see Algorithm 6.2.1), the algorithm only considers the edges in $\mathcal{E}$. In this chapter we also perform experiments using the pruning tool in combination with the other algorithms, see Section 6.3.8.

Improvement

As mentioned above the running time of the implemented algorithm is $\mathcal{O}(n^3 \log n)$, which is very slow when performing experiments with up to 13,000 points. We use a speed-up strategy that turned out to decrease the running time considerably in practice. The original algorithm performs a shortest path query for each pair of points to check if there is a t-path between the two points or not. But there are two simple observations:

1. We only need to know if there is a t-path between the two points, we do not need to *find* a shortest path (or even a path).
2. The algorithm only adds $\mathcal{O}(n)$ edges to the graph in total, so only a small number of changes are made to the graph during the execution.

Therefore, we use a matrix to save the length of the shortest path between each pair of points and update it only when we need to, thus it is not always up to date. Instead of computing a shortest path for each pair (line 6 of Algorithm 6.2.1), we first check the matrix to see if there is a t-path or not and if the answer is no, then we do a shortest path query and update the matrix which enables us to answer the distance query correctly, see Algorithm 6.2.2 for more details.

Algorithm 6.2.2: Imp. Greedy

```
Input: $P$ and $t > 1$.
Output: $t$-spanner $\mathcal{G}' = (P, \mathcal{E}')$.

foreach $(u, v) \in P^2$ do $Weight(u, v) := \infty$;
Construct the complete graph of $P$, denoted $\mathcal{G} = (P, \mathcal{E})$;
Sort the edges in $\mathcal{E}$ by increasing weight; /*ties are broken arbitrarily */
$\mathcal{E}' := \emptyset$;
$\mathcal{G} := (P, \mathcal{E}')$;
foreach $(u, v) \in \mathcal{E}$ do
    if $Weight(u, v) \leq t \cdot \|uv\|$ then
        Discard $(u, v)$;
    else
        Compute single-source shortest path with source $u$;
        foreach $w \in P$ do update $Weight(u, w)$ and $Weight(w, u)$;
        if $Weight(u, v) \leq t \cdot \|uv\|$ then Discard $(u, v)$;
        else $\mathcal{E}' := \mathcal{E}' \cup \{(u, v)\}$;
    end
end
return $\mathcal{G}'(P, \mathcal{E}')$;
```

With these changes, the space complexity of the original greedy algorithm increases by a constant factor, but the gain in running time is considerable. We counted the number of shortest path queries that this algorithm performs in the experiments and surprisingly it seems that only $\mathcal{O}(n)$ shortest path queries is performed. More details can be found in Section 6.3.9.

Conjecture 6.2.2 Algorithm 6.2.2 performs $\mathcal{O}(n)$ shortest path queries.

Implementation

The implementations of the two algorithms are straight-forward. The shortest path queries are done by using the `Dijkstra` function in LEDA.

6.2.2 The approximate greedy algorithm

Even though the improved algorithm is faster than the original greedy algorithm the running time is still $\Omega(n^2 \log n)$, thus any approach that can speed-up the algorithm would be of great interest. The running time is mainly due to the fact that $\Theta(n^2)$ shortest path queries needed to be answered in a graph with $\mathcal{O}(n)$ edges, each of which could take $\mathcal{O}(n \log n)$ time. Das and Narasimhan [DN97] showed how to use clustering to speed up shortest path queries. The approximate greedy algorithm starts with a $\sqrt{t/t'}$-spanner $\mathcal{G}'$ with $\mathcal{O}(n)$ edges and constant degree generated by an $\mathcal{O}(n \log n)$-time algorithm. Note that this network may not have small weight. Then it computes a $\sqrt{tt'}$-spanner of $\mathcal{G}'$ using an approximate variant of the greedy algorithm. To obtain $\mathcal{G} = (P, \mathcal{E})$ from $\mathcal{G}'$ the approximate greedy algorithm starts with $\mathcal{E} = \emptyset$ and adds all the short edges (i.e. those of length at most D/n, where D is the distance between the farthest pair of points) to $\mathcal{E}$. For the remaining edges, the algorithm sorts them by increasing weight and processes them in $\log n$ phases. Processing an edge $e = (u, v)$ entails a shortest path query which is answered by performing an approximate shortest path query on a "cluster graph" H, which is simultaneously maintained. The cluster graph H has the following properties:

1. distances in H "closely" approximate distances in the current graph $\mathcal{G}$.
2. every vertex in H has bounded degree, and
3. "specialized" shortest path queries in H can be answered in constant time.

Algorithm 6.2.3 gives a sketch of the approximate greedy algorithm. For more details see [DN97] or [NS07]. The time complexity of this algorithm is $\mathcal{O}(n \log^2 n)$. Note that the graph generated by this algorithm is an approximate version of the graph generated by the original greedy algorithm since the algorithm prunes a graph with a linear number of edges and answers shortest path queries approximately.

Gudmundsson *et al.* [GLN02] later improved the running time to $\mathcal{O}(n \log n)$ (using the algebraic decision tree model extended with indirect addressing) but the modified version is quite involved and therefore we decided to only implement the above version. The following theorem states the theoretical bounds.

Algorithm 6.2.3: Apx. Greedy

```
Input: $P$, $t > 1$, $t' \in (1, t)$, $\mu > 1$.
Output: $t$-spanner $\mathcal{G} = (P, \mathcal{E})$.

Construct a bounded-degree $\sqrt{t/t'}$-spanner $\mathcal{G}'(P, \mathcal{E}')$;
Sort the edges in $\mathcal{E}'$ by increasing weight; /*ties are broken arbitrarily */
$D :=$ maximum length of any edge in $\mathcal{E}'$;
$\mathcal{E}_0 :=$ edges of $\mathcal{E}'$ having length in $(0, D/n]$;
$\mathcal{E} := \mathcal{E}_0$;
$\mathcal{G} := (P, \mathcal{E})$;
for $k := 1$ to $\lceil \log_\mu n \rceil$ do
    Construct cluster graph $H$ of $\mathcal{G}$;
    $\mathcal{E}_k :=$ sorted sequence of edges of $\mathcal{E}'$ having length in $(\mu^{k-1} D/n, \mu^k D/n]$;
    foreach $(u, v) \in \mathcal{E}_k$ do                    /* in sorted order */
        if ShortestPath$(H, u, v, \sqrt{tt'}\|uv\|) = false$ then
            $\mathcal{E} := \mathcal{E} \cup \{(u, v)\}$;
            Update cluster graph $H$;
        end
    end
end
return $\mathcal{G} = (P, \mathcal{E})$;
```

Theorem 6.2.3 *The approximate greedy graph is a t-spanner of P with $\mathcal{O}(\frac{n}{(t-1)^3})$ edges, $\mathcal{O}(\frac{1}{(t-1)^3})$ maximum degree and weight $\mathcal{O}(\frac{1}{(t-1)^4} \cdot wt(MST(P)))$, and can be computed in time $\mathcal{O}(\frac{n}{(t-1)^7} \log n)$.*

Implementation

The initial $\sqrt{t/t'}$-spanner $\mathcal{G}'$ was constructed using the sink-spanner algorithm (Section 6.2.6). This guarantees that the number of edges is $\mathcal{O}(n)$ and that the graph has constant degree. For the ShortestPath procedure we implemented a variant of Dijkstra's algorithm which answers shortest path queries in constant time in the cluster graph. The query time can be achieved since the maximum degree of the cluster graph is constant and there is a constant upper bound B on the number of edges along a shortest path in the cluster graph, thus we may discard any path containing more than B edges in the priority queue. The bound B can be obtained by choosing the size of the clusters in the cluster graph appropriately.

6.2.3 The Θ-graph algorithm

The Θ-graph was discovered independently by Clarkson [Cla87] and Keil [Kei88]. Keil only considered the graph in two dimensions while Clarkson extended his construction to also include three dimensions. Later Ruppert and Seidel [RS91] and Althöfer *et al.* [ADD$^+$93] defined the Θ-graph for higher dimensions.

Initially we set θ such that $t = \frac{1}{\cos\theta - \sin\theta}$. For each point $u \in P$ consider k non-overlapping cones, $C_i, 1 \leq i \leq k$, with angle $\theta = 2\pi/k$ and with apex u. For each cone C_i we add an edge between u and the point within C_i whose orthogonal projection onto the bisector of C_i is closest to u. Note that instead of the bisector of C_i, we can use any line in the cone passing through the apex of the cone. In our implementation we use one of the boundary lines of the cone instead of the bisector.

Algorithm 6.2.4: Θ-GRAPH

```
Input: $P$ and $t > 1$.
Output: $t$-spanner $\mathcal{G} = (P, \mathcal{E})$.

1  Set $\theta$ such that $t = 1/(\cos\theta - \sin\theta)$;
2  $k := 2\pi/\theta$;
3  $\mathcal{E} := \emptyset$;
4  foreach $u \in P$ do
5      Consider $k$ non-overlapping cones $C_1, \ldots, C_k$ with angle $\theta$ and with
       apex at $u$;
6      foreach cone $C_i$ do
7          $v :=$ the point within $C_i$ whose orthogonal projection onto the
           bisector of $C_i$ is closest to $u$;
8          $\mathcal{E} := \mathcal{E} \cup \{(u, v)\}$;
9      end
10 end
11 return $\mathcal{G} = (P, \mathcal{E})$;
```

Theorem 6.2.4 *The Θ-graph is a t-spanner of P for $t = \frac{1}{\cos\theta - \sin\theta}$ with $\mathcal{O}(n/\theta)$ edges and can be computed in $\mathcal{O}(\frac{n \log n}{\theta})$ time.*

Note that even though the "out-degree" of each vertex is bounded by k the "in-degree" could be linear. Finally, by placing n points on a line it follows that the diameter of the Θ-graph is $\Omega(n)$. Also by placing $n-1$ points on a circle and one point on the center of the circle, we have a Θ-graph with weight $\Omega(n \cdot wt(MST(P)))$.

Implementation

To implement the Θ-graph algorithm, we need a dynamic data structure that, given a cone C, can find the point within C whose orthogonal projection onto the bisector of C is closest to the apex of C in $\mathcal{O}(\log n)$ time, see [NS07] for more details. This data structure is implemented using red-black trees. Since there is no dependency between the cones, one can work on one cone direction at a time, which means that only $\mathcal{O}(n)$ work space is needed.

The same edge may be computed twice during the execution of the algorithm so one needs to check that an edge is not added twice to the graph. Using the graph data structure of LEDA, we need time proportional to the "out"-degree of the points in the graph to check existence of an edge.

A problem that we do not consider in the Θ-graph implementation is rounding errors, which may cause some edges not to be added. For example, if a point lies on the boundary of an, otherwise empty, cone then a small rounding error may "move" the point outside the cone. One way to get rid of this error is to use exact arithmetic. A different possibility is to allow the cones to slightly overlap.

6.2.4 The ordered Θ-graph algorithm

A simple variant of the Θ-graph that has been shown to have good theoretical performance is the *ordered Θ-graph* by Bose *et al.* [BGM04]. An ordered Θ-graph of P is obtained by inserting the points of P in some order. When a point p is inserted, we draw the cones around p and connect p to the previously inserted point with closest orthogonal projection in each cone, like the Θ-graph algorithm.

The order is decided as follows. Initially choose an arbitrary vertex $v_n \in P$ and set its order to n, i.e. this is the last point that will be added to the graph. Process v_n by placing $k = 2\pi/\theta$ cones with apex at v_n and then adding the edges as in the Θ-graph algorithm. In a generic step, assume we processed $i-1$ vertices. In the ith step, choose a point with maximum degree from $P \setminus \{v_n, \ldots, v_{n-(i-1)}\}$ and set its order to $n-i$ and then process v_{n-i} assuming that we have the point set $P \setminus \{v_n, \ldots, v_{n-(i-1)}\}$. This decides an order on the point set. See Algorithm 6.2.5 for more details.

Theorem 6.2.5 *The ordered Θ-graph is a t-spanner of P for $t = \frac{1}{\cos\theta - \sin\theta}$ with $\mathcal{O}(n/\theta)$ edges and $\mathcal{O}(\frac{\log n}{\theta})$ degree, and can be computed in $\mathcal{O}(\frac{n \log n}{\theta})$ time.*

Implementation

For the implementation we use a data structure which is somewhat more complicated than the data structure used for the Θ-graph, since we require the structure

Algorithm 6.2.5: ORDERED-Θ-GRAPH

```
Input: $P$ and $t > 1$.
Output: $t$-spanner $\mathcal{G} = (P, \mathcal{E})$.

1  Set $\theta$ such that $t = 1/(\cos\theta - \sin\theta)$;
2  $k := 2\pi/\theta$;
3  $\mathcal{E} := \emptyset$;
4  $P' := \emptyset$;
5  for $i := 1, 2, \ldots, |P|$ do
6      Pick an unmarked vertex $u \in P \setminus P'$ with maximum degree in
       $\mathcal{G}' := (P, \mathcal{E})$;
7      $P' := P' \cup \{u\}$;
8      foreach cone $C_i$ with apex at $u$ do (See Algorithm 6.2.4)
9          $v :=$ the point in $P \setminus P'$ within $C_i$ whose orthogonal projection onto
           the bisector of $C_i$ is closest to $u$;
10         $\mathcal{E} := \mathcal{E} \cup \{(u, v)\}$;
11     end
12 end
13 return $\mathcal{G} = (P, \mathcal{E})$;
```

to allow for deletions. We use k range trees, as suggested by Bose *et al.* [BGM04], one for each cone with apex at the origin. In each range tree we store all points represented in the coordinate system of the two boundaries of the cone. To find the suitable point in a cone with apex at u, it is sufficient to perform a range query with coordinates of u as keys and choose the suitable point between the points reported by the query. We add one extra pointer to each node of the range tree which shows the point with minimum y (or x) coordinate in the subtree. Using this pointer, we can find the suitable point without going through all reported points of the range query. Each range query requires $\mathcal{O}(\log^2 n)$ time, so the total time complexity of the implemented algorithm is $\mathcal{O}(n \log^2 n)$ which is slightly more than the theoretical time bound but much simpler to implement.

In each step of the ordered Θ-graph algorithm the node with maximum degree has to be selected. To find this point, we used a priority queue of all the points. Initially all the nodes have priority n. When an edge (u, v) is added to the partial spanner graph, the priority of u and v is decreased by 1. The point with minimum priority in the queue is the point with maximum degree in the graph.

There is a major difference between the Θ-graph algorithm and the ordered Θ-graph algorithm when it comes to the space complexity. As mentioned in the previous section, one can construct the Θ-graph by working on one cone direction at a time. This is not possible for the ordered Θ-graph, instead we have to keep all the cones (range trees) in memory. This is due to the fact that the order is not known in advance. During the processing of one node, we need to check all the cones and add edges if necessary, thus $\Theta(kn)$ space is needed. For small values of t this might cause a major problem. To be more precise, the Θ-graph algorithm used roughly 2% of the memory when constructing a 1.05-spanner on a set with 10K points, while the ordered Θ-graph algorithm used almost 85% of the memory.

6.2.5 The random ordered Θ-graph algorithm

The ordered Θ-graph algorithm inserts points into the graph in a specific order. However, if the points are processed in random order then the spanner diameter will be bounded by $\mathcal{O}(\log n)$ with high probability [BGM04]. Unfortunately, the degree bound does not hold in this case. There are two reasons why we implemented the random ordered Θ-graph.

1. Random ordered Θ-graphs, skip-list spanners (Section 6.2.7) and a variant of WSPD-spanner are the only three spanners guaranteed to have bounded spanner diameter. Thus an experimental comparison between the three graphs is interesting.
2. Since the vertices are processed in random order we may fix a random order at the beginning which implies that the algorithm only requires $\mathcal{O}(n)$ space, compared to $\mathcal{O}(kn)$ space needed in order to construct ordered Θ-graphs.

Implementation.

The implementation is the same as for the ordered Θ-graph. We only make a random permutation on the input point set and then process the points in the order they appear in the permutation.

6.2.6 The sink-spanner algorithm

The sink-spanner construction was defined by Arya *et al.* [ADM$^+$95] which construct t-spanners with constant degree. The main idea is as follows. We start with a directed $\sqrt{t}$-spanner with bounded out-degree, denoted $\overrightarrow{\mathcal{G}}$. We will use the Θ-graph which easily can be seen to have out-degree k, but linear in-degree. For each vertex q in $\overrightarrow{\mathcal{G}}$, replace every "star" (the subgraph consisting of all edges in $\overrightarrow{\mathcal{G}}$ pointing to q) in $\overrightarrow{\mathcal{G}}$ by a $\sqrt{t}$-q-sink spanner, see Algorithm 6.2.6 for the details. A $\sqrt{t}$-q-sink spanner is a directed graph where each point has a directed $\sqrt{t}$-path to q. It can be obtained by processing each node q in $\overrightarrow{\mathcal{G}}$ as follows. Consider all points which have an edge pointing to q. Let A_q be the set of all such nodes. We replace all the edges pointing to q by a $\sqrt{t}$-path using the partial sink spanner procedure, see Algorithm 6.2.7.

