Experimental Study of Kinetic Geometric *t*-Spanner Algorithms

Master thesis by
Jonas Suhr Christensen, 20032491
jsc@umbraculum.org

Supervised by
Gerth Stølting Brodal

Department of Computer Science
Aarhus University
Denmark

31 August, 2010

Abstract

In this thesis we describe, implements and experiments with two algorithms that constructs and maintains geometric t-spanners. The algorithms are the *Kinetic Spanner in* $\mathbb{R}^d$ by Abam *et al.* and the *Deformable Spanner* by Gao *et al.* The implementations has only been done in $\mathbb{R}^2$.

The experiments are focused on two different aspects of the algorithms namely the qualities of the spanners constructed by the algorithms and the kinetic qualities of the algorithms. The results of the experiments are compared with other known non-kinetic spanner algorithms and a kinetic Delaunay triangulation.

The results shows that the qualities of the spanners in general is worse than the spanners constructed by the non-kinetic algorithms. However on input with specific characteristics properties of the constructed spanners are on level with spanners of the non-kinetic algorithms.

The experiments on the kinetic qualities can be summed up to that the kinetic spanner in $\mathbb{R}^d$ is a lot more expensive to maintain than the kinetic Delaunay triangulation. The deformable spanner is not a part of experiments on the kinetic properties and the experiments do not consider the responsiveness of the kinetic spanner in $\mathbb{R}^d$.

Acknowledgements

I would like to thank Mohammad Ali Abam for guidance on implementing and understanding the kinetic algorithms. Furthermore I would like to thank Gerth Stølthing Brodal for advises doing the process of writing the thesis.

Contents

Chapter 1

Introduction

A general problem in graph theory is to optimize a solution with respect to some property. E.g. to create a systems of roads such that it is possible to get to and from selected cities in an area or create a network of routers such that every router is connected. In the road example a economic property is that the total length of the roads are minimized, a property of convenience is that the roads has as few intersections as possible or in the router example it is important to minimize the number of routers a package has to pass to avoid high latency.

A straightforward property for graph problem of this nature is to put an upper limit on the length between two elements in the graph representation of the problem. E.g. when driving between two cities one will at most accept some penalty on the extra length of the trip compared to the beeline between the start and end location. In the graph representation this is equal to that the sum of the length of the edges connecting the two vertices that represents the cities has an upper limit. Problems with an upper limit defined like this can be solved by constructing *k-spanners*. If the length of the edges corresponds to the length between points in Euclidean space the spanners are called *geometric t-spanners*. Algorithms that constructs *geometric t-spanners* has been researched since the eighties and there exists many different approaches each with its own characteristics in form of properties such as those mentioned above.

However a new challenge came around. What do we do if we want to solve these problems when they are defined on things that does not stay constant but moves around? This could be that we want to keep a roaming mobile phone connected to some home base. It is easier on the battery to communicate over a short distance so we prefer to communicate with bases that are in a short distance of the phone. However communicating through other bases introduces some delay and it is preferable that communication is not delayed to much. As long as it is possible to send a message through other bases without exceeding some maximum delay time we do that. But if the delay time is exceeded we just send it directly to the home base or finds another base to communicate through. In this example a solution has to be updated continuously as the mobile phone roams around between the bases.

There exists algorithms to effectively handle the problem of maintaining solutions in such continuously changing environment. These algorithms are called *Kinetic Data Structure* and in the case of spanners kinetic spanners. This is algorithms that focus on how to detect when the structure needs to be updated and how handle updates in an efficient way.

1.1 Overview of the Thesis

This thesis will consider two different kinetic spanners; *A Kinetic Spanner in* $\mathbb{R}^d$ by Abam and Berg [2] and the *Deformable Spanner* by Gao *et al.* [14] both are valid in $\mathbb{R}^d$.

The two kinetic spanners will be implemented in $\mathbb{R}^2$ and experimented with on data with different characteristics. The experiments will be structured into two parts; a part that experiments with the properties of the spanner constructed by the algorithms and a part that experiments with the kinetic properties of the algorithms. The results of the experiments will be compared to the theoretical bounds of the algorithms and will be compared to other spanners from the literature.

The thesis is organized as follows:

- **Chapter 2** gives a formal definition of *geometric t-spanners*, properties of geometric t-spanners and a brief history of *geometric t-spanners*.
- **Chapter 3** considers *kinetic data structures* in general, properties of Kinetic Data Structures and a brief history of *kinetic data structures* with focus on kinetic spanners.
- **Chapter 4** defines the *Kinetic Spanner in* R^d, concerns some properties of the constructed spanner, explains how to maintain it and considers its kinetic properties.
- **Chapter 5** defines the *Deformable Spanner*, concerns some properties of the constructed spanner, explains how to maintain it and considers its kinetic properties.
- **Chapter 6** explains and considers the implementation of the kinetic spanners.
- **Chapter 7** presents the experiments and results on spanner properties.
- **Chapter 8** presents the experiments and results on kinetic properties.

- **Chapter 9 and 10** considers what can be further improved and explored and concludes the thesis.

The source code can be found at:
`http://www.umbraculum.org/esox/esox.tgz`
The data for the experiments can be found at:
`http://www.umbraculum.org/esox/experiments.tgz`
The source code and data can also be found on the enclosed DVD.

Part I

Fundamental Theory

Chapter 2

Geometric t-Spanners

This chapter will consider *geometric t-spanners*. Section 2.1 defines a *geometric t-spanner*. Section 2.2 introduces a set of properties that can be used to measure the quality of *geometric t-spanners* and finally Section 2.3 gives a brief view of the development of *geometric t-spanner* algorithms with focus on the properties of the spanners constructed by these algorithms.

2.1 Geometric t-Spanners

Given a set of points P' in Euclidean space. A *geometric spanner* is a graph $S = (P, E)$ consisting of a set of vertices P and edges E such that every point $p \in P'$ corresponds to a vertex $v \in P$ and for every pair of vertices (u, v) there is at least one path in S between the two points. An edge $(u, v) \in E$ has weight corresponding to the Euclidean distance $|uv|$ between the points. A path between two points $u, v \in P$ has a cost equal to the sum of the weight of the edges contained in the path.

It is possible to find the minimal k such that every pair of points in S are connected by some path with cost at most $k \cdot |uv|$. This k is called the *stretch-factor* or *dilation* of S represented by t. The graph that contains an edge $(u, v) \in E$ for every pair of points has $t = 1$ and is called a *complete graph*. If $t > 1$ then the graph S may be a sub-graph of the complete graph. A *geometric t-spanner* or *geometric* $(1 + \varepsilon)$*-spanner* is a spanner S_t such that the stretch-factor of S_t is at most t.

2.2 Properties of Spanners

The quality of a spanner can be measured by looking at different properties and is normally a trade-off between properties. Due to this, defining a "best" spanner depends on the purpose of the spanner. In some applications it is important to minimize the total number of edges but it is still appreciated to have paths of few edges or low cost between the nodes. This is an example of a trade-off between properties as adding more edges will lower the distance of the paths and vice-versa.

Some common properties for measuring the quality of a spanner are the number of edges, the degree of the nodes in the spanner, the weight of the edges and the diameter of the graph. This thesis will use the same definitions of spanner properties as Farshi and Gudmundsson in *Experimental Study of Geometric t-Spanners* [13]:

- **Size** — The size of a spanner is the number of edges in the spanner graph.
- **Degree** — The degree of the spanner is the maximum number of edges incident to a point in the spanner.
- **Weight** — The weight of a spanner is the sum of the weight of the edges in the spanner graph. In the context of Euclidean spanners the weight of an edge between two points $u, v \in S$ is defined as the Euclidean distance $|uv|$ between the points.
- **Diameter** — The diameter $d_{p,q}$ between two points p, q is the smallest number of edges one has to follow to get from p to q, ie. the number of edges in the path consisting of fewest edges between p and q. The diameter d of a spanner S is the minimal number such that every pair of points $p, q \in S$ are connected by a path consisting of at most d edges.

2.3 Geometric t-Spanners in Historical View

Geometric spanners were first introduced into the field of computational geometry by Chew [9] in the middle eighties. In this paper Chew investigated the stretch-factor for triangulations with different convex distance functions and showed that there exists a geometric 2-spanner for any input in $\mathbb{R}^2$. Dobkin *et al.* [12] later showed that the stretch-factor for a standard Delaunay triangulation is less than $t \approx 5.08$. Later it was showed that the stretch-factor of a standard Delaunay triangulation is $t \approx 2.42$ [19]. The Delaunay triangulation has $O(n)$ edges and maximum degree $O(n)$.

Algorithms for constructing t-spanners have since been developed such that the stretch-factor is a parameter to the algorithm. In 1989 Althöfer [4] and Bern introduced an algorithm independently that constructs spanners with $O(n)$ edges, constant maximum degree and weight depending on the weight of the minimum spanning tree $wt(MST)$. This algorithm is referred to as *the greedy algorithm*.

Clarkson [10] and Keil [18] independently introduced the concept of Θ-*graphs* in the last half of the eighties. However their algorithms only worked in two and three dimension respectively. The approach were later generalized into higher dimensions by Altöfer *et al.* [4] and Ruppert and Seidel [21] in the beginning of the nineties. The Θ-*graph* algorithm produces a t-spanner for $t = \frac{1}{\cos\theta - \sin\theta}$ with $O(\frac{n}{\theta})$ edges, maximum degree $\Theta(n)$ and weight $\Theta(n{\cdot}\mathrm{wt}(\mathrm{MST}))$. For more information about Θ-graphs see Section 4.1.

A variant of the Θ-*graph* is the *ordered* Θ-*graph* introduced by Bose *et al* [7] around ten years later. An *ordered* Θ-*graph* has stretch-factor similar to the "normal" Θ-*graph*. It has $\Theta(\frac{n}{\theta})$ edges, maximum degree $O(\frac{1}{\theta} \cdot \log n)$ and weight $O(n{\cdot}\mathrm{wt}(\mathrm{MST}))$.

Callahan and Kosaraju [8] showed in the mid-nineties that by computing the *well-separated pair decomposition* of the input set and adding an edge between each *well-separated pair* in the decomposition one has a geometric t-spanner. The WSPD-graph is a t-Spanners that have $\Theta((\frac{t}{(t-1)})^2 \cdot n)$ edges, maximum degree $\Theta(n)$ and weight $\Theta(\log n{\cdot}\mathrm{wt}(\mathrm{MST}))$.

Farshi and Gudmundsson [13] experimented with the above algorithms showing properties and trade-offs for the constructed spanners on input sets with different characteristic. This paper is the fundamental basis of the experiments in Section 7 and will be used as comparison for the qualities of the spanners constructed by the kinetic spanner algorithms. Table 7.1 in Section 7 summarize the properties of the different algorithms.

Chapter 3

Kinetic Data Structures

This chapter is about kinetic data structures. Section 3.1 presents the difference between handling static data and mobile data. Section 3.2 introduces the basic concepts and terminology of kinetic data structures. Section 3.3 presents the four properties that are used to analyze kinetic data structures. Section 3.4 gives a brief historical view of kinetic data structures and kinetic spanners.

3.1 Static And Mobile Data

In the previous chapter all points were defined as fixed points in Euclidean space. This type of data will be referred to as *static data*. In a kinetic setting points are not fixed instead they move along trajectories in the space. By evaluating trajectories with a parameter referred to as *time* the positions of the points in an instant moment can be found. This type of data will be referred to as *mobile data*.