In the partial sink spanner procedure we look at k cones with apex at q and we partition the points in A_q based on the cones. Let S_i be the points in the ith cone. For each cone i, add an edge between q and the closest point in S_i, say q_i, and then recurse on the partial sink spanner procedure on q_i and $S_i \setminus \{q_i\}$. In the case that one cone contains more than half of the points, split the points in the cone to two almost equal parts and do the same thing as above. This guarantees that the subproblems half in size, thus we get:

Theorem 6.2.6 *The sink-spanner is a t-spanner for P with $\mathcal{O}(kn)$ edges and $\mathcal{O}(\frac{1}{(t-1)^2})$ maximum degree, and can be constructed in time $\mathcal{O}(kn \log n)$.*

Algorithm 6.2.6: SINK-SPANNER

```
Input: $P$ and $t > 1$.
Output: $t$-spanner $\mathcal{G} = (P, \mathcal{E})$.

1  Compute a directed $\sqrt{t}$-spanner $\overrightarrow{\mathcal{G}}(P, \overrightarrow{\mathcal{E}})$ with bounded out-degree;
2  $\mathcal{E}' := \emptyset$; $\overrightarrow{\mathcal{G}'} := (P, \overrightarrow{\mathcal{E}'})$;
3  foreach $q \in P$ do
4      $A_q :=$ all nodes $p \in P$ such that $(p, q) \in \overrightarrow{\mathcal{E}}$;
5      Call PARTIAL-SINK-SPANNER($\overrightarrow{\mathcal{G}'}, q, A_q, \sqrt{t}$);
6  end
7  return $\mathcal{G}' = (P, \mathcal{E}')$;
```

Algorithm 6.2.7: PARTIAL-SINK-SPANNER

```
Input: $\overrightarrow{\mathcal{G}}(P, \overrightarrow{\mathcal{E}})$, a node $q$, a set of nodes $A_q$ and $t > 1$.
Output: partial $q$-sink-spanner $\overrightarrow{\mathcal{G}}(P, \overrightarrow{\mathcal{E}})$.

1  Set $\theta$ such that $t = 1/(\cos\theta - \sin\theta)$;
2  $k := 2\pi/\theta$;
3  Consider $k$ non-overlapping cones $C_1, \ldots, C_k$ with angle $\theta$ and apex at $q$;
4  for $i := 1, 2, \ldots, k$ do
5      $S_i := A_q \cap C_i$;
6      $q_i :=$ closest point in $S_i$ to $q$;
7      $\overrightarrow{\mathcal{E}} := \overrightarrow{\mathcal{E}} \cup \{(q_i, q)\}$;
8      if $|S_i| \leq |P|/2$ then
9          PARTIAL-SINK-SPANNER($\overrightarrow{\mathcal{G}}, q_i, S_i \setminus \{q_i\}, t$)
10     else
11         Split $S_i \setminus \{q_i\}$ to two sets $S_1$ and $S_2$ with almost equal size;
12         PARTIAL-SINK-SPANNER($\overrightarrow{\mathcal{G}}, q_i, S_1, t$);
13         PARTIAL-SINK-SPANNER($\overrightarrow{\mathcal{G}}, q_i, S_2, t$);
14     end
15 end
16 return $\overrightarrow{\mathcal{G}} = (P, \overrightarrow{\mathcal{E}})$;
```

Implementation

To construct the first directed $\sqrt{t}$-spanner, we used the Θ-graph algorithm, with the modification that we add directed edges instead of undirected edges.

6.2.7 The skip-list spanner algorithm

To obtain a spanner with bounded spanner diameter, one can use skip-list spanners as suggested by Arya *et al.* [AMS99]. The idea is to generalize skip-lists and apply them to the construction of t-spanners.

To construct a t-spanner of P, we construct a sequence of subsets of P,

$$P = P_0 \supseteq P_1 \supseteq \cdots \supseteq P_k = \emptyset.$$

To construct P_{i+1}, we flip a fair coin for each element of P_i and then add the point to P_{i+1} if the flip produces heads. The construction ends when the set is empty. Now we construct a t-spanner using the Θ-graph algorithm for each P_i and the union of all these graphs is the skip-list spanner of P, see Algorithm 6.2.8.

```
Algorithm 6.2.8: SKIP-LIST-SPANNER
  Input: $P$ and $t > 1$.
  Output: $t$-spanner $\mathcal{G} = (P, \mathcal{E})$.

$P_0 := P$;
$i := 0$;
while $P_i \neq \emptyset$ do
    $P_{i+1} := \{x \in P_i |$ coin flip for $x$ produces heads$\}$;
    $i := i + 1$;
end
$i := 0$;
while $P_i \neq \emptyset$ do
    $\mathcal{G}_i(P_i, \mathcal{E}_i) :=$ a $t$-spanner on $P_i$ using the $\Theta$-graph algorithm;
end
$\mathcal{E} := \bigcup_i \mathcal{E}_i$;
return $\mathcal{G} = (P, \mathcal{E})$;
```

Theorem 6.2.7 *The skip-list spanner is a t-spanner for P with $\mathcal{O}(kn)$ edges, $\mathcal{O}(\log n)$ spanner diameter and can be constructed in time $\mathcal{O}(kn \log n)$. All the bounds are expected with high probability.*

Implementation

To construct a skip-list spanner, we construct a t-spanner on P using the Θ-graph algorithm. Then for each point in the set we produce a random number between 0 and 10,000 using the `random_source` type in LEDA and remove the point if the outcome is less than 5,000. Then again we construct the Θ-graph on the remaining points and we add the generated edges to the previous graph. We continue this procedure until we have no remaining points in the set.

6.2.8 The WSPD algorithm

Constructing a t-spanner using the WSPD, see Section 2.3, is surprisingly easy. It is sufficient to compute a WSPD of P with respect to $s = 4(t+1)/(t-1)$ and then for every well-separated pair (A, B) in the WSPD an edge is added between an arbitrary point in A and an arbitrary point in B.

Algorithm 6.2.9: WSPD-GRAPH

```
Input: $P$ and $t > 1$.
Output: $t$-spanner $\mathcal{G} = (P, \mathcal{E})$.

1 $\mathcal{W}$ := the well-separated pair decomposition of $P$ using $s := \frac{4(t+1)}{t-1}$;
2 $\mathcal{E} := \emptyset$;
3 foreach $(A_i, B_i) \in \mathcal{W}$ do
4     Select an arbitrary node $u \in A_i$ and an arbitrary node $v \in B_i$;
5     $\mathcal{E} := \mathcal{E} \cup \{(u, v)\}$;
6 end
7 return $\mathcal{G} = (P, \mathcal{E})$;
```

Theorem 6.2.8 *The WSPD-graph is a t-spanner for P with $\mathcal{O}((\frac{t}{t-1})^2 n)$ edges, $\mathcal{O}(\log n \cdot wt(MST(P)))$ weight and can be constructed in time $\mathcal{O}((\frac{t}{t-1})^2 n + n \log n)$.*

An $\Omega(n)$ lower bound on the degree and the spanner diameter of a WSPD-graph can be shown by placing n points on a line with exponentially decreasing inter point distance from left to right.

We also implemented two special versions of the algorithm to improve the degree [ADM+95] and the spanner diameter [AMS99].

Improving the degree

We perform the following modification [ADM+95] to the standard WSPD-algorithm to improve the maximum degree. Instead of adding an arbitrary edge, it selects the point which has the smallest maximum degree from each set and adds an edge between them. This does not improve the theoretical upper bound but we were hoping to see some improvements in the experimental bounds. Note also that the $\Omega(n)$ lower bound does not hold for the modified WSPD-algorithm. In the experiments one can see a small improvement for graphs with $t > 1.5$, for smaller values the difference is negligible. For $t = 1.5$ the improvement is roughly a factor 1.5 and increases to about 3 for $t = 4$.

Another way to improve the maximum degree is to select the point in each well-separated pair using a weighted randomization. That is, we assign a weight to each point which is the total number of points minus the number of well-separated pairs that contain the point. In other words, points that appear in a smaller number of well-separated pairs have greater weight and vice versa. Now we simply select a point randomly from each set based on the weight of the points and then we add an edge between them. Unfortunately, we could not see any improvements in the experiments using this approach compared to the standard approach.

Improving the spanner diameter

Arya *et al.* [AMS99] found a way to bound the spanner diameter of a WSPD-graph. The WSPD is constructed from the so-called fair-split tree, and Arya *et al.* [AMS99] chose a representative point by a search in the fair-split tree. For a node u in the split tree, follow the path down the tree by always choosing the larger subtree. The point stored at the leaf in which the path ends is the representative point for u. This approach guarantees that the spanner diameter of the constructed t-spanner is bounded by $2 \cdot \log n$.

Implementation

We used a split tree for the construction of the WSPD. The points stored at a node is partitioned into two sets by partitioning the non-empty bounding box along its longest side into two boxes of equal size. The tree construction only requires a few percent of the total running time in all our tests.

To decide in constant time if two sets are well-separated we save the smallest enclosing circle of the points in each node. As we mentioned in the note after Definition 2.5.1, two sets are well-separated with respect to s if the distance between the smallest enclosing circles is at least s times the maximum radius of the two smallest enclosing circles.

6.3 Experimental results

In this section we discuss the experimental results in more detail by considering the properties of the graphs generated by the algorithms and the running times of the algorithms. The known theoretical bounds for the algorithms can be found in Table 6.1. The experiments were done on point sets ranging from 100 to 13,000 points with five different distributions:

- uniform distribution,
- normal distribution with mean 500 and deviation 100,
- gamma distribution with shape parameter 0.75,
- $n/100$ uniformly distributed unit squares, each with 100 uniformly distributed points (100-clustered), and
- $\sqrt{n}$ uniformly distributed unit squares, each with $\sqrt{n}$ uniformly distributed points ($\sqrt{n}$-clustered).

In most cases the difference between the last two distributions is negligible so we will refer to both of them as clustered distributions, and the other point sets will be called the non-clustered point sets.

We produced t-spanners using values of t between 1.05 and 2.

-	Size	$\frac{\text{Weight}}{wt(MST)}$	Degree	Diameter	Time
Greedy-graph	$\mathcal{O}(\frac{n}{t-1})$	$\mathcal{O}\left(\frac{1}{(t-1)^4}\right)$	$\mathcal{O}(\frac{1}{t-1})$	$\Theta(n)$	$\mathcal{O}(n^3 \log n)$
Apx. greedy-graph	$\mathcal{O}(\frac{n}{(t-1)^3})$	$\mathcal{O}\left(\frac{1}{(t-1)^4}\right)$	$\mathcal{O}(\frac{1}{(t-1)^3})$	$\Theta(n)$	$\mathcal{O}(\frac{n}{(t-1)^7} \log n)^\dagger$
Θ-graph	$\mathcal{O}(\frac{n}{t-1})$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\mathcal{O}(\frac{n \log n}{t-1})$
O. Θ-graph	$\Theta(\frac{n}{t-1})$	$\mathcal{O}(n)$	$\mathcal{O}\left(\frac{\log n}{t-1}\right)$	$\Theta(n)$	$\mathcal{O}(\frac{n \log n}{t-1})^\dagger$
WSPD-graph	$\Theta(\frac{n}{(t-1)^2})$	$\mathcal{O}(\log n)$	$\Theta(n)$	$\Theta(n)$	$\mathcal{O}((\frac{t}{t-1})^2 n + n \log n)$
Sink-spanner	$\Theta(\frac{n}{t-1})$	$\mathcal{O}(n)$	$\mathcal{O}(\frac{1}{(t-1)^2})$	$\Theta(n)$	$\mathcal{O}(\frac{n \log n}{t-1})$
Skip-list spanner	$\Theta(\frac{n}{t-1})^*$	$\Theta(n)^*$	$\Theta(n)$	$\mathcal{O}(\log n)^*$	$\mathcal{O}(\frac{n \log n}{t-1})^*$

Table 6.1: Summarizing the known bounds for the algorithms presented in the chapter. The entries marked (*) implies that the values are expected with high probability. The entries marked with (†) indicates that the versions implemented in this chapter has an additional $\log n$-factor in their running times.

	Size			Degree			Weight			Diameter			Running time (seconds)		
	Min.	Aver.	Max.	Min.	Aver.	Max.	Min.	Aver.	Max.	Min.	Aver.	Max.	Min.	Aver.	Max.
	$n = 8000$, $t = 1.5$ and uniform distribution														
Org. greedy	15060	15100	15128	7	7	8	3	3	3	103	108	111			
Imp. greedy	15060	15100	15128	7	7	8	3	3	3	103	108	111	213	225	231
Apx. greedy	45879	46469	47543	44	47	54	31	32	33	21	22	23	175	182	192
Θ-graph	118402	118509	118646	51	55	59	58	58	58	26	27	28	0.9	1	1.1
O. Θ-graph	125162	132594	147023	39	44	46	211	259	396	20	22	26	10	12	18
R. O. Θ-graph	164860	165064	165289	169	190	200	135	136	138	7	7	8	6.8	6.9	7
WSPD	1695349	1716652	1734094	1242	1288	1326	5657	5733	5788	3	3	3	11.9	12	12.4
Skip-list	192203	192379	192506	183	192	197	154	156	156	7	5	8	4.7	4.9	4.9
Sink-spanner	115150	115333	115481	44	45	46	58	59	59	24	24.6	25	2.4	2.4	2.4
	$n = 8000$, $t = 1.1$ and clustered distribution														
Org. greedy	29603	29680	29769	15	16	17	8.3	8.7	8.9	49	55	68			
Imp. greedy	29603	29680	29769	15	16	17	8.3	8.7	8.9	49	55	68	531	585	619
Apx. greedy	160497	162482	163208	132	141	151	509	533	573				136	137	137.8
Θ-graph	361672	362557	363762	1750	1933	2110	5156	5376	5608	6	6	6	3.5	5	4.1
O. Θ-graph	346791	358055	362146	119	122	126	8203	9384	10233	6	6	6	90	136	283
R. O. Θ-graph	389186	394404	399675	1773	2512	3012	7129	7585	8012	5	5	5	40	43	46
WSPD	352487	353778	355374	174	179	194	721	766	825	5	5	5	2.8	5	6.3
Skip-list	531298	536602	541053	2347	2409	2496	10692	11043	11311	5	5	5	19	20	21
Sink-spanner	215624	216324	216739	121	123	126	340	345	349	7	7	7	9.8	10	10.4

Table 6.2: The minimum, average and maximum values.

To avoid the effect of specific instances, we ran the algorithms on 5-10 different instances and took the average of the results. However, in almost all cases the difference between the minimum value and the maximum value is negligible. Two example are given in Table 6.2.

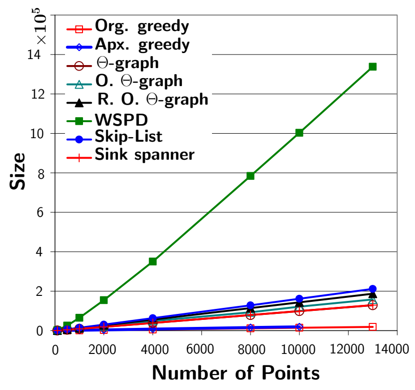
6.3.1 Implementation details

The algorithms were implemented in C++ using the LEDA 5.01 library [MN00]. In the cases when LEDA did not contain the required data structure needed for the algorithms, we implemented it ourselves.

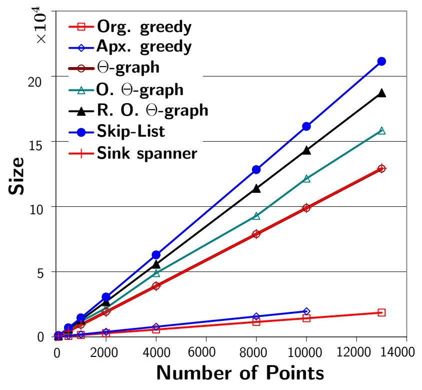
The experiments were performed on an AMD Opteron 250 (2.4 GHz), 1GB L2 cache and 4GB RAM. The OS was Fedora 3.4 and it used g++ 3.4.4 for compiling the program using the `-O2` option. All sample points sets were generated using the NEWRAN03 [Dav05] pseudo random number generator.

6.3.2 Size

Overall the algorithms can roughly be divided into three groups with respect to the size of the produced graphs: (1) the greedy algorithms, (2) the (ordered) Θ-graphs together with the skip-list spanner and the sink-spanner, and (3) the WSPD-graph. The size of the WSPD-graph is roughly a factor 7 to 13 times greater than the size of the (ordered) Θ-graph which in turn is roughly a factor 5 to 10 times greater than the size of the greedy graph. For the uniform distribution and for $t = 2$ the results can be seen in Figure 6.1.



(a) Uniform distribution, $t = 2$.



(b) Same diagram as in (a), but the results generated by the WSPD-algorithms is omitted.

Figure 6.1: Illustrating the number of edges produced by the algorithms for uniformly distributed point sets.