A data structure that works in this continuously changing environment needs to be able to do fast updates when its *attribute* becomes invalid due changes in the topology. The *attribute* of a structure is in this thesis the problem that the structure solves e.g. a Delaunay triangulation, the convex hull or a t-spanner. When data structures working on static data becomes invalid they either has to be reconstructed or if they are dynamic update the part of the structure that failed by deleting or reinserting the point(s) that caused the invalidation.

A difficult thing is to know when the data structure becomes invalid and find the point(s) that cause(s) the invalidation. A straightforward solution is to check the structure every time it is used. However this solution has the tedious effect that potentially a lot of work has to be done exactly when the structure is needed — at the very moment where it is supposed to be fast.

Another approach is to fix intervals in advance, this can be at some fixed frequency or some clever pre-calculated intervals. However the fixed frequency updating approach is clearly broken, it does not guarantee that the data structure is valid in all moment of time. E.g. if a structure turns invalid at time i and the structure is first updated at time $i+j$ the structure is invalid in the interval $[i; j[$. If it is assumed that the pre-calculated intervals are exactly when there are changes in the topology, this approach seems well but lacks the ability to handle the dynamic of a "real" environment e.g. when trajectories are changed due to an event in the environment.

Kinetic data structures encapsulates an effective way of detecting when the data structure becomes invalid in an continuously changing environment and considers a way to handle this knowledge so that the structure can be efficiently updated exactly when it becomes invalid.

3.2 Basics of Kinetic Data Structures

Basch [6] proposed a model for analyzing and handling data structures working on mobile data or as it is written in his thesis "keeping track of discrete attributes of moving data" [6, page 15] called *Kinetic Data Structures*.

The basic thought is to have a set of *proofs* that ensures that the attribute of a data structure is valid. A proof is build around the position of the mobile data and can be evaluated as a function of time. It is possible by evaluating the proof set to find the correct time of when the data structure changes to an invalid state. By defining data structures such that it is easy to define and update proofs the data structures can be constructed such that it can handle continuously changing environments.

In short a kinetic data structure is a structure that maintains some attribute on an input set, contains a proof set that ensures that this attribute stays valid and has a system that updates the structure when proofs are failing.

Below are the terminology and concepts of kinetic data structures explained in more details.

3.2.1 Mobile Data

Before it make sence to talk about data structures working on mobile data it is necessary to explain what mobile data is.

Static data are fixed by some positions in a space e.g. a point represented by a vector. *Mobile data* is represented by a trajectory function f evaluated over time t. The function $f(t)$ maps some data to a position in space depending on the time t e.g. a mobile point could be the point moving at the trajectory specified by the linear function $f(t) = 2x + 2$. It is not given that a trajectory stays constant through the whole lifespan of a data structure. If trajectories changes it implies that the structure has to be updated according to the new trajectories this is called *motion plan update*.

3.2.2 Certificates and Events

An important part of a kinetic data structure is to generate proofs, called *certificates*, that ensure that the structure is valid with respect to some attribute. A certificate is a function of time that *fails* or *expires* when the sign of its outcome change. E.g. consider a kinetic data structure maintaining a priority queue based on a sorted list where the input objects $p \in P$ are sorted after a value v_p (from high to low). The certificate set for this structure consists of $n - 1$ certificates of the type $v_i(t) - v_{i+1}(t) > 0$ where $0 \leq i < n - 1$.

When a certificate fails an *event* corresponding to the type of the failed certificate are triggered. A failure of a certificate implies not only updating the structure maintaining the attribute but also requires changes to the certificate set. In the example above the event would be to update the structure by swapping the two points in the list, change the certificates that involved the swapped objects, so that they corresponds to the new order of the list, and finally update the registered failure times of the changed events.

There are two kinds of events *internal* and *external* events. External events are triggered by topological changes in the input set and are common for all data structures maintaining this attribute. In the priority queue example external events are when the head of the queue is updated. Internal events are related to the kinetic data structure itself. These events are part of the kinetic data structure and are not specific for maintaining the attribute. In the priority queue example internal events are when two objects that are not in the head of the queue are swapped. These events do not change the attribute of the data structure but are necessary due to the way that the priority queue works.

3.2.3 Event Queue

Another part of a kinetic data structure is to generate an event system that keeps track of failing certificates. As the trajectories of the elements are partially known in advance the failing time of each certificate can be calculated.

A common approach is to use a priority queue (sorting from low to high) for storing events with a key corresponding to the failure time of certificate that triggers the event. This is called an *event queue*. By doing this events can be triggered one by one as the time increase and the problem of detecting when the data structure changes into a invalid state is reduced to popping events from the event queue.

3.3 Kinetic Properties

When working with kinetic data structures there are two obvious properties to measure the quality by; the number of certificate failures and the complexity to restore the data structure into a valid state when a certificate fails. In addition it is also desirable that each object is involved in a minimal number of certificates so certificate updates and motion plan updates can be handled effectively.

The kinetic quality of a kinetic data structure can be described by the following four properties; *responsiveness*, *compactness*, *locality* and *efficiency*. If a kinetic data structure performs well with respect to all four properties it will be referred to as "good".

3.3.1 Responsiveness

When a certificate fails the data structure is invalid and a number of update operations are needed to restore it. The certificate set have to change by adding, changing and/or deleting certificates. Furthermore when changes happens to the certificate set the event queue also has to be updated.

The *responsiveness* is the complexity of handling this — from when a certificates fails to when the kinetic data structure is fully restored. A kinetic data structure is called *responsive* if the complexity is asymptotic poly-logarithmic.

3.3.2 Compactness

The certificate set of a kinetic data structure should be small. It is desirable that a kinetic data structure do not require many more certificates compared to the input set.

The *compactness* of a kinetic data structure is the maximum number of certificates that a structure needs at any a given time. If the size of the certificate set of a kinetic data structure is always near-linear in the number of input objects it is called *compact*.

3.3.3 Locality

When objects are inserted into kinetic data structures they will be be part of a number of certificates. If objects are later removed or undergoes motion plan updates all these certificates have to be either removed from the kinetic data structure or updated according to the new trajectory.

The *locality* of a kinetic data structure is the maximum number of certificate changes a single object can trigger i.e. the maximum number of certificates that depends on the object. If this number is asymptotic poly-logarithmic the kinetic data structure is called *local*.

3.3.4 Efficiency

As certificates in a kinetic data structure fail events are triggered. Some of these are internal events others are external events. The external events are the minimum number of events the kinetic data structure has to trigger to keep its attribute valid — the number of topological changes. The internal events are the additional events added by the kinetic data structure to keep its state intact.

The *efficiency* is the relation between the number of the two types of events. A kinetic data structure is called efficient if the total number of processed events are small compared to number of topological changes. More precisely a kinetic data structure are *efficient* if the ratio between the worst case number of internal events and the worst case number of external events is poly-logarithmic.

3.4 Kinetic Data Structures and Spanners in Historical View

A lot of study on how to maintain a certain attribute on moving data was done in the early eighties. In the mid-eighties Atallah [5] introduced a framework for handling moving "dynamic" data. However he assumed a constant environment with constant trajectories which is insufficient when simulating a "real" dynamic environment.

Basch and Guibas [16, 17, 6] introduced the concept of kinetic data structures in the last half of the nineties. A framework suited for analyzing and developing data structures and algorithms on mobile data in a compared to Atallahs framework more dynamic environment. By letting data structures be on-line and robust to missing trajectory information and motion plan updates the framework is designed to handle simulation of "real" dynamic environments.

Besides introducing the model they also introduced a number of kinetic data structures concerning maximum maintenance. But also two-dimensional problems such as a kinetic data structure maintaining the convex hull and the closest pair for a set of points.

Spanners working in this framework are called kinetic spanners. They were first (According to [2]) studied by Gao *et. al* [14]. They studied the problem of maintaining spanners on moving objects that has a high degree of influence on each others trajectories e.g. by collision. They showed how to construct "good" kinetic $(1+\varepsilon)$-spanners in $\mathbb{R}^d$ under certain assumption on the input set (see *aspect ratio* Section 5.1). The spanner has $\mathrm{O}(\frac{n}{\varepsilon^d})$ edges and a maximum degree $O(\log_2 \frac{n}{\varepsilon^d})$.

Abam *et al.* introduced the simple and efficient spanner [3] that only works in $\mathbb{R}^2$. This spanner is responsive, effective and compact, but not local. They use the property that the the Delaunay triangulation of the input set has a linear number of edges in $\mathbb{R}^2$. This property however is only valid in $\mathbb{R}^2$ thus the spanner is restricted to two dimensions. The spanner has $O(\frac{n}{\varepsilon^2})$ edges but as it is based on Delaunay triangulation the maximum degree is $O(n)$.

Abam and Berg [2] later introduced another spanner algorithm which was capable of being generalized into R^d. The spanner is "good" in kinetic sence and has size $\mathrm{O}(\frac{n}{\varepsilon^{d-1}})$ and a maximum degree on $O(\log^d n)$. This spanner is the first (According to [2]) known "good" kinetic spanner maintaining points in R^d that do not depend on assumption on the input points (as it is the case with the spanner proposed by Gao *et al.* [14]).

The kinetic spanner in $\mathbb{R}^d$ by Abam and Berg [2] and the spanner by Gao *et al.* [14] is implemented as a part of this thesis. More details about these two spanners can be found in chapter 4 and 5.

Part II

Kinetic Geometric t-Spanners

Chapter 4

Kinetic Spanner in $\mathbb{R}^d$

In [2] Abam and Berg presents a kinetic $(1+\varepsilon)$-spanner working in $\mathbb{R}^d$ called *Kinetic Spanner in* $\mathbb{R}^d$. It is "good" in kinetic sence; it is responsive handling events in $O(\log^{d+1} n)$ time. It has locality of $O(\frac{1}{\varepsilon^{d-1}})$ certificates per point and the total number of events is $O(\frac{n^2}{\varepsilon^{d-1}})$[1]. The spanner has size $O(\frac{n}{\varepsilon^{d-1}})$ and maximum degree $O(\log^d n)$.

This chapter presents the theory of the kinetic spanner with focus on $\mathbb{R}^2$ but keeps the results in $\mathbb{R}^d$. As the spanner is based on Θ-graphs theory [10, 18, 20] Section 4.1 briefly explains the idea behind Θ-graphs. Section 4.2 introduces the theory of the spanner. Section 4.3 explains how the theory can be used to construct a kinetic spanner and Section 4.4 explain the certificates and events required to maintain the spanner.

4.1 Θ-Graphs and Spanners

The Θ*-graph* approach is a way to connect a point set P by adding edges from every $p \in P$ to another point $q \in P$ such that q is the nearest point to p in a subset that contains all points that are inside a *cone* emanating from p.

The *cones* in a Θ-graph are are called θ*-cones* and can be defined as follows in $\mathbb{R}^2$. Define an angle θ and draw $O(\frac{1}{\theta})$ cones emanating from Origin. The cones can at most span over an angle θ, none of the cones can overlap and together they cover the whole space. A translation of a cone σ to a point p such that σ emanates from p is denoted $\sigma(p)$ and is called a *range*. The range that $\sigma(p)$ defines contain the points that are inside $\sigma(p)$.

From a set C containing θ-cones created as described above for some θ and a point set P we can construct a spanner containing at most $O(|C| \cdot n)$ edges. This can be done by for every range $\sigma(p)$ connecting p to a point $q \in \sigma(p)$ such that q is the point nearest to p. If θ are chosen such that

[1]Assuming that trajectories are bounded-degree polynomials.