The number of edges in the produced graphs were all linear with respect to the number of points and, as expected, the greedy graph had the smallest number of edges, see Figure 6.1. For $t = 2$, $t = 1.1$ and $t = 1.05$ the number of edges in the greedy graph is approximately $2n$, $4n$ and $6n$ respectively, which is surprisingly small. For comparison it is interesting to note that the Delaunay triangulation has approximately $3n$ edges and dilation bounded by 2.42 [KG92]. The greedy graph has slightly lower number of edges on clustered point sets, see Table 6.3.

The approximate greedy algorithm generates graphs of similar size as the graphs produced by the original greedy algorithm for values of t close to 2. When t decreases the size of the graphs generated by the approximate greedy algorithm deteriorates rapidly and for $t = 1.1$ the size of the approximate greedy graph is almost 24 times the size of the greedy graph, see Table 6.3. The reason for this is that the approximate greedy algorithm approximates the greedy algorithm in two steps (see Section 6.2.3); first the complete graph is approximated using a dense t'-spanner $\mathcal{G}'$ (although of linear size) and then the shortest path queries in $\mathcal{G}'$

	Uniform	Normal	Gamma	Clustered
	$t = 2$			
Original greedy	11K	11K	11K	11K
Approximate greedy	16K	16K	16K	18K
Θ-graph	79K	81K	78K	83K
Skip-list	128K	133K	127K	128K
Sink-spanner	79K	81K	78K	69K
O. Θ-graph	93K	89K	92K	96K
Random O. Θ-graph	114K	117K	112K	109K
WSPD-graph	784K	870K	709K	181K
	$t = 1.1$			
Original greedy	36K	36K	35K	30K
Approximate greedy	852K	894K	809K	162K
Θ-graph	370K	388K	364K	363K
Skip-list	587K	621K	575K	537K
Sink-spanner	413K	432K	403K	216K
O. Θ-graph	390K	402K	399K	358K
Random O. Θ-graph	519K	545K	506K	394K
WSPD-graph	11119K	12822K	9218K	354K

Table 6.3: The size of the spanners generated by the algorithms on point set with 8,000 points for different distributions.

are approximated using a cluster graph H. This works well for large values of t, and in theory for any constant, however, in practice t becomes too small at some point and the error when doing the approximation becomes too large. A notable exception is that it generates graphs of small size on clustered point sets even for small values of t, see Figure 6.2. The main reason being that the approximate

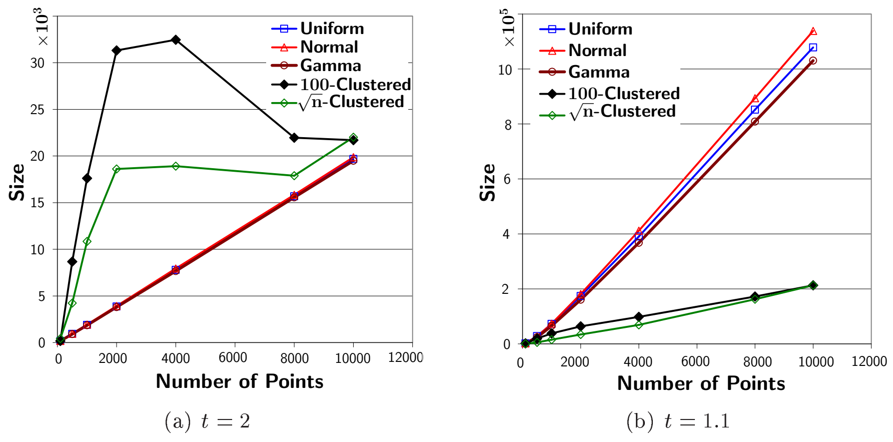


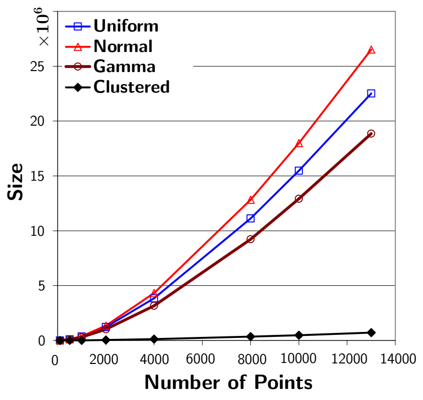
Figure 6.2: Size of the graphs generated by the Apx. greedy algorithm for different distributions.

greedy approach heavily relies on the hierarchical cluster graph. Intuitively, the error in a cluster graph is smaller for point sets that are clustered.

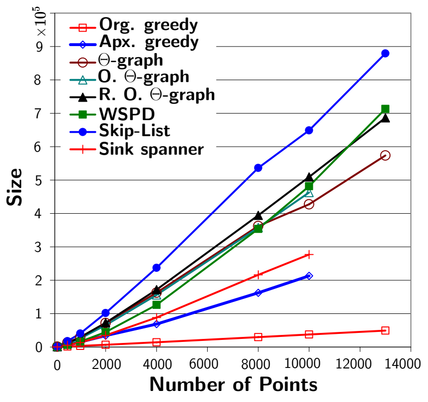
This effect can only be seen for small t values. We believe this depends on line 5 of the Algorithm 6.2.3, i.e., all edges that are shorter than D/n are added to the graph, where D is the length of the longest edge. For clustered sets there are many “short” edges thus the number of short edges will dominate the total number of edges.

The Θ-graph and its variants all produced graphs of similar size (within a factor of 2). Furthermore, the differences in size between the graphs produced for different distributions is very small, usually within 10%.

In general the WSPD algorithm produces very dense t-spanners, see Table 6.3, but it was expected to perform slightly better on clustered sets since it uses a clustering approach. However, the improvement was greater than predicted, see Figure 6.3(a). On clustered sets the WSPD algorithm produced graphs where the number of edges is comparable, or even smaller, to the number of edges in the (ordered) Θ-graph, see Figure 6.3(b).



(a) The WSPD algorithm with different distributions.



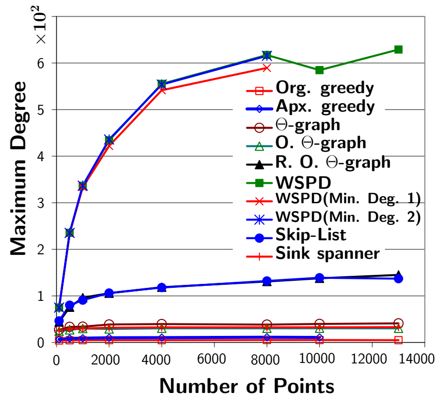
(b) Clustered point sets.

Figure 6.3: Size of the generated graphs for $t = 1.1$.

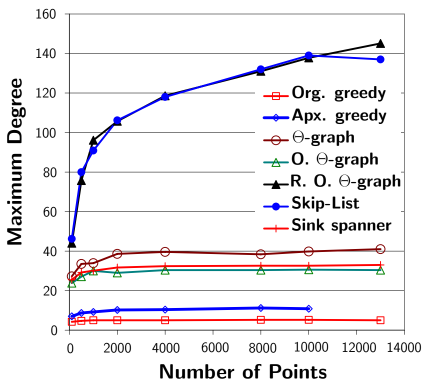
6.3.3 Degree

As in the previous section the algorithms can roughly be divided into three groups depending on the maximum degree of the generated graphs. The first group produced graphs whose maximum degree has a very small dependency on n, or even no dependency. This group contains the (approximate) greedy algorithm, the sink-spanner algorithm and the (ordered) Θ-graph algorithms. Recall from Table 6.1

that the maximum degree of the (approximate) greedy graphs and the sink-spanner is bounded by a function that only depends on t. This was also clearly observed in the experiments, see Figure 6.4.



(a) Uniform distribution, $t = 2$.



(b) Same diagram as in (a), but the WSPD-algorithms results is omitted.

Figure 6.4: Maximum degree for uniform distribution point sets and $t = 2$.

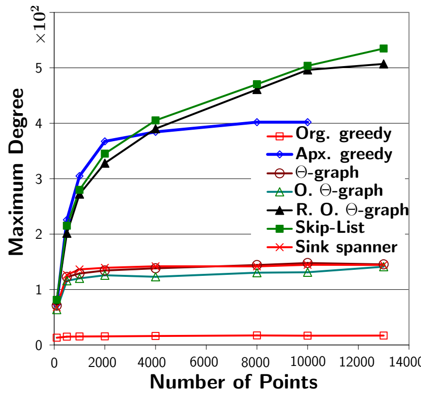
In the tests the greedy algorithm produced graphs with degree roughly 5, 7 and 23 for $t = 2$, $t = 1.5$ and $t = 1.05$ respectively and the bounds are about the same for all the test sets. The degree of the graphs generated by the approximate greedy algorithm also converge to a constant but it is slightly more than the degree of greedy-graphs for large values of t. However for small values of t the degree of the approximate greedy-graph increase rapidly but it still converges to a constant, see Figure 6.4 and Figure 6.5(a). For example, for uniformly distributed point sets with $t = 2$ and $t = 1.1$ the degree of the graph generated by the approximate greedy algorithm is 12 and 400 respectively, see Table 6.4.

Surprisingly, both the ordered Θ-graphs and the Θ-graphs had very low degree on uniformly distributed point sets. For example, for $t = 1.5$ the maximum degree for both seems to converge to approximately 50, which is roughly the same as the maximum degree for the approximate greedy and the sink-spanner. For non-clustered sets the degree of the (ordered) Θ-graphs increases very slowly with respect to the number of points, for example for the uniform distribution the ordered Θ-graph has degree 24 for 100 points and the degree then slowly increases to 31 for 10,000 points. The ordered Θ-graph generally performs slightly better than the Θ-graph.

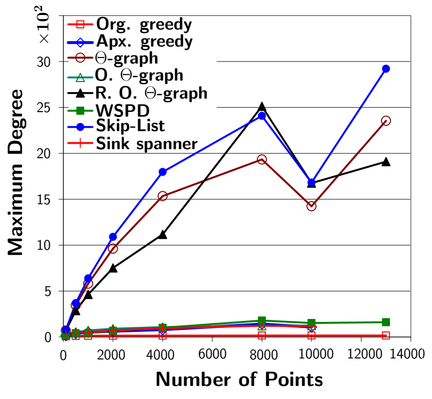
However, for clustered sets the results change unexpectedly. The degree of the Θ-graph deteriorates rapidly and the degree varies highly between different instances, see Table 6.2. The random ordered Θ-graphs and skip-list spanners also have larger degree on clustered point sets, see Figure 6.6. The ordered Θ-graph

	Uniform	Normal	Gamma	Clustered
	$t = 2$			
Original greedy	5	5	5	5
Approximate greedy	11	11	12	12
Θ-graph	38	39	41	357
Skip-list	132	131	128	421
Sink-spanner	33	32	32	39
O. Θ-graph	30	31	31	32
Random O. Θ-graph	131	129	131	430
WSPD-graph	617	589	573	115
WSPD-graph- Mindeg1	590	544	557	115
WSPD-graph- Mindeg2	615	590	575	115
	$t = 1.1$			
Original greedy	17	16	17	16
Approximate greedy	403	396	399	141
Θ-graph	144	145	141	1933
Skip-list	470	471	478	2409
Sink-spanner	142	143	143	124
O. Θ-graph	130	141	137	122
Random O. Θ-graph	461	439	463	2512
WSPD-graph	5192	5993	4938	179
WSPD-graph- Mindeg1	5191	5991	4901	179
WSPD-graph- Mindeg2	5193	5992	4941	179

Table 6.4: The maximum degree of the spanners generated by the algorithms on sets with 8,000 points for different distributions.



(a) Uniform distribution.

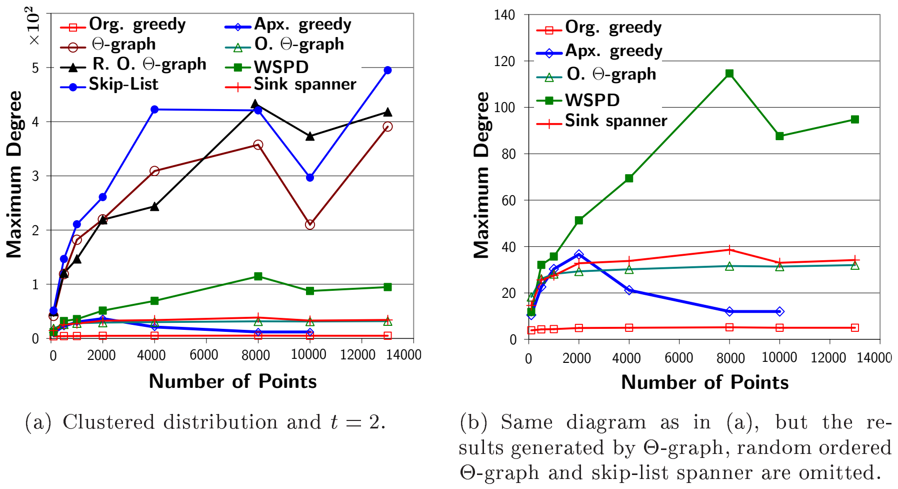


(b) Clustered distribution.

Figure 6.5: Maximum degree of the generated graphs for $t = 1.1$.

on the other hand performs slightly better than the other distributions, as shown in Figure 6.5(b). It is surprising that the (ordered) Θ-graph and other variants of it have almost the same size for uniform and clustered point sets but the maximum degree increases rapidly on clustered point sets (except for the sink-spanner). Again it seems that the experiments supports the theory stated in Theorem 6.2.5 since the ordered Θ-graph has $\mathcal{O}(k \log n)$ degree.

The second group of algorithms contains the skip-list spanner and the random order Θ-graph. No sub-linear upper bound is known on the maximum degree of the graphs produced by these algorithms. One can observe that the maximum degree slowly increases when the size of the graphs increases. As an example, for $t = 1.5$ the maximum degree starts at approximately 50 for $n = 100$ and then steadily increases to roughly 200 for $n = 13K$. Intuitively the expected maximum degree of the skip-list spanner should be $\mathcal{O}(\log n)$ times higher than the maximum degree of the Θ-graph since the expected number of rounds the Θ-graph is produced when constructing the skip-list spanner is $\mathcal{O}(\log n)$.



(a) Clustered distribution and $t = 2$.

(b) Same diagram as in (a), but the results generated by Θ-graph, random ordered Θ-graph and skip-list spanner are omitted.

Figure 6.6: Maximum degree of generated graphs.

The third and final group includes the WSPD-algorithm and its variants. They generated the graphs with the highest degree. For small values of t it almost shows a linear behavior for sets with up to 13K points. Although for larger values of t it seems to converge slowly, but to be able to draw any distinct conclusions more experiments have to be performed with much larger point sets.

As observed in the previous section, the WSPD-algorithm performs much better on clustered sets and seems to converge to a constant for large point sets. For example for $t = 1.05$ and $t = 1.1$ the degree is bounded by 350 and 290 respectively, and it does not seem to increase.

In Section 6.2.8 we proposed two modifications to the WSPD-algorithm that we believed would improve the degree bound in practice. However, the experiments show that the improvements are negligible, see Figure 6.4(a) and Table 6.4.

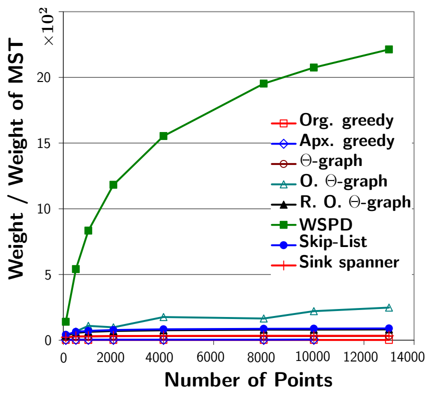
6.3.4 Weight

Recall that theoretically the weight of the greedy graph and the approximate greedy graph is bounded by $\mathcal{O}(wt(MST))$ while the weight of the WSPD-graph is $\mathcal{O}(\log n \cdot wt(MST))$ and the weight of the (ordered) Θ-graph, sink spanner and skip-list spanner is only bounded by $\mathcal{O}(n \cdot wt(MST))$. So the fact that the weight of the greedy graphs in our experiments is much less than the weight of the other graphs is hardly surprising. For $t = 2$ the weight of the greedy graph is approximately 2 times $wt(MST)$ and for $t = 1.1$ and $t = 1.05$ the factors are 10 and 18 respectively, as can be seen in Figure 6.7-6.9. For the clustered sets the bounds are even slightly better, see Table 6.5.