$\cos\theta - \sin\theta \geq \frac{1}{1+\varepsilon}$ this spanner is a $(1+\varepsilon)$-spanner. For further information on Θ-graphs see [20].

4.2 A Kinetic Spanner in $\mathbb{R}^d$

A basic concept of the kinetic spanner in $\mathbb{R}^d$ is to avoid using the Euclidean distance function by measuring the distance by a faster and easier maintainable function. This section will define this distance function and show that it is possible to create a variant of Θ-graphs using the distance function such that the constructed graph is a $(1+\varepsilon)$-spanner.

4.2.1 A Maintainable Distance Function

As described above the edges of a Θ-graph are created by locating the closest points to a point p for each subset created by the cones emanating from p. However locating the Euclidean nearest point for every combination of points and cones is costly and hard to maintain in a kinetic environment [2, page 44]. We will therefore define and use another distance function $dist_\sigma(p,q)$ with the property that it fits into the Θ-graph approach and is easily maintained.

When locating the closest point to p in a point set of a range we know that the set has been created with respect to a θ-cone that emanates from p i.e. we always measure the distance between points in relation to a θ-cone. This observation can be used and we can define a distance function that depends on the θ-cones it is used with.

For every $\sigma \in C$ one of the lines defining the cone are chosen and is called the *representative line* of σ. The distance between two points p, q can then be measured as the distance between the projection of the two points onto the representative line. The distance function is denoted $dist_\sigma(p,q)$.

By using this distance function finding closest points is reduced to a matter of keeping track of the positions of the points projection onto the representative edge for every cone $\sigma \in C$. This can easily be done by storing the position of the points projections in a kinetic sorted list (see Section 4.4.1 for more information about kinetic sorted lists). Hence we have a easy maintainable structure from which it is easy to find the closest points (the neighbors in the list).

4.2.2 The Fundamental Lemma

The fundamental property of the spanner is lemma 2.1 from [2]. The lemma says that it is possible to construct $(1-\varepsilon)$-spanners from the θ-graphs approach using $dist_\sigma(p,q)$.

Lemma 1. *For a set C of θ-cones chosen such that $\cos 2\theta - \sin 2\theta \geq \frac{1}{1+\epsilon}$ and a point set P. If there for a point $p \in P$ and a cone $\sigma \in C$ exists two*

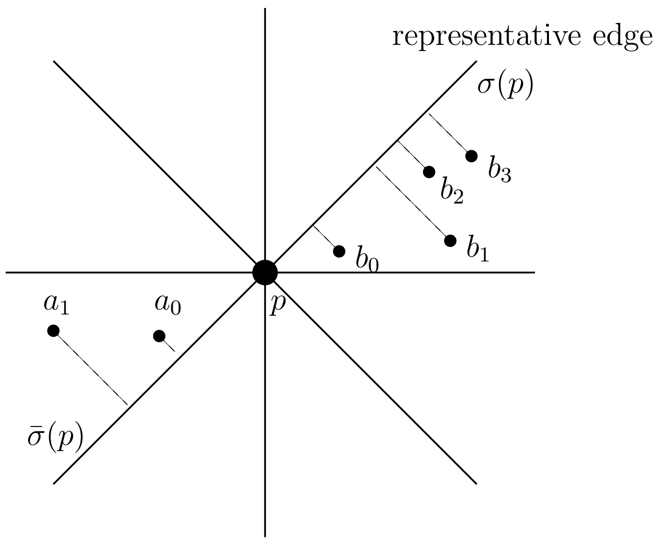


Figure 4.1: Showing pair $(A_i, B_i) = (\{a_1, a_0\}, \{b_0, b_1, b_2, b_3\})$ of Ψ_σ separated by a point p.

points q, r in $\sigma(p)$ such that $dist_\sigma(p, r) \leq dist_\sigma(p, q)$ then the path from p to q going through r has cost at most $(1 + \varepsilon) \cdot |pq|$.

With a structure preserving lemma 1 we can construct a $(1+\varepsilon)$-spanner by replacing a direct edge between two points p and q with a path going through r.

4.2.3 Cone-Separated Pairs of Decomposition

To construct a structure that preserves lemma 1 we use a decomposition of the input points called *cone-separated pair decomposition* (*CSPD*). This decomposition is based on a cone σ and its *opposite cone* denoted $\bar{\sigma}$. In $\mathbb{R}^2$ the opposite cone $\bar{\sigma}$ is created from mirroring σ by following its lines after the crossing in Origin. Figure 4.1 shows a cone σ and its opposite cone $\bar{\sigma}$ in $\mathbb{R}^2$.

Definition. *A CSPD with respect to a cone σ denoted Ψ_σ is a set $\Psi_\sigma = \{(A_1, B_1), ..., (A_m, B_m)\}$ of pairs of subsets of the point set P such that for every pair of points $(p, q) \in P$ if $q \in \sigma(p)$ then there is a pair $(A_i, B_i) \in \Psi_\sigma$ such that $p \in A_i$ and $q \in B_i$. Furthermore Ψ_σ may only contain pair such that if $p \in A_i$ and $q \in B_i$ then $q \in \sigma(p)$ and $p \in \bar{\sigma}(q)$.*

By this definition a collection of CSPDs for a cone set C and a point set P contains $|C|$ (one for each $\sigma \in C$) CSPDs each having a number of pairs depending on the topology of P. Figure 4.1 shows a pair in a CSPD.

A collection of CSPDs can be constructed by using a *rank based range tree* as explained in Section 4.3.

4.2.4 Well-Connected Pair of Points

It remains to explain how to construct a spanner from a collection of CSPDs such that lemma 1 are satisfied. To do this we connect the points of the pairs of a CSPD such that they are *well-connected*.

Definition. *For a spanner $S = (E, P)$ and a CSPD Ψ_σ.*
A pair $(A_i, B_i) \in \Psi_\sigma$ is well-connected *in S if for every pair of points $p \in A_i$ and $q \in B_i$ there exists a point $r \in \sigma(p)$ such that $dist_\sigma(p, r) \leq dist_\sigma(p, q)$ and a corresponding edge $(p, r) \in E$ or there exists a point $r \in \bar{\sigma}(q)$ and $dist_{\bar{\sigma}}(q, r) \leq dist_{\bar{\sigma}}(q, p)$ and a edge $(q, r) \in E$.*

As two points are always connected either directly via an edge or through a path going through one or more points it is easy to see that the edges from a well-connected collection of CSPDs forms a spanner. From lemma 2.4 in [2] we have that this is a $(1 + \varepsilon)$-spanner.

Lemma 2. *For a collection of CSPDs of a point set P and a graph $S = (P, E)$ such that the E is the edges from the CSPDs. If every pair in each Ψ_σ is well-connected then S is a $(1 + \varepsilon)$-spanner on P.*

Section 4.3.3 explains how to connect a pair in a CSPD such that it is well-connected.

4.3 Constructing a Spanner From the Theory

From the previous section it follows that we have a spanner if we can construct a well-connected collection of CSPDs from a point set P. Furthermore if the set of θ-cones satisfies that $\cos 2\theta - \sin 2\theta \geq \frac{1}{1+\epsilon}$ then by lemma 2 it is a $(1 + \varepsilon)$-spanner.

It remains to show how to construct a collection of CSPDs from P and how to make the pairs of a CSPD well-connected such that it is easy to maintain. This section will explain how this can be done.

4.3.1 Rank Based Range Trees

A collection of CSPDs can be obtained in a fairly easy manner using a modified *range tree*. A *range tree* [11] is a data structure that in $O(\log^d n + k)$ time can return the subset of points that are inside a given range. Range trees can easily be generalized to k-dimensions but this section will focus on the two-dimensional version. The algorithm uses a variant of the range tree called a *rank based rank tree* (*RBRT*) which will be explained below. Normally a tree has a dynamic size but in the following we assume that the size is fixed such that it can contain every input point.

Rank Based Range Tree

A two-dimensional RBRT consists of two types of balanced binary search trees. The first tree called the *first-level tree* store points in its leafs and a pointer to a tree of the second type. The second type of tree called *second-level tree* is like the first-level tree except that it do not store a pointer to another tree in its internal nodes but instead stores the canonical subset containing the points in the subtree rooted at the internal node (by definition the point stored in a leaf is also a canonical subset). A RBRT contains one first-level tree and a set of second-level tree such that every internal node in the first-level tree points to its own instance of a second-level tree.

Points are inserted into the trees using as key the *rank* of a point in a dimension. The $rank_i(p)$ of a point p is the position of p in a set that contains all the points and are sorted according to the i'th dimension, e.g. if the dimensions are x and y we sort the sets according to the order of the points projections onto the x-line and y-line.

A point p is first inserted into the first-level tree using $rank_1(p)$ as key finding a path to a leaf where the point is stored. When an internal node is visited the point is inserted into the second-level tree pointed to by the internal node. Inserting p into a second-level tree works as insertion into the first-level tree except that the $rank_2(p)$ is used as key and that visiting an internal node does not trigger an insertion into another level. Instead the point is added to the canonical subset of the internal node [2]. Deleting a point is done in the same way only removing from the canonical subset instead of adding to it.

This construction can be used to make an open *range query* by defining the two ranks r_1 and r_2 that limits the range. All points inside that range are located by searching from the root in the first-level tree with r_1 until a leaf is reached. Whenever we branch left in a internal node the second-level tree that the right sibling points at is searched in using r_2. When searching in a second-level tree we report the points of the canonical set of an visited internal node every time we branch left. The reported points are the set of points inside the range.

CSPD and RBRT

To be able to construct a CSPD from a RBRT we need to store some further informations. In every internal node or leaf of the second-level trees we store all the ranges that selects that internal node or leaf. This is all the ranges that contains the points of the canonical subset. By construction of the RBRT all possible ranges in a RBRT has a unique point that the range

[2] In a k-dimensional range tree the second-level would trigger an insertion into a third-level tree using the third dimensional coordinate of the point. This continues until the last level k is reached.

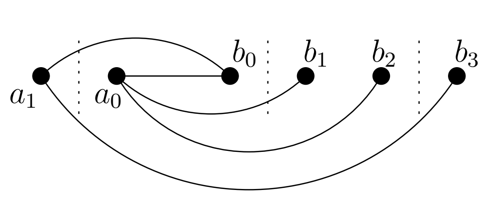


Figure 4.2: The pair (A_i, B_i) from Figure 4.1 packed into logarithmic number of groups and connected as described in Section 4.3.3.

emanates from such that there emanates a range from every point p in the RBRT. The set we store contains the points defining the ranges and is called a *range set*.

The ranges selecting an internal node or leaf can be found by doing a range query for every point $p \in P$. The range sets are constructed by adding p to the range set of visited internal nodes or leafs instead of reporting the canonical subset.

4.3.2 Constructing Cone-Separated Pair Decompositions

To construct a Ψ_σ from a RBRT we do not use the normal dimensions of the Euclidean space in the RBRT. Instead we use the lines that defines and limits σ. These lines are called the the *directions* of σ.

To get a collection of CSPDs we create a RBRT for every $\sigma \in C$ denoted $RBRT_\sigma$. In $RBRT_\sigma$ the points are sorted according to the rank of the projection onto the orthogonal of the two directions of σ. This specifies a range $\sigma(p)$ for a point p as every point that has a lower rank on one of the projection onto the orthogonal of a direction is outside the range $\sigma(p)$.

To construct Ψ_σ every point $p \in P$ are inserted into $RBRT_\sigma$ as described above. This constructs the canonical subsets and range sets of the second-level trees. For every node v in a second-level tree containing non-empty canonical subset and range set we create a pair (A_v, B_v) such that A_v is the range set of v and B_v is the canonical subset of v. From the definition of CSPD the set of such pairs from a $RBRT_\sigma$ forms Ψ_σ.