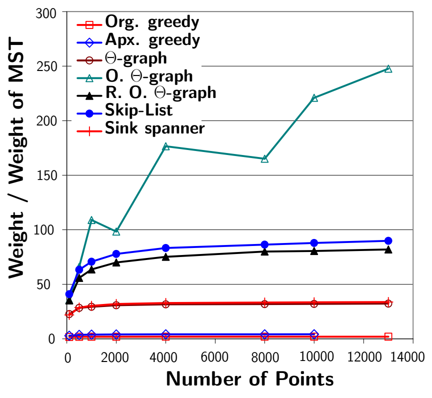
	Uniform	Normal	Gamma	Clustered
	$t = 2$			
Original greedy	2	2	2	2
Approximate greedy	4	4	4	3
Θ-graph	32	33	32	359
O. Θ-graph	165	94	172	1302
Random O. Θ-graph	80	77	74	750
Skip-list	86	86	83	946
Sink-spanner	33	33	32	46
WSPD-graph	1953	2033	1383	233
	$t = 1.1$			
Original greedy	11	11	10	9
Approximate greedy		1441	1333	
Θ-graph	327	327	311	5376
O. Θ-graph	840	634	886	9384
Random O. Θ-graph	727	675	649	7585
Skip-list	815	770	738	11042
Sink-spanner	403	380	357	345
WSPD-graph	70470	64810	41314	766

Table 6.5: The weight of the spanners generated by the algorithms for different distributions.

Even though the WSPD-graph has an $\mathcal{O}(\log n \cdot wt(MST))$ bound the observed weights in the experiments are very large. Its behavior is similar to the degree of the WSPD-graph. For small values of t it shows a linear dependency on n and the weight of the minimum spanning tree, see Figure 6.8(a), and for larger values of t it seems to converge slowly, see Figure 6.7(a). Just as for the degree, the WSPD-algorithm performs very well on clustered sets and its weight almost seems



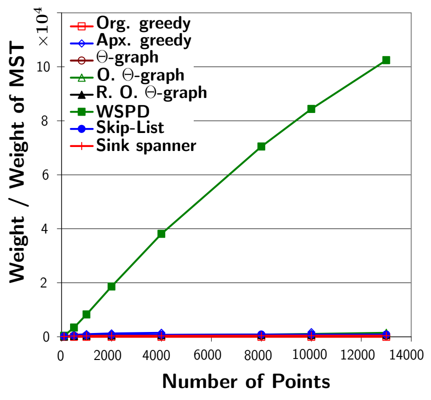
(a) Uniform distribution and $t = 2$.



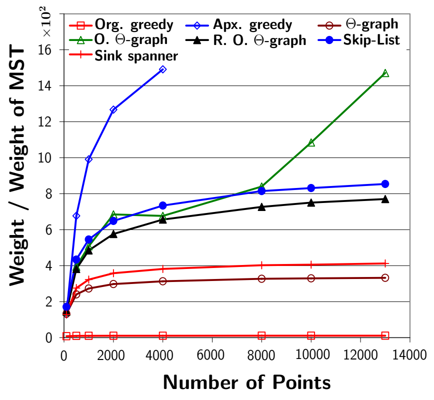
(b) Same diagram as (a), but the results generated by the WSPD-algorithm is omitted.

Figure 6.7: Weight/$wt(MST)$ of the generated graphs.

to converge to a large constant times the weight of the minimum spanning tree for large sets, see Figure 6.9. For example for $t = 1.05$ the ratio was bounded by 900 and for $t = 2$ it was bounded by 230. However, to verify the theoretical bounds experimentally much larger points sets would have to be considered.



(a) Uniform distribution and $t = 1.1$.



(b) Same as in (a) but the results generated by the WSPD-algorithm is omitted.

Figure 6.8: Weight/$wt(MST)$ of the generated graphs.

For the non-clustered sets the weight of the Θ-graph was unexpectedly small and the ratio between its weight and the weight of the minimum spanning tree increased very slowly, see Figure 6.8(b). For example for $t = 1.1$ it went from 133 for 100 points to 330 for 13K points. An interesting question that we have

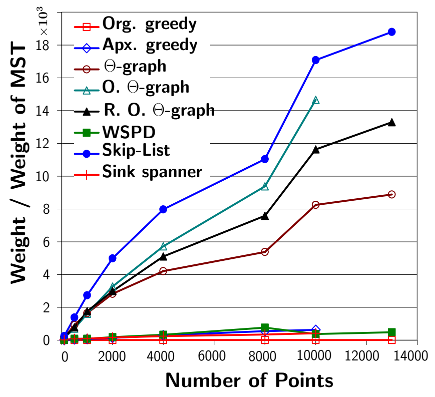


Figure 6.9: Weight/$wt(MST)$ for clustered sets and $t = 1.1$.

not been able to answer, neither through the experiments nor in theory, is if the expected weight of the Θ-graph for uniform point sets is bounded by a constant times the $wt(MST)$? For clustered sets the weight of the Θ-graph is almost linear with respect to the number of clusters and its weight is highly dependent on the different instances, see Figure 6.9 and Table 6.2.

One would expect a similar behavior for the ordered Θ-graph but the weight of the ordered Θ-graph is much higher than both the greedy graph and the Θ-graph. The ratio between the weight of the ordered Θ-graph and the weight of the minimum spanning tree is almost a linear function with respect to the number of points up to 10K points before it starts to level out. Note that the number of edges produced by the Θ-graph and the ordered Θ-graph is very similar for all values on t and n. However, it is easily observed that on average the edges in the ordered Θ-graph are much longer than the edges in the Θ-graph. This follows immediately from the construction since the Θ-graph always adds edges to the closest point in P, while the ordered Θ-graph only can add edges to the points in P that have already been processed. Moreover the behavior of the weight of the ordered Θ-graph is unpredictable and seems to be highly dependent on the specific instances, see Figure 6.7(b) and Table 6.2.

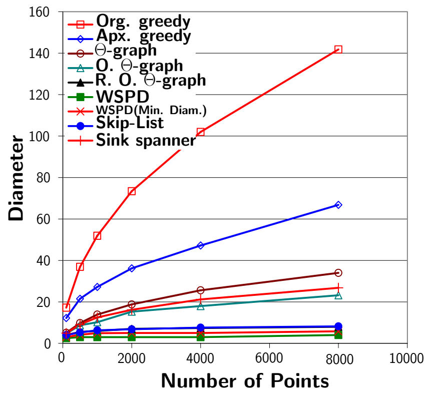
The weight of the graphs generated by the (random/ordered) Θ-graph and the skip-list spanners increase rapidly on clustered point sets, see Table 6.5.

6.3.5 Spanner diameter

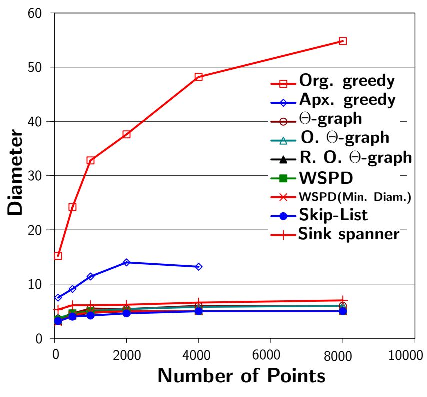
Due to the high time complexity to compute the spanner diameter of a graph we could only compute the diameter for graphs with up to 8,000 points. To compute the spanner diameter of the graphs we implemented the algorithm suggested in Chapter 5.

Note that most of the algorithms that we implemented may produce graphs whose spanner diameter is $\Theta(n)$. The only known algorithms that produce graphs with smaller diameter are the skip-list spanner a variant of WSPD-algorithm with $\mathcal{O}(\log n)$ diameter and the random ordered Θ-graph which was shown to have $\mathcal{O}(\log n)$ diameter with high probability [AMS99, BGM04].

As expected the greedy graphs had the highest diameter, see Figure 6.10. This follows from the fact that the greedy graph has fewer edges than the other graphs and the greedy approach favors short edges and avoids adding long edges. The diameter of a 2-spanner generated by the greedy algorithm (uniform distribution) is about 17 for a set with 100 points and it reaches 142 for a set with 8,000 points. The diameter seems to depend linearly on the size of the input set. Also the diameter changes slightly depending on the different distributions, for $t = 1.1$ and $n = 8,000$ the diameter varies between 54 for the normal distribution to 70 for the gamma distribution.



(a) Uniform distribution and $t = 2$.



(b) Clustered distribution and $t = 1.1$.

Figure 6.10: Diameter of generated graphs.

The diameter of the (ordered) Θ-graph is much smaller than that of the greedy graph. A 2-spanner generated by the (ordered) Θ-graph has diameter approximately 5 for a set with 100 points and it increases to 23 (ordered Θ-graph) and 34 (Θ-graph), for a set with 8,000 points. The ordered Θ-graph has generally a slightly lower diameter as compared to the Θ-graph. The diameter of the (ordered) Θ-graphs is almost the same for all the distributions except for clustered distributions for which the diameter is roughly half of the diameter of (ordered) Θ-graphs on non-clustered point sets, see Table 6.6.

Because of the tradeoff between the maximum degree and the diameter we expected the graph generated by the sink-spanner algorithm to have a larger diameter compared to the Θ-graph. However, in the experiments the diameters are

	Uniform	Normal	Gamma	Clustered
	$t = 2$			
Original greedy	142	109	155	164
Approximate greedy	67	51	72	72
Θ-graph	34	28	36	11
O. Θ-graph	23	21	23	11
Random O. Θ-graph	8	8	8	8
Skip-list	8	8	8	8
Sink-spanner	27	25	26	13
WSPD-graph	4	3	4	7
WSPD-graph (Min. diam.)	6	5	6	8
	$t = 1.1$			
Original greedy	57	54	70	55
Approximate greedy		7	7	
Θ-graph	14	13	16	6
O. Θ-graph	13	12	14	6
Random O. Θ-graph	6	6	6	5
Skip-list	5	5	6	5
Sink-spanner	11	11	13	7
WSPD-graph				5
WSPD-graph (Min. diam.)				5

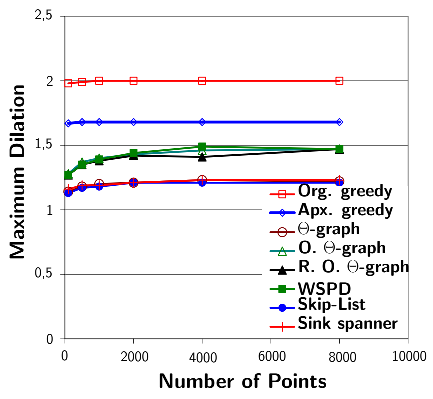
Table 6.6: The diameter of the spanners generated by the algorithms on point set with 8,000 points for different distributions.

almost the same and sometimes even lower for non-clustered distributions.

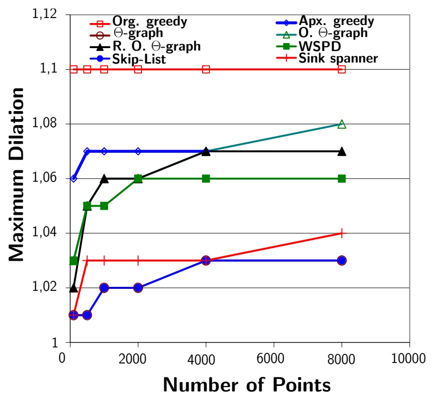
Finally, the WSPD-graph has very low diameter and it converges fast. This follows from the simple fact that the WSPD-graph is very dense, which intuitively implies that it will have a small diameter. The diameter of a 2-spanner on a set with 100 points (uniform distribution) is 3 and it increases to 4 for a point set with 8,000 points. In the clustered sets, the diameter is slightly higher, between 3 and 8, probably because the number of edges in these graphs are smaller compared to the graphs for the non-clustered sets. An oddity is that the modified WSPD algorithm which improves the spanner diameter of the generated graphs has slightly higher diameter compared with the standard WSPD-graphs, see Figure 6.10(a) or Table 6.6.

6.3.6 Maximum and average dilation

One of the spanner properties is the real dilation of the generated graphs. In Figure 6.11 one can observe that there is a large discrepancy between the algorithms. The original greedy algorithm generate graphs with maximum dilation equal to t, or very close to t, for all the tested instances.



(a) Uniform distribution and $t = 2$.



(b) Clustered distribution and $t = 1.1$.

Figure 6.11: Maximum dilation of generated graphs.

Since the approximate greedy algorithm approximates the greedy spanner the maximum dilation is slightly smaller than for the greedy. In general we could see that the greedy spanner had a maximum dilation that was roughly 30% larger than the maximum dilation of the approximate greedy spanner.

The WSPD-graph and the (random/ordered) Θ-graphs produce graphs with considerably smaller dilation, see Table 6.7. For example, for uniformly distributed point sets using $t = 1.5$ the (random/ordered) Θ-graphs and the WSPD-graph had maximum dilation 1.25, while the approximate greedy and the original greedy had maximum dilation 1.36 and 1.5, respectively.

	$t = 2$					$t = 1.1$				
n	500	1000	2000	4000	8000	500	1000	2000	4000	8000
Org. greedy	1.99	2	2	2	2	1.1	1.1	1.1	1.1	1.1
Imp. greedy	1.99	2	2	2	2	1.1	1.1	1.1	1.1	1.1
Apx. greedy	1.68	1.68	1.68	1.68	1.68	1.07	1.07	1.07	1.07	
Θ-graph	1.18	1.2	1.21	1.23	1.22	1.02	1.02	1.03	1.03	1.03
O. Θ-graph	1.37	1.4	1.43	1.46	1.47	1.06	1.07	1.07	1.07	1.07
R. O. Θ-graph	1.35	1.38	1.42	1.41	1.47	1.06	1.06	1.07	1.07	1.07
Skip-list	1.17	1.18	1.21	1.21	1.21	1.02	1.02	1.03	1.03	1.03
Sink-spanner	1.19	1.19	1.21	1.23	1.23	1.03	1.03	1.03	1.03	1.04
WSPD-graph	1.35	1.39	1.44	1.49	1.47	1.04	1.04	1.05	1.05	1.05

Table 6.7: The maximum dilation of graphs generated by different algorithms on uniformly distributed point sets.

Finally, the Θ-graph, skip-list spanner and sink-spanner had the smallest maximum dilation, approximately 1.13 for $t = 1.5$. It is interesting to note that even though the size of the WSPD-graph is much greater than the size of the Θ-graph the maximum dilation of the WSPD-graph is clearly greater than the maximum dilation of the Θ-graph.

The difference between the different approaches is, more or less, the same for all input instances, the results for clustered point sets are shown in Figure 6.11(b).

During the experiments we also measured the average dilation. This is a property that has not been considered previously. However, it might be of interest to know that the average dilation in the graphs is usually extremely small. Even for $t = 2$ the average dilation for all graphs, except the (approximate) greedy graph, is less than 1.01. For the greedy graph the average dilation is 1.19 for $t = 2$ and 1.1 for $t = 1.5$.

6.3.7 Crossings

The last property we discuss is the number of crossings in the generated graphs. Obviously this is highly dependent on the number of edges, therefore it is not surprising that the greedy graph is superior to the other graphs and it is the only graph with a reasonable number of crossings. The experiments could only be done on sets with up to 2,000 points since the number of crossings is bounded by $\mathcal{O}(m^2)$ where m is the number of edges in the graph. For $t = 1.5$ and $n = 2,000$ the number of crossings in the greedy graph is on average 94. Also the number of crossings seems to increase linearly with respect to the number of points which is surprisingly low. For $t = 1.1$ the number of crossings for $n = 100$, 500 and 2,000 is on average 397, 2,750, and 12,411.

We did some initial experiments with the other algorithms but the number of crossings were very high. For example for $t = 1.1$ and $n = 100$ the Θ-graph has 2,437 edges and 300K crossings! Thus if the number of crossings is a priority to the user then the only option is to use the greedy algorithm.

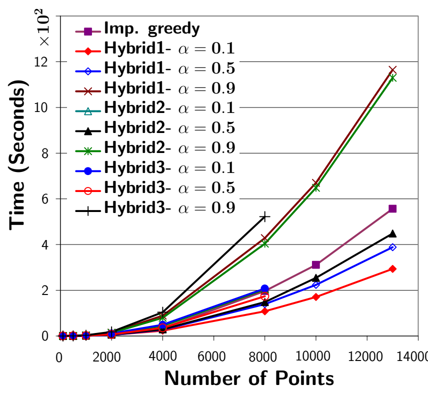
6.3.8 The hybrid algorithms

During the experiments it rapidly became clear that the greedy algorithm produced graphs whose size, weight, maximum degree and number of crossings are superior to the graphs produced from the other approaches. However the running time of the greedy algorithm is $\mathcal{O}(m \cdot n \log n)$, where m is the number of edges in the original graph. Since the input is a set of n points we have to consider the complete graph, thus $m = \Theta(n^2)$. A way to improve the running time while, hopefully, still obtaining high-quality graphs is to first compute a t^α-spanner $\mathcal{G}$ of the input set and then compute a $(t^{1-\alpha})$-spanner of $\mathcal{G}$ using the greedy pruning algorithm with $0 < \alpha < 1$. The dilation of the resulting graph is bounded by $t^\alpha \cdot t^{1-\alpha} = t$. The t^α-spanner can be constructed using (ordered) Θ-graphs or WSPD-graphs, ensuring that the number of edges is $\mathcal{O}(n)$, and consequently the total running time would decrease to $\mathcal{O}(n^2 \log n)$.

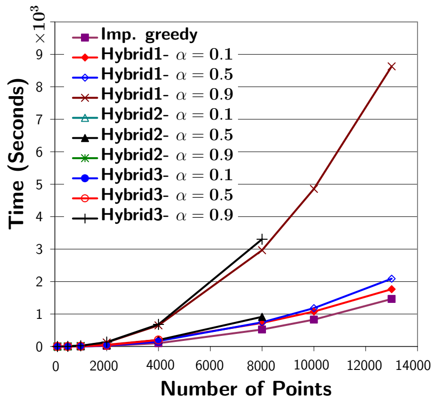
A second reason why we consider hybrid algorithms is the fact that the (ordered) Θ-graphs and the WSPD-graph actually have much smaller dilation than the specified t-value. For example for $t = 2$ the greedy graph has dilation close to 2 while the ordered Θ-graph and the WSPD-graph has dilation 1.4 and the Θ-graph has dilation 1.2. For $t = 1.1$ the Θ-graph has dilation 1.02, the ordered Θ-graph has dilation 1.06 and the WSPD-graph has dilation 1.04, see Figure 6.11. By first producing the (ordered) Θ-graph or the WSPD-graph we use the fact that they can be constructed fast and the number of edges remaining is linear. Since their dilation in practice is very small it leaves a lot of freedom for the greedy algorithm to produce a t-spanner with good properties.

This approach has another advantage which is that the parameter α can be adjusted to fit the application. If α is chosen to be close to zero then the resulting graph is very similar to the greedy graph but the gain in running time is small. If α is chosen close to 1 then the algorithm is faster but the quality of the graph is worse. We test three hybrid algorithms, Θ-graph algorithm plus greedy pruning (Hybrid1), ordered Θ-graph algorithm plus greedy pruning (Hybrid2) and WSPD algorithm plus greedy pruning (Hybrid3). The test are performed using three values of α, 0.1, 0.5 and 0.9. To optimize the complexity of the hybrid algorithms we used the improved greedy algorithm (see Algorithm 6.2.2) for the pruning step.

Note that the difference between the graphs generated by the (ordered) Θ-graph algorithms and the WSPD algorithm is very big but the pruning step reduces the gap between them. Thus the graphs generated by the hybrid algorithms have almost the same properties no matter which algorithms we use as a first step, see Table 6.8. Therefore, for the rest of this section, we only consider the Hybrid1



(a) Uniform distribution and $t = 2$



(b) Uniform distribution and $t = 1.1$

Figure 6.12: The performance of the hybrid algorithms for different values of α.

$n = 8000$	$t = 2$								
	Greedy			Hybrid1		Hybrid2		Hybrid3	
	Uni.	Clus.	α	Uni.	Clus.	Uni.	Clus.	Uni.	Clus.
Size	11K	11K	0.1	12K	11K			12K	11K
			0.5	17K	15K	17K	15K	17K	15K
			0.9	41K	33K	38K	32K	43K	35K
Degree	5.2	5.2	0.1	5.6	5			5.6	5.8
			0.5	8.2	7.8	8.2	7.8	8	8.2
			0.9	19.2	18.4	17.2	16.8	18.8	18.6
Weight	2	1.8	0.1	2.1	1.9			2.1	1.9
			0.5	3.5	3	3.5	3	3.5	3
			0.9	12.7	10.7	12	11	14.3	11.5
Diameter	142	164	0.1	135	143			135	100
			0.5	100	84	101	79	99	56
			0.9	53	26	53	27	46	21
Max. dilation	2	2	0.1	1.9	1.9			1.9	1.9
			0.5	1.5	1.4	1.57	1.55	1.55	1.53
			0.9	1.3	1.28	1.43	1.42	1.39	1.44
Running time in seconds (Imp. greedy)	197	168	0.1	108	109			208	53
			0.5	140	95	148	133	174	61
			0.9	428	223	404	262	522	121

Table 6.8: The properties and running time of the hybrid algorithms. Note how the value of α influences the properties and running time of the hybrid algorithms.

algorithm.

One should note that for $\alpha = 0.1$, the hybrid algorithms generate graphs which are very close to the greedy graph, see Table 6.8. The larger α value we choose the

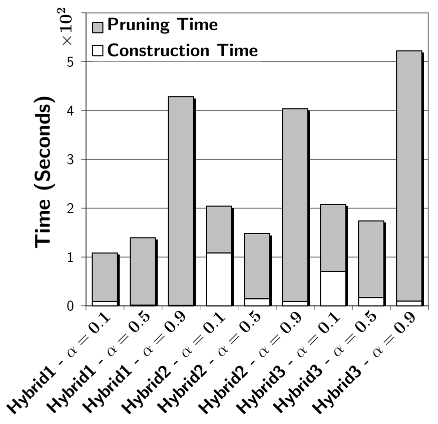


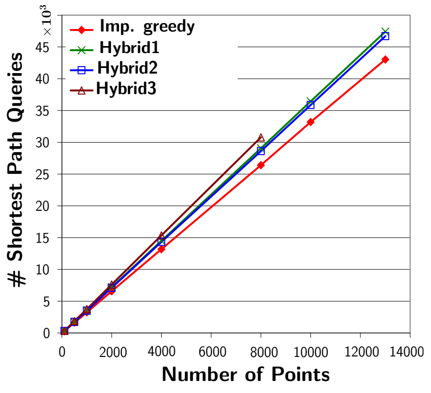
Figure 6.13: Comparing the construction time of the initial graph with the pruning time (Uniform distribution and $t = 2$ and $n = 8000$).

larger is the difference between the greedy graph and the graph generated by the hybrid algorithms. For example for $t = 2$ and $\alpha = 0.9$, the generated graph is even worse than the approximate greedy-graph. However for small t-values, the hybrid algorithm generates a graph with smaller size than the approximate greedy-graph.

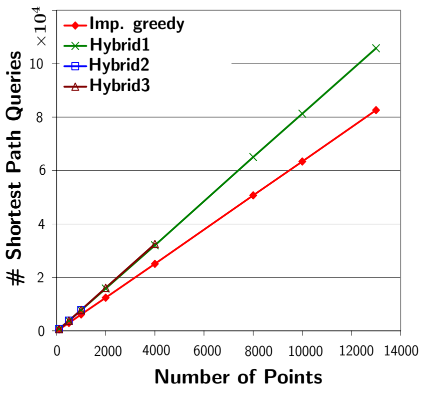
The hybrid algorithms are in general not very sensitive to the distribution of the input points. The running times for the hybrid algorithms are dependent on the value of α. For small α values the dependency is almost linear on the number of points and for large α values the dependency on n increases, see Figure 6.12. This is because the largest fraction of the time is needed for the pruning step which increases when we have small t values. By increasing α, less time is needed to build the initial graph while more time is needed for the pruning process, see Figure 6.13. However, the added time spent on pruning results in a better quality network, i.e. small size, low degree and low weight.

6.3.9 Number of shortest path queries

In Section 6.2.1 we conjectured that the improved greedy algorithm only performs a linear number of shortest path queries. As can be seen in Figure 6.14, the experiments strongly support the conjecture. The number of performed shortest paths queries is even smaller than the number of shortest paths queries performed by the hybrid algorithms. This is important since the hybrid algorithm only performs $\mathcal{O}(n)$ shortest paths queries, one for each edge in the initial graph. By decreasing t, the number of shortest path queries increases but it still shows a linear dependency on n, and the dependency on t is small.



(a) Uniform distribution $t = 2$.



(b) Uniform distribution $t = 1.1$.

Figure 6.14: The number of shortest paths queries performed by the improved greedy algorithm and hybrid algorithms.

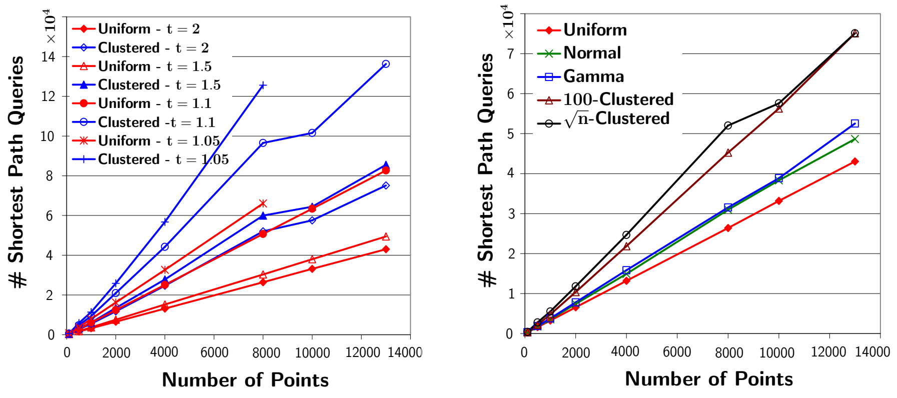
$n = 8000$		Uniform	Normal	Gamma	Clustered
		$t = 2$			
Improved greedy		26K	31K	32K	52K
Hybrid1	$\alpha = 0.1$	26K	26K	26K	42K
	$\alpha = 0.5$	29K	29K	29K	38K
	$\alpha = 0.9$	48K	49K	48K	54K
Hybrid2	$\alpha = 0.1$				
	$\alpha = 0.5$	29K	29K	29K	37K
	$\alpha = 0.9$	45K	45K	44K	52K
Hybrid3	$\alpha = 0.1$	27K		31K	29K
	$\alpha = 0.5$	31K		32K	30K
	$\alpha = 0.9$	52K		52K	45K
		$t = 1.1$			
Improved greedy		51K	54K	55K	97K
Hybrid1	$\alpha = 0.1$	26K	53K	52K	86K
	$\alpha = 0.5$	65K	66K	65K	104K
	$\alpha = 0.9$	127K	130K	125K	171K
Hybrid2	$\alpha = 0.1$				
	$\alpha = 0.5$				
	$\alpha = 0.9$				
Hybrid3	$\alpha = 0.1$				
	$\alpha = 0.5$				
	$\alpha = 0.9$	130K		130K	95K

Table 6.9: The number of the shortest path queries performed by the improved greedy algorithm and hybrid algorithms.

The number of shortest paths queries increases slightly for clustered point sets but it is still linear, see Figure 6.15(b). This is somewhat surprising since the size of the greedy graph is smaller for clustered point sets than for non-clustered point sets.

For example, to construct a 2-spanner on a set of 8K uniformly distributed points, the improved greedy algorithm performed approximately 26K shortest path queries while the original algorithm performs roughly 32 million queries. For clustered point sets with the same size the number of shortest path queries increases to 52K, see Table 6.9.

To see the reason for this, we partition the shortest path queries performed by the improved greedy algorithm into two sets. The first set contains all the queries which results in adding an edge to the graph. The second set contains the remaining queries. That is, these queries occur when the value in the weight matrix (see Algorithm 6.2.2) is not properly updated. Instead a query is performed, but since no edge is added the graph already contains a t-path. This means that if the entries in the weight matrix is far from the real shortest paths then the size of the second shortest path query set increases.



(a) Different t's. (b) $t = 2$.

Figure 6.15: Comparing the number of shortest paths queries performed by the improved greedy algorithm.

The size of the greedy-graph, and therefore the number of shortest paths queries in the first set, is almost the same for clustered and non-clustered point sets. So the difference in the number of queries between clustered and non-clustered point sets comes from the second set of queries.

In clustered point sets, when an edge connects two points in the same cluster, it changes the path lengths between points which lie within the same cluster. But when it comes to an edge which connect two clusters, especially when the clusters are not connected, it affects the shortest path lengths of a much wider range. This means, in the clustered point sets the size of the second shortest path query set is much larger than in the non-clustered case.

Another interesting observation comes from comparing the size of the greedy graphs with the number of shortest path queries performed by the improved greedy algorithm. For non-clustered point sets with 8K points and for $t = 2$ the number of shortest paths queries is at most a factor 3 times the number of edges and in the clustered points sets, this increases to 5, see Table 6.3 and Table 6.9. By adding an edge to the graph, we actually update the graph and it might change the length of many shortest paths but the experiments show that on average the improved greedy algorithm performs roughly 5 shortest path queries per edge which is surprisingly small.

6.3.10 Running time

In this section we study the running times of the implemented algorithms. We first consider their behavior on uniformly distributed sets, and then we point out the differences for different distributions.

Uniform distribution

The running times of all the implemented algorithms (except the original greedy algorithm) for $t = 2$ and $t = 1.1$ are depicted in Figure 6.16.

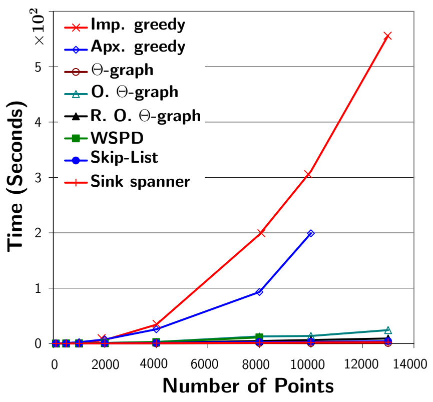
We start with the greedy approaches. As the theoretical bounds suggest, the original greedy algorithm has the highest time complexity of all the implemented algorithms and it clearly shows in the experiments, see Table 6.10. As an example, constructing a greedy 2-spanner on a set of 4K uniformly distributed points, the original greedy algorithm required 12K seconds while the improved algorithm only needed roughly 34 seconds.

Using the improved algorithm we are able to construct greedy graphs for much larger points sets. For instance for a set of 10K points we can construct a 2-spanner greedy graph in about 300 seconds.

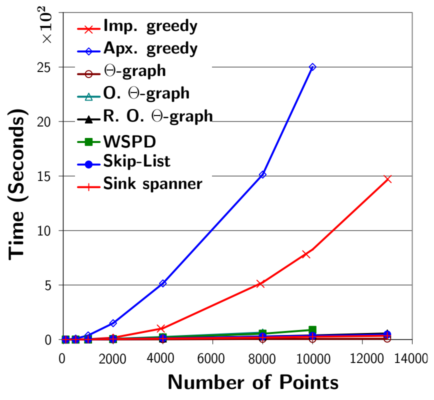
Based on the experiments, the running time of the improved greedy algorithm is comparable to the running times of the hybrid algorithms using $\alpha = 0.5$ for $t = 2$ and the algorithm performs even better for smaller values of t, see Figure 6.12 for

	$n = 4000$				$n = 8000$			
	Uni.	Nor.	Gam.	Clus.	Uni.	Nor.	Gam.	Clus.
	$t = 2$							
Org. greedy	12273	12755	11844	9205				
Imp. greedy	34	39	38	28	197	237	214	168
Apx. greedy	26	26	24	12	93	93	84	52
Θ-graph	0.3	0.3	0.3	0.2	0.9	0.6	0.7	0.6
O. Θ-graph	3.4	3.4	3	7	13	7	7	19
R. O. Θ-graph	1.8	1.8	1.8	3	4.6	4.7	4.6	8
WSPD-graph	2.7	3.7	3	0.6	11	10	12	1.6
Skip-list	1.2	1.2	1.1	1	2.9	2.6	2.8	2.4
Sink-spanner	0.7	0.7	0.7	0.6	1.4	1.4	1.4	1.2
	$t = 1.1$							
Org. greedy	42198	45334	41810	29361				
Imp. greedy	103	115	110	111	524	570	519	585
Apx. greedy	515	592	398	28	1513	1717	1271	128
Θ-graph	2.3	2.4	2.2	1.6	5.4	5.7	5.2	3.9
O. Θ-graph	23	22.5	22	42	63	73	64	136
R. O. Θ-graph	10.6	11	10.8	16	27.6	28.5	27.3	42
WSPD-graph	20	23	21	1	53	66	58	5
Skip-list	12	13	11	8	28	30	26	20
Sink-spanner	9.5	9.4	8	3.7	19.6	21	19.7	10

Table 6.10: The running times are shown in seconds for the algorithms with different distributions.



(a) Uniform distribution, $t = 2$.



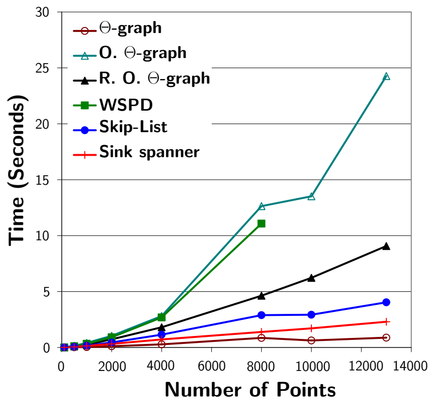
(b) Uniform distribution, $t = 1.1$.

Figure 6.16: Comparing the running times of the implemented algorithms for uniformly distributed point sets. Note the difference between the approximate greedy algorithm and the improved greedy algorithm for the two values of t.