By constructing a CSPD in this way we know that each point can at most be in $O(\log^d n)$ canonical subsets and each range $\sigma(p)$ can at most be in $O(\log^d n)$ range sets. From this we can state lemma 3.

Lemma 3. *For a cone set C and a set of points P. There exist for every cone $\sigma \in C$ a CSPD Ψ_σ such that for every point $p \in P$ it is at most in $O(\log^d n)$ pairs in Ψ_σ.*

4.3.3 Connecting Points In a Pair of a CSPD

It remains to show how to connect the pairs in a CSPD such that they are well-connected.

To do this we sorts the points in every pair of Ψ_σ according to the representative edge of σ. The canonical subset is sorted such that the rank of the points are increasing with the direction of the representative edge i.e. for three points $p, q, r \in P$ such that $q, r \in \sigma(p)$ the point p or r closest to p is the one with the lowest rank. The points in the range set are also sorted according to the representative edge just in reverse order such that a low rank implies lesser distance to the points in the canonical subset.

A sorted pair (A_i, B_i) is easy to make well-connected. We only need to connect the every point $p \in A$ with a point $q \in B$ such that p and q has the same rank (if such to points exists). From the definition of well-connected pair of points this is clearly well-connected. However this way of connecting a pair in a CSPD may require linear time to maintain e.g. when a point with rank i is added to a subset all points with rank $j \mid j > i$ has to change and all the connections of these points has to be updated.

To overcome this we pack the sets of a pair (A_i, B_i) into a logarithmic number of groups. A set $S_i = S_i^0, ..., S_i^j$ is then grouped such that each group contains 2^j points and $S_i^0 = s_0$ and $S_i^j = s_{2^j-1}, ..., s_{2^{j+1}-2}$. For every group j in A_i we connected each point $p \in A_i^j$ to two points $q, r \in B_i^{j+1}$ such that no points in B_i^{j+1} are connected to more than one point in A_i^j. We do the same for every group $B_i^j \in B_i$. Besides this we also connect the two points in A_i^0 and B_i^0. From this way of connecting the groups we get lemma 4.

Lemma 4. *A pair $(A_i, B_i) \in \Psi_\sigma$ is well-connected and every point $p \in A_i$ is at most connected to three points in B_i and every point $q \in B_i$ is at most connected to three points in A_i.*

Proof. That each pair is at most connected to three other points follows from the construction. Every pair is well-connected as for any $p \in A_i^j$ and $q \in B_i^{j'}$ either p is connected to a point $r \in B_i^{j-1}$ and then $dist_\sigma(p, r) \leq dist_{sigma}(p, q)$ or q is connected to a point $r \in A_i^{j'-1}$ and then $dist_{\bar{\sigma}}(q, r) \leq dist_{\bar{\sigma}}(q, p)$ or $p \in A_i^0 \vee q \in B_i^0$ and then p or q is directly connected to the closest point in the other set. □

4.3.4 Linear Sized Spanner From a CSPD

A spanner $S = (P, E)$ constructed from the algorithm above is by lemma 2 a $(1+\epsilon)$-spanner but the size of E is $O(n \log^d \frac{n}{\epsilon^{d-1}})$ [2, page 47].

To get a pruned spanner $S^* = (P, E_{S^*})$ with a linear number of edges we only add edges from E to E_{S^*} for any cone $\sigma \in C$ and two points p, q

where $q \in \sigma(p)$ or $p \in \bar{\sigma}(q)$. If for all edges (p, q) the point q is the point in $\sigma(p)$ that are closest to p or p is the point in $\bar{\sigma}(q)$ that is closest to q.

The pruned spanner S^* is a $(1+\varepsilon)$-spanner as the CSPDs are still well-connected (we always keep the edge (p, q) of the closest point q for any range $\sigma(p)$ or $\bar{\sigma}(p)$) and it has $O(\frac{n}{\varepsilon^d})$ edges ([2] states that it has $O(\frac{n}{\varepsilon^{d-1}})$ edges) as the number connections between points depends on the number of cones and points.

We can now summarize the properties of the spanner in lemma 5.

Lemma 5. *A spanner S^* has $O(\frac{n}{\varepsilon^d})$ edges and has maximum degree $O(\frac{\log^d}{\varepsilon})$.*

Proof. As the number of edges is bounded by the number of points and cones (see above) the number of edges is $O(\frac{n}{\varepsilon^d})$ ([2] states $O(\frac{n}{\varepsilon^{d-1}})$).

By lemma 3 a point can at most participate in $O(\log^d n)$ pairs in a CSPD and by lemma 4 each point can at most be connected to three other points in a pair. So the maximum degree of S^* is $O(\frac{\log^d n}{\varepsilon})$ ([2] states $O(\log^d n)$). □

4.4 Maintaining the Spanner

To keep the spanner valid we need to maintain the set of RBRTs, the edges created from the CSPDs and the edges in the pruned spanner.

4.4.1 Maintaining Rank Based Range Trees

Recall that a RBRT is stored for each cone such that the directions defining a cone corresponds to the dimensions in the RBRT. A RBRT is valid as long as the order - with respect to each direction - are unchanged thus to maintain a RBRT we just need to maintain a one-dimensional sorted list for each directions of the corresponding cone.

Kinetic sorted Lists

A one-dimensional sorted list can be maintained by swapping two points if the order of two points changes. This requires a certificate for each pair of points that are neighbors in the list. Each certificate compares the position of the two points and expires when the unequality $pos(p_{i-1}) - pos(p_i) > 0$ (inverted if the sorting is done in reverse order) is not satisfied. This leads to $n - 1$ certificates per list.