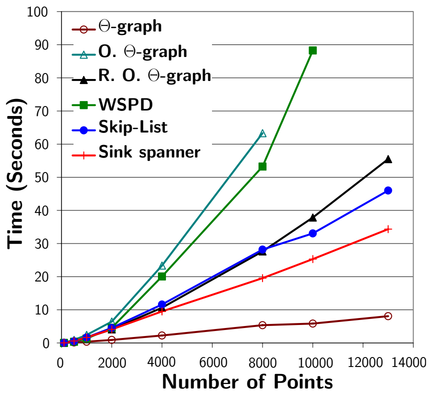
a comparison. Thus, if high quality networks are a priority the improved greedy algorithm is probably the best choice, especially for small values of t. Note that the improved greedy algorithm generates the same graph as the original greedy algorithm.

The remaining algorithms all have time complexity $\mathcal{O}(n \log^2 n)$, or even $\mathcal{O}(n \log n)$. However, the difference in their actual running times is quite substantial and for some a bit surprising. The Θ-graph algorithm is superior to the others with respect to the running time. For sets containing 10K points and for t between 1.5 and 2 the Θ-graph was constructed in less than two seconds. For $t = 1.1$ the running time increased to approximately 6 seconds, which is to be expected since its running time is highly dependent on the value of $1/(t-1)^2$. The fastest algorithms after the Θ-graph construction were the sink-spanner algorithm, the skip-list spanner algorithm and the random ordered Θ-graph algorithm which basically are all modified Θ-graph algorithms. Again for 10K points they required a couple of seconds for $t = 2$ and approximately half a minute for $t = 1.1$. These algorithms almost show a linear time behavior in our experiments, see Figure 6.17.

For uniformly distributed point sets the running times of the ordered Θ-graph algorithm and the WSPD algorithm clearly show a superlinear behavior but they are still fast enough to handle 8K points with $t = 1.1$ in roughly one minute. For smaller values of t and larger point sets the ordered Θ-graph algorithm ran into memory problems. The simplified version that we implemented uses $\Omega(\frac{2\pi}{\theta} n \log n)$ space (instead of $\Omega(\frac{2\pi}{\theta} n)$ space) and for small values of t and large values of n this function grows rapidly.



(a) Uniform distribution, $t = 2$.



(b) Uniform distribution, $t = 1.1$.

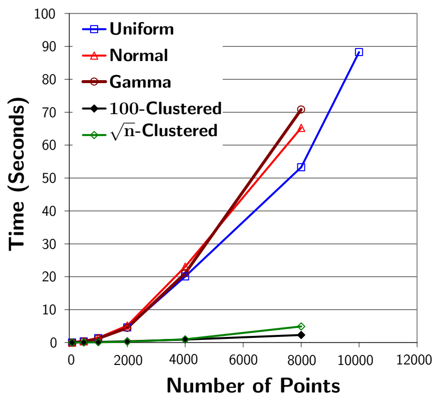
Figure 6.17: The running time for the $\mathcal{O}(n \log n)$-time algorithms in the experiments for Uniform distributions.

The approximate greedy algorithm works well for large values of t, however, when t decreases the running time deteriorates rapidly. It is comparable to the running time of the improved greedy algorithm for $t = 2$ and even slower for $t = 1.1$, see Figure 6.16. Thus, even if the theoretical bound on the running time of the implemented algorithm is $\mathcal{O}(n \log^2 n)$ it seems that the constant hidden in the $\mathcal{O}$-notation is so large that for points sets up to 13K points the actual running time is much worse than what we would expect. Recall that the conjectured running time of the improved greedy algorithm is $\mathcal{O}(n^2 \log n)$.

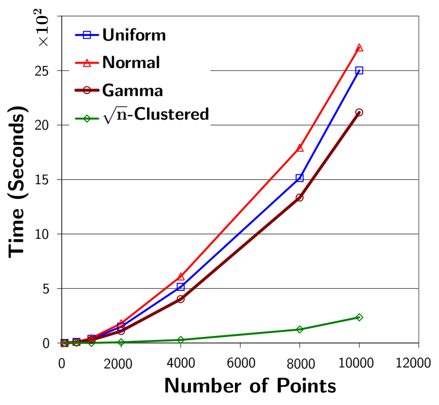
Other distributions

Most of the algorithms perform slightly better on the clustered point sets, except the WSPD-algorithm and the approximate greedy algorithm which both show a considerable improvement. For example, to construct a 2-spanner on a uniformly distributed set which contains 8K points, the WSPD algorithm needs roughly 11 seconds while the corresponding running time for the clustered set is about 1.6 seconds. For $t = 1.1$ the improvement is even bigger; 53 seconds compared to 2.5 seconds, see Figure 6.18(a). The WSPD algorithm was expected to perform slightly better for clustered sets since it uses a clustering approach, but the improvement was greater than predicted. Especially for small values of t the algorithm performs better, it is even comparable to the Θ-graph algorithm for the clustered set with 10K points and $t = 1.1$.

As you can see in Figure 6.18(b), a similar observation can be made for the approximate greedy algorithm where the corresponding running times for $t = 1.1$ and $n = 8K$ are 1500 seconds and 128 seconds. As for the WSPD-approach the



(a)



(b)

Figure 6.18: The running time of (a) the WSPD algorithm and (b) the approximate greedy algorithm for $t = 1.1$ and with different distributions.

approximate greedy algorithm also uses a clustering approach however the main gain comes from the fact that the algorithm does not process any edges in the initial graph $\mathcal{G}'$ of length at most D/n (they are just added to the partial spanner graph), where D is the diameter of the point set. In the clustered case there will be many such edges and thus only "long" edges have to be processed.

An interesting observation that can be seen in Figure 6.15(b) is that the number of shortest path queries performed by the improved greedy algorithm on uniformly distributed sets is considerably smaller than for the clustered points set, while the running time is almost the same or even larger for large t's and for uniformly distributed sets, see Figure 6.19 and Table 6.10. Consider the case when the input contains 10K points. The number of shortest path queries performed on

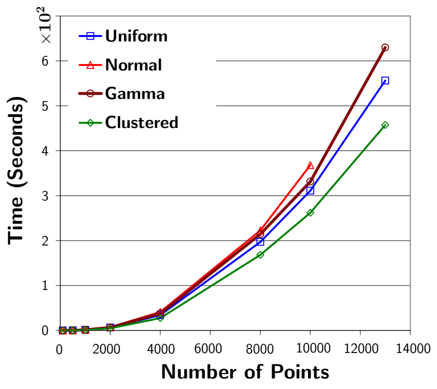


Figure 6.19: The running time of the improved greedy algorithm for $t = 2$ and with different distributions.

the uniform set is approximately 33K while it is about 57K for the clustered set. The number of clusters is 100, with 100 points per cluster. From the experiments it follows that the number of shortest-path queries performed between two points within the same cluster of uniformly distributed points is approximately 300. Since there are 100 clusters the number of shortest path queries needed for the "intra-cluster" edges in the clustered set is approximately 30K. These queries are all performed on very small graphs and are therefore processed extremely fast. Next approximately 27K "inter-cluster" queries are performed. We believe that the smaller number of "inter-cluster" queries together with the fact that the 2-spanner of the clustered set is slightly smaller than for the uniform set explains why the running times for the two different distributions are almost identical.

6.4 Concluding remarks

In short the conclusions from the experiments are as follows:

- The greedy graph has surprisingly good quality when it comes to the number of edges, the weight, the degree and the number of crossings. The diameter of the greedy graph is considerably higher than the diameter of the other graphs.

- The improved greedy algorithm worked much better than expected and it would be very interesting if one could prove that the improved greedy algorithm has an expected running time of $\mathcal{O}(n^2 \log n)$.

- The WSPD-algorithm produces graphs with unexpectedly poor quality for non-clustered sets. For clustered sets the results are much better; the weight and degree of the WSPD-graph are considerably smaller than the weight and degree of the (ordered) Θ-graphs. The weight and the degree of the WSPD-graphs seem to converge for very large data sets. However, to answer this conjecture we need to perform tests on much larger sets.

- The approximate greedy algorithm performs worse than expected in most cases, even though the theoretical bounds are very good.

- The Θ-graph construction algorithm is the fastest algorithm, however if it is important to obtain a high quality network then the improved greedy algorithms seems to be the most suitable choice.

- We also tested three hybrid algorithms each one being a combination of a fast, but with low quality, algorithm and greedy pruning. The graphs generated by these three hybrid algorithms have roughly the same properties as the greedy graph. We also have a parameter α for these algorithms which we can adjust based on the application. The higher α we choose the higher difference between the greedy graph and the graph generated by hybrid algorithms but lower time to construct the spanner.

The main question that remains to be answered experimentally is the dependency on the number of dimensions, i.e. how the algorithms and the quality of the produced graphs depends on the number of dimensions. Also it is interesting to improve the running time of the improved greedy algorithm. We can obtain an improvement by using algorithms which perform shortest path queries faster. So one parameter which could be used to measure the efficiency of all the algorithms is the number of shortest path queries performed by an algorithm.

The experiments show that the weight of the Θ-graph is very small for non-clustered sets. Proving, or disproving, that the expected weight of the Θ-graph is small for uniform distributions is an interesting and challenging open question.

Acknowledgements

We would like to thank the anonymous reviewers for comments on an earlier version of this chapter.

Chapter 7

Conclusions

In this thesis we concentrated on geometric spanner networks. The area attracted a lot of attention in past few years [Epp00, GK07, NS07, Smi00] and finds application in several branches like robotics, network topology design, metric space searching and bioinformatics. Below we briefly discuss the most important results from the thesis and the most interesting open problems that remain.

One of the major issues in designing a network is reliability. Reliability is concerned with the fact that in many applications the nodes and/or links in a network may fail. A network is reliable, or fault-tolerant, when it retains its good properties even after some nodes or links fail. With respect to spanners this means there should still be a short path between any two nodes in what remains of the spanner after the fault. The problem of designing k-vertex (or: k-edge) fault-tolerant geometric spanners, faults that can destroy an arbitrary collection of k vertices or edges, was studied before [CZ03, LNS98, LNS02, Luk99]. For geometric spanners, however, it is natural to consider faults that destroy all vertices and edges intersecting some geometric fault region. This is relevant, for instance, when the spanner models a road network and a natural (or other) disaster makes all the roads in some region inaccessible.

In Chapter 2 we introduced region-fault tolerant spanners and we proved that any planar point set admits a sparse t-spanner, of near-linear size, which is fault tolerant under any convex region fault; for several special cases or by adding Steiner points we could improve the size to linear. The main problem which remains open is whether one can improve the spanner size for arbitrary point sets to linear. We also considered fault-tolerant geodesic t-spanners: this is a variant where, for any disk D, the distance between any two points outside D in the remaining graph is at most t times the geodesic distance between them in $\mathbb{R}^2 \setminus D$. We proved that for any point set we can add Steiner points to obtain a fault-tolerant geodesic t-spanner of linear size. It would be interesting to generalize the problem to the case when we have curves as the edges in a network instead of lines. It would also be interesting to have a fast algorithm which checks whether a given network is fault tolerant under any convex region fault or not. Using a simple trick we obtain an algorithm which in $\mathcal{O}(n^5)$ time using $\mathcal{O}(n^2)$ space can find the answer for a network with n vertices.

Given a geometric network, natural questions to ask are how we can improve the quality of the network by adding one or more edges. Similarly, one can ask how we can remove an edge from the network such that the quality of the network does not decrease too much. In the case of extending the network, we studied the problem of finding the additional edge while maximizing the improvement, which in our case means minimizing the dilation of the resulting graph. Chapters 3 addressed the problem of optimal edge augmentation of a given geometric network. The main results are an exact algorithm and several approximation algorithms. The main approximation algorithm finds a $(2 + \varepsilon)$-approximation of the optimal edge. A challenging open problem is to improve the approximation factor of the approximation algorithms to $1 + \varepsilon$ with the same time complexity. Also, is it possible to find an exact algorithm which solves the problem in $o(n^4)$ time using linear space?

In Chapter 4 we considered the problem of finding/deciding an optimal edge deletion in a simple case. We showed that given a polygonal cycle in the plane, in near-linear expected time we can compute the edge of the cycle whose removal results in a polygonal path of smallest possible dilation. It is also shown that the edge whose removal gives a polygonal path of largest possible dilation can be computed in near-linear time; when the cycle is convex then we can find the edge in linear time. Generalizing the results to more general Euclidean graphs is a challenging open problem.

The spanner diameter of a t-spanner is the smallest integer $\mathbb{D}$ such that for any pair of vertices, there is a t-path in the graph between them containing at most $\mathbb{D}$ edges. As far as we know there is no known algorithm to compute the spanner diameter of a t-spanner. In Chapter 5 we presented several algorithms, using a dynamic programming approach, for computing the spanner diameter of a given t-spanner. The fastest algorithm computes the spanner diameter of a t-spanner with n vertices and m edges in $\mathcal{O}(\mathbb{D} \cdot mn)$ time using $\mathcal{O}(m + n)$ space, where $\mathbb{D}$ is the spanner diameter of the input network. The algorithms are purely combinatorial and do not take advantage of the fact that the graphs are embedded in Euclidean space. An interesting problem is to improve the complexity bounds in the geometric case.

The empirical study of algorithms is a rapidly growing research area. Implementing and doing experiments on algorithms shows their efficiency in practice and brings the algorithms to the practical stage. In Chapter 6, we compared the most well-known geometric t-spanner algorithms (in 2-dimension) experimentally. The experiments compared the quality of the generated graphs, like size, weight, maximum degree and diameter as well as the running time. We showed that the greedy algorithm generates graphs with surprisingly good quality when it comes to the number of edges, the weight, the degree and the number of crossings. However, the greedy algorithm is very slow; we gave several speedup strategies for the greedy algorithm. The Θ-graph algorithm is the fastest algorithm for constructing t-spanners. The major open problem is to investigate the effect of the number of dimensions on the quality of the graphs and the running time of the algorithms.

References

[AdBFG07] Mohammad Ali Abam, Mark de Berg, Mohammad Farshi, and Joachim Gudmundsson. Region-fault tolerant geometric spanners. In *SODA '07: Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1–10, 2007.

[ADD$^+$93] Ingo Althöfer, Gautam Das, David P. Dobkin, Deborah Joseph, and José Soares. On sparse spanners of weighted graphs. *Discrete and Computational Geometry*, 9(1):81–100, 1993.

[ADM$^+$95] Sunil Arya, Gautam Das, David M. Mount, Jeffrey S. Salowe, and Michiel Smid. Euclidean spanners: short, thin, and lanky. In *STOC '95: Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, pages 489–498, 1995.

[AFK$^+$07] Hee-Kap Ahn, Mohammad Farshi, Christian Knauer, Micheil Smid, and Yajun Wang. Dilation-optimal edge deletion in polygonal cycles. In *ISAAC '07: Proceedings of the 18th International Symposium on Algorithms and Computation*, volume 4835 of *Lecture Notes in Computer Science*, pages 88–99. Springer-Verlag, 2007.

[AGSS89] Alok Aggarwal, Leonidas J. Guibas, James Saxe, and Peter W. Shor. A linear-time algorithm for computing the Voronoi diagram of a convex polygon. *Discrete and Computational Geometry*, 4:591–604, 1989.

[AKK$^+$] Pankaj K. Agarwal, Rolf Klein, Christian Knauer, Stefan Langerman, Pat Morin, Micha Sharir, and Michael Soss. Computing the detour and spanning ratio of paths, trees and cycles in 2D and 3D. *Discrete and Computational Geometry*. to appear.

[ALW$^+$03] Khaled M. Alzoubi, Xiang-Yang Li, Yu Wang, Peng-Jun Wan, and Ophir Frieder. Geometric spanners for wireless ad hoc networks. *IEEE Transactions on Parallel and Distributed Systems*, 14(5):1–14, 2003.

[AMS99] Sunil Arya, David M. Mount, and Michiel Smid. Dynamic algorithms for geometric spanners of small diameter: Randomized solutions. *Computational Geometry: Theory and Applications*, 13(2):91–107, 1999.

[BGM04] Prosenjit Bose, Joachim Gudmundsson, and Pat Morin. Ordered theta graphs. *Computational Geometry: Theory and Applications*, 28:11–18, 2004.

[Cal95] Paul B. Callahan. *Dealing with higher dimensions: the well-separated pair decomposition and its applications.* Ph.D. thesis, Department of Computer Science, Johns Hopkins University, Baltimore, Maryland, 1995.

[CDNS95] Barun Chandra, Gautam Das, Giri Narasimhan, and José Soares. New sparseness results on graph spanners. *International Journal of Computational Geometry and Applications*, 5:124–144, 1995.

[Cha94] Barun Chandra. Constructing sparse spanners for most graphs in higher dimensions. *Information Processing Letters*, 51(6):289–294, 1994.

[Che86] L. Paul Chew. There is a planar graph almost as good as the complete graph. In *SCG '86: Proceedings of the 2nd Annual ACM Symposium on Computational Geometry*, pages 169–177, 1986.

[CHL07] Otfried Cheong, Herman Haverkort, and Mira Lee. Computing a minimum-dilation spanning tree is NP-hard. In *CATS '07: Proceedings of the 19th Computing: The Australasian Theory Symposium*, pages 15–24. CRPIT, 2007.

[CK93] Paul B. Callahan and S. Rao Kosaraju. Faster algorithms for some geometric graph problems in higher dimensions. In *SODA '93: Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 291–300, Philadelphia, PA, USA, 1993.

[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*, 42:67–90, 1995.

[CL00] Artur Czumaj and Andrzej Lingas. Fast approximation schemes for Euclidean multi-connectivity problems. In *ICALP '00: Proceedings* *of the 27th International Colloquium on Automata, Languages and Programming*, volume 1853 of *Lecture Notes in Computer Science*, pages 856–868. Springer-Verlag, 2000.

[Cla87] Kenneth L. Clarkson. Approximation algorithms for shortest path motion planning. In *STOC '87: Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 56–65, 1987.

[CNBYM01] Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and José Luis Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.

[CS88] Kenneth L. Clarkson and Peter W. Shor. Algorithms for diametral pairs and convex hulls that are optimal, randomized, and incremental. In *SCG '88: Proceedings of the 4th Annual ACM Symposium on Computational Geometry*, pages 12–17, New York, NY, USA, 1988. ACM Press.

[CZ03] Artur Czumaj and Hairong Zhao. Fault-tolerant geometric spanners. In *SCG '03: Proceedings of the 19th Annual ACM Symposium on Computational Geometry*, pages 1–10, New York, NY, USA, 2003. ACM Press.

[Dav05] Robert B. Davis. http://www.robertnz.net/nr03doc.htm, 2005.

[DDS92] Matthew T. Dickerson, R. L. Scot Drysdale, and Jörg-Rüdiger Sack. Simple algorithms for enumerating interpoint distances and finding k nearest neighbors. *International Journal of Computational Geometry and Applications*, 2:221–239, 1992.

[DGK01] Christian A. Duncan, Michael T. Goodrich, and Stephen Kobourov. Balanced aspect ratio trees: Combining the advances of k-d trees and octrees. *Journal of Algorithms*, 38:303–333, 2001.

[DHN93] Gautam Das, Paul J. Heffernan, and Giri Narasimhan. Optimally sparse spanners in 3-dimensional Euclidean space. In *SCG '93: Proceedings of the 9th Annual ACM Symposium on Computational Geometry*, pages 53–62, 1993.

[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.

[DNS95] Gautam Das, Giri Narasimhan, and José Salowe. A new way to weigh malnourished Euclidean graphs. In *SODA '95: Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 215–222, 1995.

[EP04] Michael Elkin and David Peleg. $(1+\epsilon,\beta)$-spanner constructions for general graphs. *SIAM Journal on Computing*, 33:608–631, 2004.

[Epp00] David Eppstein. Spanning trees and spanners. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 425–461. Elsevier Science Publishers, Amsterdam, 2000.

[Eri01] Jeff Erickson. Nice point sets can have nasty Delaunay triangulations. In *SCG'01: Proceedings of the 7th Annual ACM Symposium on Computational Geometry*, pages 96–105, New York, NY, USA, 2001. ACM Press.

[EW07] David Eppstein and Kevin A. Wortman. Minimum dilation stars. *Computational Geometry: Theory and Applications*, 37(1):27–37, 2007.

[FG05] Mohammad Farshi and Joachim Gudmundsson. Experimental study of geometric t-spanners. In *ESA '05: Proceedings of the 13th Annual European Symposium on Algorithms*, volume 3669 of *Lecture Notes in Computer Science*, pages 556–567. Springer-Verlag, 2005.

[FG06] Mohammad Farshi and Joachim Gudmundsson. On algorithms for computing the diameter of a t-spanner. In *AIMC 37: Proceedings of the 37th Annual Iranian Mathematics Conference*, pages 638–641, 2006.

[FG07] Mohammad Farshi and Joachim Gudmundsson. Experimental study of geometric t-spanners: A running time comparison. In *WEA '07: Proceedings of the 6th Workshop on Experimental Algorithms*, volume 4525 of *Lecture Notes in Computer Science*, pages 270–284. Springer-Verlag, 2007.

[FGG05a] Mohammad Farshi, Panos Giannopoulos, and Joachim Gudmundsson. Finding the best shortcut in a geometric network. In *SCG '05: Proceedings of the 21st Annual ACM Symposium on Computational Geometry*, pages 327–335, New York, NY, USA, 2005. ACM Press.

[FGG05b] Mohammad Farshi, Panos Giannopoulos, and Joachim Gudmundsson. Finding the best shortcut in a geometric network. In *EWCG '05: Proceedings of the 21st European Workshop on Computational Geometry*, pages 29–32, 2005.

[FHP05] John Fischer and Sariel Har-Peled. Dynamic well-separated pair decomposition made easy. In *CCCG '05: Proceedings of the 17th Canadian Conference on Computational Geometry*, pages 235–238, 2005.

[FKS84] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31:538–544, 1984.

[For87] Steven J. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[FPZW04] Arthur M. Farley, Andrzej Proskurowski, Daniel Zappala, and Kurt J. Windisch. Spanners and message distribution in networks. *Discrete Applied Mathematics*, 137(2):159–171, 2004.

[FR02] Stefan Funke and Edgar A. Ramos. Smooth-surface reconstruction in near-linear time. In *SODA '02: Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 781–790, 2002.

[FW94] Michael L. Fredman and Dan E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences*, 48(3):533–551, 1994.

[GGST86] Hall N. Gabow, Zvi Galil, Thomas H. Spencer, and Robert E. Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6(2):109–122, 1986.

[GK07] Joachim Gudmundsson and Christian Knauer. Dilation and detour in geometric networks. In Teofilo Gonzalez, editor, *Handbook on approximation algorithms and metaheuristics.* Chapman & Hall/CRC, Amsterdam, 2007.

[GKM07] Panos Giannopoulos, Christian Knauer, and Dániel Marx. Minimum-dilation tour (and path) is NP-hard. In *EWCG '07: Proceedings of the 23rd European Workshop on Computational Geometry*, pages 18–21, 2007.

[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.

[GLNS02a] Joachim Gudmundsson, Christos Levcopoulos, Giri Narasimhan, and Michiel Smid. Approximate distance oracles for geometric graph. In *SODA '02: Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 828–837, 2002.

[GLNS02b] Joachim Gudmundsson, Christos Levcopoulos, Giri Narasimhan, and Michiel Smid. Approximate distance oracles revisited. In *ISAAC '02: Proceedings of the 13th International Symposium on Algorithms and Computation*, volume 2518 of *Lecture Notes in Computer Science*, pages 357–368. Springer-Verlag, 2002.

[GP97] Nicolas Guex and Manuel C. Peitsch. SWISS-MODEL and the Swiss-PdbViewer: an environment for comparative protein modeling. *Electrophoresis*, 18(15):2714–23, 1997. `http://www.expasy.org/spdbv/`.

[Grü05] Ansgar Grüne. Tightness of upper bound on t_B/t_P. Manuscript, March 2005.

[HP98] Sariel Har-Peled. On the expected complexity of random convex hulls. *Technical Report 330/98, School Math. Sci., Tel-Aviv Univ., Tel-Aviv, Israel*, 1998.

[Kei88] J. Mark Keil. Approximating the complete Euclidean graph. In *SWAT '88: Proceedings of the 1st Scandinavian Workshop on Algorithm Theory*, volume 318 of *Lecture Notes in Computer Science*, pages 208–213. Springer-Verlag, 1988.

[KG92] J. Mark Keil and Carl A. Gutwin. Classes of graphs which approximate the complete Euclidean graph. *Discrete and Computational Geometry*, 7:13–28, 1992.

[KG01] Menelaos I. Karavelas and Leonidas J. Guibas. Static and kinetic geometric spanners with applications. In *SODA '01: Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 168–176, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

[Kir83] David G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM Journal on Computing*, 12:28–35, 1983.

[KK07] Rolf Klein and Martin Kutz. Computing geometric minimum-dilation graphs is NP-hard. In Michael Kaufmann and Dorothea Wagner, editors, *Graph Drawing, Karlsruhe, Germany, September 18-20, 2006*, pages 196–207. Springer, 2007.

[KKT95] David R. Karger, Philip N. Klein, and Robert E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *Journal of ACM*, 42(2):321–328, 1995.

[Li03] X.-Y. Li. Applications of computational geometry in wireless ad hoc networks. In X.-Z. Cheng, X. Huang, and D.-Z. Du, editors, *Ad Hoc Wireless Networking*. Kluwer, 2003.

[LL92] Christos Levcopoulos and Andrzej Lingas. There are planar graphs almost as good as the complete graphs and almost as cheap as minimum spanning trees. *Algorithmica*, 8:251–256, 1992.

[LNS98] Christos Levcopoulos, Giri Narasimhan, and Michiel Smid. Efficient algorithms for constructing fault-tolerant geometric spanners. In *STOC '98: Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 186–195, New York, NY, USA, 1998. ACM Press.

[LNS02] Christos Levcopoulos, Giri Narasimhan, and Michiel Smid. Improved algorithms for constructing fault-tolerant spanners. *Algorithmica*, 32:144–156, 2002.

[LP78] D. T. Lee and Franco P. Preparata. The all nearest-neighbor problem for convex polygons. *Information Processing Letters*, 7:189–192, 1978.

[LS95] Hans-Peter Lenhof and Michiel Smid. Sequential and parallel algorithms for the k closest pairs problem. *International Journal of Computational Geometry and Applications*, 5:273–288, 1995.

[Luk99] Tamás Lukovszki. New results of fault tolerant geometric spanners. In *WADS '99: Proceedings of the 6th Workshop on Algorithms and Data Structures*, volume 1663 of *Lecture Notes in Computer Science*, pages 193–204. Springer-Verlag, 1999.

[Mar02] Eric Martz. Protein explorer: easy yet powerful macromolecular visualization. *Trends in biochemical sciences*, pages 107–109, 2002. `http://proteinexplorer.org`.

[MN00] Kurt Mehlhorn and Stefan Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, UK, 2000.

[MS02] Sergei Maslov and Kim Sneppen. Specificity and stability in topology of protein networks. *Science*, pages 910–913, 2002.

[Nar02] Giri Narasimhan. Geometric spanner networks: Open problems. In *Invited talk at the* 1st Utrecht-Carleton Workshop on Computational Geometry, 2002.

[NG04] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical Review E*, page 026113, 2004.

[NP03] Gonzalo Navarro and Rodrigo Paredes. Practical construction of metric t-spanners. In *ALENEX '03: Proceedings of the 5th Workshop on Algorithm Engineering and Experiments*, pages 69–81. SIAM Press, 2003.

[NPC02] Gonzalo Navarro, Rodrigo Paredes, and Edgar Chávez. t-spanners as a data structure for metric space searching. In *Proceedings of the 9th International Symposium on String Processing and Information Retrieval*, volume 2476 of *Lecture Notes in Computer Science*, pages 298–309. Springer-Verlag, 2002.

[NS00] Giri Narasimhan and Michiel Smid. Approximating the stretch factor of Euclidean graphs. *SIAM Journal on Computing*, 30(3):978–989, 2000.

[NS07] Giri Narasimhan and Michiel Smid. *Geometric spanner networks.* Cambridge University Press, 2007.

[Pel00] David Peleg. *Distributed Computing: A Locality-Sensitive Approach.* SIAM, Philadelphia, PA, 2000.

[PS85] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, USA, 1985.

[PS89] D. Peleg and A. Schäffer. Graph spanners. *Journal of Graph Theory*, 13:99–116, 1989.

[Ras] Rasmol. `http://www.umass.edu/microbio/rasmol/`.

[RG05] Daniel Russel and Leonidas J. Guibas. Exploring protein folding trajectories using geometric spanners. *Pacific Symposium on Biocomputing*, pages 40–51, 2005.

[Ros98] Sheldon Ross. *A First Course in Probability.* Prentice-Hall, 5th edition, 1998.

[RS63] A. Rényi and R. Sulanke. Über die konvexe hülle von n zufällig gerwähten punkten. *I. Z. Wahrsch. Verw. Gebiete*, 2:75–84, 1963.

[RS91] Jim Ruppert and Raimund Seidel. Approximating the d-dimensional complete Euclidean graph. In *CCCG '91: Proceedings of the 3rd Canadian Conference on Computational Geometry*, pages 207–210, 1991.

[RS98] Satish B. Rao and Warren D. Smith. Approximating geometrical graphs via "spanners" and "banyans". In *STOC '98: Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 540–550. ACM, 1998.

[Rui86] E. V. Ruiz. An algorithm for finding nearest neighbours in (approximately) constant average time. *Pattern Recognition Letters*, 4(3):145–157, 1986.

[Sal91] Jeffrey S. Salowe. Constructing multidimensional spanner graphs. *International Journal of Computational Geometry and Applications*, 1:99–107, 1991.

[Sal92] Jeffrey S. Salowe. Enumerating interdistances in space. *International Journal of Computational Geometry and Applications*, 2(1):49–59, 1992.

[SH75] M.I. Shamos and D. Hoey. Closest point problems. In *FOCS '75:Proceedings of the 16th Annual IEEE Symposium on Foundations of Computer Science*, pages 151–162, 1975.

[Smi00] Michiel Smid. Closest point problems in computational geometry. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 877–935. Elsevier Science Publishers, Amsterdam, 2000.

[Soa94] José Soares. Approximating Euclidean distances by small degree graphs. *Discrete and Computational Geometry*, 11:213–233, 1994.

[SZ04] Mikkel Sigurd and Martin Zachariasen. Construction of minimum-weight spanners. In *ESA '04: Proceedings of the 12th Annual European Symposium on Algorithms*, volume 3221 of *Lecture Notes in Computer Science*, pages 797–808. Springer-Verlag, 2004.

[TZ01] Mikkel Thorup and Uri Zwick. Approximate distance oracles. In *STOC '01: Proceedings of the Annual ACM Symposium on Theory of Computing*, pages 183–192, 2001.

[Vai88] Pravin M. Vaidya. Minimum spanning trees in k-dimensional space. *SIAM Journal on Computing*, 17(3):572–582, 1988.

[Vai89] Pravin M. Vaidya. An $O(n \log n)$ algorithm for the all-nearest-neighbors problem. *Discrete and Computational Geometry*, 4:101–115, 1989.

[Vai91] Pravin M. Vaidya. A sparse graph almost as good as the complete graph on points in K dimensions. *Discrete and Computational Geometry*, 6(4):369–381, 1991.

[Var98] Kasturi R. Varadarajan. A divide-and-conquer algorithm for min-cost perfect matching in the plane. In *FOCS '98: Proceedings of the 39th Annual IEEE Symposium on Foundations of Computer Science*, pages 320–331, 1998.

[Wat05] Roger Wattenhofer. Algorithms for ad hoc and sensor networks. *Computer Communications*, 28:1498–1504, 2005.

[Yao82] Andrew C. Yao. On constructing minimum spanning trees in k-dimensional spaces and related problems. *SIAM Journal on Computing*, 11:721–736, 1982.

[ZADB06] Pavel Zezula, Giuseppe Amato, Vlastislav Dohnal, and Michal Batko. *Similarity Search - The Metric Space Approach.* Series: Advances in Database Systems , Vol. 32. Springer, 2006.

Summary

A Theoretical and Experimental Study of Geometric Networks

A geometric network on a set V of points in d-dimensional Euclidean space is a weighted undirected graph $G(V, E)$ with vertex set V such that the weight of an edge is the Euclidean distance between its endpoints. We say that the geometric network $G(V, E)$ is a *t-spanner* of V, for a $t > 1$, if for each pair of points u and v in V there exist a path in G between u and v of length at most t times the Euclidean distance between u and v. The *dilation*, or *stretch factor*, of a geometric graph G is the minimum t for which G is a t-spanner. We can easily extend the definition of a t-spanner of a point set to a t-spanner of a weighted graph. A graph $G'(V, E')$ on the same vertex set but with edge set $E' \subseteq E$ is a *t-spanner* of $G(V, E)$ if for each pair of vertices $u, v \in V$ there exist a path in G' between u and v of length at most t times the length of the shortest path between u and v in G.

This thesis contains theoretical and experimental results on geometric networks. In the theoretical part, we introduce the concept of region-fault tolerant spanners for planar point sets, and prove the existence of region-fault tolerant spanners of small size. For a geometric graph G on a point set V and a region F, we define $G \ominus F$ to be what remains of G after the vertices and edges of G intersecting F have been removed. A *$\mathcal{C}$-fault tolerant t-spanner* is a geometric graph G on V such that for any convex region F, the graph $G \ominus F$ is a t-spanner for $G_c(V) \ominus F$, where $G_c(V)$ is the complete geometric graph on V. We prove that every set V of n points admits a $\mathcal{C}$-fault tolerant $(1 + \varepsilon)$-spanner of size $\mathcal{O}(n \log n)$, for any constant $\varepsilon > 0$. If adding Steiner points is allowed then the size of the spanner can be reduced to $\mathcal{O}(n)$, and for several special cases we show how to obtain region-fault tolerant spanners of $\mathcal{O}(n)$ size without using Steiner points. We also consider *fault-tolerant geodesic t-spanners*; this is a variant where, for any disk D, the distance in $G \ominus D$ between any two points $u, v \in V \setminus D$ is at most t times the geodesic distance between u and v in $\mathbb{R}^2 \setminus D$. We prove that for any point set V we can add $\mathcal{O}(n)$ Steiner points to obtain a fault-tolerant geodesic $(1 + \varepsilon)$-spanner of size $\mathcal{O}(n)$.

We also include some results on optimal edge augmentation and optimal edge deletion on geometric networks. The goal is to add/remove an edge to/from a graph such that the dilation of the resulting graph is minimized. In edge augmentation case, we present one exact algorithm and several approximation algorithms. The best approximation algorithm computes a $(2+\varepsilon)$-approximation of the optimal solution in $\mathcal{O}(nm + n^2 \log n)$ time using $\mathcal{O}(n^2)$ space, where n is the number of vertices and m is the number of edges in the input network.

For the special case, when the dilation of the input network is constant, we can improve the approximation factor to $1+\varepsilon$ and the running time to $\mathcal{O}(n^2)$. For the problem of dilation optimal edge deletion, we solve the problem in a restricted case, when the network is a simple cycle. A randomized algorithm is presented which, given a cycle on a set of n points, computes in $\mathcal{O}(n \log^3 n)$ expected time, the edge of the cycle whose removal results in a polygonal path of smallest possible dilation. It is also shown that the edge whose removal gives a polygonal path of largest possible dilation can be computed in $\mathcal{O}(n \log n)$ time. If the input cycle is a convex polygon, the latter problem can be solved in $\mathcal{O}(n)$ time. Finally, it is shown that given a cycle C, for each edge e of C, a $(1-\varepsilon)$-approximation to the dilation of the path $C \setminus \{e\}$ can be computed in $\mathcal{O}(n \log n)$ total time.

Computing the spanner diameter of a t-spanner is another problem which is addressed in this thesis. We present an algorithm which computes spanner diameter of a t-spanner with n vertices and m edges in $\mathcal{O}(\mathbb{D} \cdot mn)$ time using $\mathcal{O}(m+n)$ space, where $\mathbb{D}$ is the spanner diameter of the input network.

Finally, we experimentally study the performance and quality of the most common t-spanner algorithms for points in the Euclidean plane. The experiments are discussed and compared to the theoretical results and in several cases we suggest modifications that are implemented and evaluated. The quality measurements that we consider are the number of edges, the weight, the maximum degree, the spanner diameter and the number of crossings. We compare the running time of the algorithms and suggest some improvements. This is the first time an extensive comparison has been made between the construction algorithms of t-spanners.

Curriculum Vitae

Mohammad Farshi was born on the 14th of June 1974 in Yazd, Iran. He graduated from Kheikhosravi high-school in Mathematics and Physics in 1992. He received his Bachelor, in Applied Mathematics (Computer Science), from Yazd University in 1996 and his Master, in Pure Mathematics (Functional Analysis), from Shiraz University in March 1999. He was then employed by Yazd University as an instructor at the Department of Mathematics. In 2002, he was granted a Ph.D. scholarship from Ministry of Science, Research and Technology of I. R. Iran and in January 2004 he started his Ph.D. studies within the Computer Science Department at the Eindhoven University of Technology (TU/e), the Netherlands. The results of his research have been accepted among the research community leading to several publications and to this thesis.

As from October 2007, Mohammad moved to Ottawa, Canada, to join the Computational Geometry Lab at Carleton University as a postdoc.

List of Notations

$\mathbf{d}(p,q)$	Distance between p and q in the metric space. 2
$\|pq\|$	Euclidean distance between p and q 9
$UDG(V)$	The unit disk graph of V 9
$G_c(V)$	Complete geometric graph on V 14
$\mathcal{C}$	The family of all convex sets in the plane 15
$\mathcal{G}\ominus F$	What remains from $\mathcal{G}$ after removing the part of $\mathcal{G}$ which lies inside fault area F 14
$\mathcal{D}$	The family of all disks in the plane 15,41
$\mathrm{CH}(A)$	The convex hull of the point set A 19
$\mathcal{H}$	The family of all half-planes in the plane 16
$\|A\|$	The number of elements in the set A 19
$d(\cdot,\cdot)$	(Euclidean)distance between objects 26
$\mathbf{radius}(D)$	The radius of a disk D 26
$\mathbf{length}(\Pi)$	The length of the path Π 31
$\mathcal{R}(\nu)$	The region associated with an (internal or leaf) node ν in a tree . 32
$\mathrm{pa}(\nu)$	The parent of ν in the tree 33
$P(\nu)$	The subset of points from P that are stored in the leaves of the subtree $\mathcal{T}_\nu$ rooted at ν 33

$\mathcal{T}_\nu$	The subtree of $\mathcal{T}$ rooted at ν	33
$N(\ell)$	The collection of nodes of weight class ℓ	33
$\mathrm{pa}(\ell,\nu)$	The ℓ-parent of ν	33
$\mathbf{diam}(R)$	The diameter of the region R	33
$\mathbf{area}(A)$	The area of the region A	36
∂R	The boundary of the region R	42
G_{opt}	A graph obtained from G by adding a dilation-optimal edge	47
t_{opt}	The dilation of G_{opt}	47
$\delta_G(u,v)$	The shortest path between u and v in graph G	47
$\mathbf{d}_G(u,v)$	The length of $\delta_G(u,v)$	48
$G_{\mathcal{B}}$	A graph obtained from G by adding a bottleneck edge	51
$t_{\mathcal{B}}$	The dilation of $G_{\mathcal{B}}$	51
$\Delta_G(p,q)$	The set of vertex pairs in $ne\mathcal{G}$ for which a shortest path between them in $\mathcal{G} \cup \{(p,q)\}$ passes through (p,q)	56
$\mathfrak{C}$	A polygonal cycle in the plane	67
$\delta^{\leq k}(p,q)$	The shortest path between p and q in the graph with at most k edges	86
$\mathbf{d}^{\leq k}(p,q)$	The length of $\delta^{\leq k}(p,q)$	86

Index

Titles in the IPA Dissertation Series since 2002

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects.* Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems.* Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems.* Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

R.J. Willemen. *School Timetable Construction: Algorithms and Complexity.* Faculty of Mathematics and Computer Science, TU/e. 2002-05

M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems.* Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. *Models of Molecular Computing.* Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems.* Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing.* Faculty of Mathematics and Natural Sciences, UL. 2002-09

D. Tauritz. *Adaptive Information Filtering: Concepts and Algorithms.* Faculty of Mathematics and Natural Sciences, UL. 2002-10

M.B. van der Zwaag. *Models and Logics for Process Algebra.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

J.I. den Hartog. *Probabilistic Extensions of Semantical Models.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12

L. Moonen. *Exploring Software Systems.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13

J.I. van Hemert. *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining.* Faculty of Mathematics and Natural Sciences, UL. 2002-14

S. Andova. *Probabilistic Process Algebra.* Faculty of Mathematics and Computer Science, TU/e. 2002-15

Y.S. Usenko. *Linearization in μCRL.* Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand.* Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition and construction*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Visser. *Generic Traversal over Typed Source Code Representations*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-03

S.M. Bohte. *Spiking Neural Networks*. Faculty of Mathematics and Natural Sciences, UL. 2003-04

T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing*. Faculty of Mathematics and Computer Science, TU/e. 2003-05

S.V. Nedea. *Analysis and Simulations of Catalytic Reactions*. Faculty of Mathematics and Computer Science, TU/e. 2003-06

M.E.M. Lijding. *Real-time Scheduling of Tertiary Storage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-07

H.P. Benz. *Casual Multimedia Process Annotation – CoMPAs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08

D. Distefano. *On Modelchecking the Dynamics of Object-based Software: a Foundational Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-09

M.H. ter Beek. *Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components*. Faculty of Mathematics and Natural Sciences, UL. 2003-10

D.J.P. Leijen. *The λ Abroad – A Functional Approach to Software Components*. Faculty of Mathematics and Computer Science, UU. 2003-11

W.P.A.J. Michiels. *Performance Ratios for the Differencing Method*. Faculty of Mathematics and Computer Science, TU/e. 2004-01

G.I. Jojgov. *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving*. Faculty of Mathematics and Computer Science, TU/e. 2004-02

P. Frisco. *Theory of Molecular Computing – Splicing and Membrane systems*. Faculty of Mathematics and Natural Sciences, UL. 2004-03

S. Maneth. *Models of Tree Translation*. Faculty of Mathematics and Natural Sciences, UL. 2004-04

Y. Qian. *Data Synchronization and Browsing for Home Environments*. Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05

F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-06

L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications*. Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications.* Faculty of Technology Management, TU/e. 2004-08

N. Goga. *Control and Selection Techniques for the Automated Testing of Reactive Systems.* Faculty of Mathematics and Computer Science, TU/e. 2004-09

M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs.* Faculty of Science, Mathematics and Computer Science, RU. 2004-10

A. Löh. *Exploring Generic Haskell.* Faculty of Mathematics and Computer Science, UU. 2004-11

I.C.M. Flinsenberg. *Route Planning Algorithms for Car Navigation.* Faculty of Mathematics and Computer Science, TU/e. 2004-12

R.J. Bril. *Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets.* Faculty of Mathematics and Computer Science, TU/e. 2004-13

J. Pang. *Formal Verification of Distributed Systems.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14

F. Alkemade. *Evolutionary Agent-Based Economics.* Faculty of Technology Management, TU/e. 2004-15

E.O. Dijk. *Indoor Ultrasonic Position Estimation Using a Single Base Station.* Faculty of Mathematics and Computer Science, TU/e. 2004-16

S.M. Orzan. *On Distributed Verification and Verified Distribution.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17

M.M. Schrage. *Proxima - A Presentation-oriented Editor for Structured Documents.* Faculty of Mathematics and Computer Science, UU. 2004-18

E. Eskenazi and A. Fyukov. *Quantitative Prediction of Quality Attributes for Component-Based Software Architectures.* Faculty of Mathematics and Computer Science, TU/e. 2004-19

P.J.L. Cuijpers. *Hybrid Process Algebra.* Faculty of Mathematics and Computer Science, TU/e. 2004-20

N.J.M. van den Nieuwelaar. *Supervisory Machine Control by Predictive-Reactive Scheduling.* Faculty of Mechanical Engineering, TU/e. 2004-21

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue.* Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures*. Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability*. Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions*. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments*. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression*. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages*. Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics*. Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication*. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data*. Faculty of Sciences, Division of

Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell*. Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting*. Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs*. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations*. Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data*. Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space*. Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices*. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization*. Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML*. Faculty of

Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models.* Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications.* Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols.* Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition.* Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems.* Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation.* Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures.* Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains.* Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use.* Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems*. Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Optimising System Behaviour in Time*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML*. Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery*. Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

B.S. Graaf. *Model-Driven Evolution of Software Architectures*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

A.H.J. Mathijssen. *Logical Calculi for Reasoning with Binding*. Faculty of Mathematics and Computer Science, TU/e. 2007-17

D. Jarnikov. *QoS framework for Video Streaming in Home Networks*. Faculty of Mathematics and Computer Science, TU/e. 2007-18

M. A. Abam. *New Data Structures and Algorithms for Mobile Data*. Faculty of Mathematics and Computer Science, TU/e. 2007-19

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy*. Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS*. Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems*. Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras*. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit*. Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks*. Faculty of Mathematics and Computer Science, TU/e. 2008-12

که تحت هر خرابی ناحیه‌ای به شکل محدب، پایاست. در برخی حالات خاص یا با اضافه کردن $\mathcal{O}(n)$ نقطه اشتاینر[۶] می‌توان تعداد یال‌ها را به $\mathcal{O}(n)$ کاهش داد.

همچنین نتایجی در زمینه افزایش/ کاهش بهینه[۷] یال در شبکه‌های هندسی ارائه می‌شود. هدف نهایی در این زمینه آن است که یک یال به شبکه اضافه یا یک یال از شبکه حذف شود به نحوی که تأخیر در شبکه حاصل کمینه شود. در حالت افزایشِ بهینه یال، یک الگوریتم دقیق و چند الگوریتم تقریبی برای محاسبه یال بهینه ارائه شده است. سریع‌ترین الگوریتم تقریبی، یک $(\varepsilon + ۲)$-تقریب برای یال بهینه در زمان $\mathcal{O}(mn + n^۲ \log n)$ با استفاده از $\mathcal{O}(n^۲)$ فضای حافظه محاسبه می‌کند، که در آن n تعداد رئوس و m تعداد یال‌های شبکه است. در حالتی که تأخیر در شبکه ورودی ثابت باشد می‌توان ضریب تقریب را به $\varepsilon + ۱$ و زمان اجرا را به $\mathcal{O}(n^۲)$ کاهش داد.

در حالتِ کاهش بهینه یال مسأله در حالتی که گراف ورودی یک دور[۸] در صفحه باشد بررسی شده است. یک الگوریتم تصادفی ارائه شده است که برای یک دور با n رأس در صفحه، یالی را که حذف آن کمترین افزایش در تاخیر مسیر حاصل ایجاد می‌کند را در زمان $\mathcal{O}(n \log^۳ n)$ محاسبه می‌کند.

محاسبه قطر[۹] یک t-پوشش مسأله دیگری است که در این پایان‌نامه به آن پرداخته شده است. نتیجه اصلی در این قسمت الگوریتمی است که قطر یک t-پوشش با n رأس و m یال را در زمان $\mathcal{O}(\mathbb{D} \cdot mn)$ با استفاده از $\mathcal{O}(m + n)$ فضای حافظه محاسبه می‌کند که در آن $\mathbb{D}$ قطرِ شبکه مورد نظر است.

در نهایت، آزمایشاتی روی الگوریتم‌های مطرح برای ساختن t-پوشش‌ها انجام شده‌است و شبکه‌های تولید شده توسط این الگوریتم‌ها و زمان اجرای این الگوریتم‌ها با هم مقایسه شده‌است. طبق بررسی انجام شده، این اولین بررسی تجربی با این وسعت روی الگوریتم‌های سازنده t-پوشش است.

---

[۶] Steiner point

[۷] optimal edge augmentation/deletion

[۸] cycle

[۹] diameter

یک مطالعه نظری و تجربی روی شبکه‌های هندسی

محمّد فرشی

چکیده

یک شبکه هندسی[۱] روی یک مجموعه V از نقاط در فضای d-بعدی حقیقی ($\mathbb{R}^d$) عبارت است از یک گرافِ وزن‌دارِ غیر جهتدار روی مجموعه رئوس V که وزن هر یال (u,v) در گراف با فاصله اقلیدسی بین u و v، یا $|uv|$، برابر است. شبکه هندسی $G(V,E)$ را یک t-پوشش[۲] برای مجموعه V می‌نامند، به ازای یک $t > ۱$، اگر به ازای هر دو رأس u و v در V، $d_G(u,v) \leq t \cdot |uv|$، که $d_G(u,v)$ طول کوتاهترین مسیر بین u و v در گراف G است. کوچکترین t که به ازای آن شبکه G یک t-پوشش است را تأخیر[۳] یا ضریب کشش[۴] شبکه G می‌نامند. تعریفِ t-پوشش برای یک مجموعه نقاط را به راحتی می‌توان به t-پوشش برای یک گرافِ وزن‌دار تعمیم داد. گراف $G'(V,E')$ را یک t-پوشش برای گراف $G(V,E)$، $E' \subseteq E$، می‌نامند اگر به ازای هر زوج از رئوس u و v در V، $d_{G'}(u,v) \leq t \cdot d_G(u,v)$.

این پایان‌نامه شامل یک سری نتایج نظری و تجربی روی پوشش‌های هندسی است. در قسمت نظری، مفهوم پوشش‌های پایا تحت خرابی‌های ناحیه‌ای [۵] برای مجموعه نقاط روی صفحه تعریف می‌شود و ثابت می‌شود برای هر مجموعه از نقاط، پوشش‌های پایا تحت خرابی‌های ناحیه‌ای محدب با تعداد یال‌های کم موجود است. برای گراف G روی مجموعه نقاط V از صفحه و یک ناحیه F از صفحه، فرض کنید $G \ominus F$ گرافی باشد که از حذف تمام رئوس و یال‌های G که در ناحیه F قرار می‌گیرند بدست می‌آید. گراف G را یک t-پوشش پایا تحت خرابی‌های محدب می‌نامند اگر به ازای هر ناحیه محدب F، گرافِ $G \ominus F$ یک t-پوشش برای $G_c(V) \ominus F$ باشد که $G_c(V)$ گراف کامل روی V است. نتیجه اصلی در این قسمت الگوریتمی است که برای هر مجموعه متشکل از n نقطه از صفحه، پوششی با $\mathcal{O}(n \log n)$ یال ایجاد می‌کند

[۱] geometric network
[۲] t-spanner
[۳] dilation
[۴] stretch factor
[۵] region fault tolerant spanners