The number of lists is bounded by the dimension of the RBRTs and the number of cones thus the total number certificates maintaining points in a list is $O(\frac{n}{\varepsilon^d})$ and the maximum number of certificates a point can be involved in is $O(\frac{1}{e^d})$.

The event generated when a certificate expires only has to swap the two points in the list and update the two other certificates that depends on the points. Handling a event thus has constant complexity.

Updating RBRT

When two points change order in a sorted list we need to update the related RBRT. This implies updating the canonical subsets and range sets that contains the points. This can easily be done by deleting the two points that have changed order from the RBRT and re-inserting them back into the RBRT with their new rank. Updating a RBRT thus takes the time of deleting and inserting two points which is $O(\log^d n)$.

4.4.2 Updating Cone-Separated Pair Decompositions

We need to update a Ψ_σ when there is a change in a pair $(A_i, B_i) \in \Psi_\sigma$ due to a rank change in the directions of σ. But as the connection between points in a set of a pair depends on the points position on the representative edge we also when need to update when the order on the representative edge changes. Thus besides maintaining the order in the directions of a cone we also need to maintain a sorted list for the order in the representative edge. Maintaining another list do not imply a change to the asymptotic bound of the number of certificates or certificates a point can be involved in.

Updating With Respect to a Direction

When two points swaps rank in a direction the points are deleted and reinserted into the corresponding RBRT. This implies that points can be inserted or removed from a pair (A_i, B_i). When this happens we have to update the connections between the points in A_i and B_i.

When a new point p enters a subset A_i of a pair we first have to determine which group A_i^j p belongs to. We can do this by keeping the points in A_i sorted according to the representative edge such that we know how many points in A_i that have a lower rank than p e.g. by using a one-dimensional RBRT and looking at the canonical subsets of the nodes that is visited doing the insertion. When the group A_i^j that p belongs to is located we can insert p into the group. If A_i^j is full i.e. already contains 2^j elements. Then the point q in the group with the highest rank has to move to the next group A_i^{j+1} and p can take over the connections of q. Then we have to insert q recursively into A_i^{j+1} and thus continuing until all points are placed into a group. If A_i^j is non-full we can insert p into it and locate the groups B_i^{j+1} and B_i^{j-1} such that we can connect p to points in these groups as explained above in Section 4.3.3.

Before the above procedure works we need to store a group structure such that each group A_i^j stores a pointer to the point with the highest rank placed in the group, a pointer to the next group A_i^{j+1}, a pointer to the previous group A_i^{j-1} and a boolean telling if the group is full. Every group also stores two lists of points that are still missing connections so that these points can be located efficiently.

Updating a pair in a CSPD requires locating the group that a point belongs to taking $(\log n)$ time, updating at most $O(\log n)$ groups and for every updated group connect at most $O(1)$ number of points.

The procedure can be used to update both the canonical subset and the range set of a pair with the difference that when updating a range set the points are sorted according to the reverse order on the representative edge.

Removing a point from a set of a pair is handled in a similar way.

Updating With Respect to the Representative Edge

When a change happens in the order in the representative edge the CSPD also has to be updated. We locate the canonical subsets and range sets that contains the points p, q that have swapped rank. If a set contains both p and q we update the one-dimensional RBRT that is based on the points rank in the representative edge and swaps the connections of p and q.

Locating the canonical subsets and range sets takes $O(\log^d n)$ time, updating one-dimensional RBRTs takes $O(\log n)$ time and swapping connections takes $O(1)$ time.

Furthermore the pruned spanner may have to be updated. This is only the case if one of the connections being updated by the swap is an edge in the pruned spanner and that one of the changed connection was selected as the closest point by a cone $\sigma(p)$ or $\bar{\sigma}(q)$. If this is the case we check to see if the edge has to be exchanged with another edge from the non-pruned spanner.

4.4.3 Kinetic Quality

Each point is only involved in a constant number of certificates for each list. As the number of lists depends on the number of cones and thus by the stretch-factor we have that the locality of the spanner is $O(\frac{1}{\varepsilon^d})$ ([2] states $O(\frac{1}{\varepsilon^{d-1}})$) making the spanner local. The total number of certificates depends on the size of the input points and is $O(\frac{n}{\varepsilon})$ thus the spanner is compact. The responsiveness of the spanner is $O(\log^{d+1} n)$ as we may have to search through all levels of a RBRT and for each visited node use $O(\log n)$ time to update the connections between the canonical subset and the range set. This makes the spanner responsive. From theorem in 2.8 [2] the spanner is also efficient as the total number of events is $O(\frac{n^2}{\varepsilon^{d-1}})$ if the trajectories are bounded-degree polynomials.

Chapter 5

Kinetic Deformable Spanner

The *deformable spanner* [14] is a $(1+\varepsilon)$-spanner working in $\mathbb{R}^d$. When a *aspect ratio* of the points (see below) is bounded by the size of the input the spanner is "good" in kinetic sence; it is responsive thus handling events in $O(\log \frac{n}{\varepsilon^d})$ time and can handle motion plan updates in $O(\log \frac{n}{\varepsilon^d})$. It has a locality of $O(\log \frac{n}{\varepsilon^d})$ certificates per point and the total number of events is $O(n^2 \log n)$. The spanner has a size of $O(\frac{n}{\varepsilon^d})$ and a maximum degree of $O(\log \frac{n}{\varepsilon^d})$.

This chapter presents the theory of the deformable spanner. The spanner is defined in section 5.1. This section also introduces and shows chosen properties of the spanner. Section 5.2.1 explains how to construct a deformable spanner and section 5.3 explains and analyse the certificates and events required to maintain the spanner.

As in the previous chapter we will focus on the two dimensional version of the spanner.

5.1 Deformable Spanner

The *deformable spanner* depends on a ratio α called the *aspect ratio*. The *aspect ratio* of a point set P is the ratio between the two points $u, v \in P$ with the maximum Euclidean distance $|uv|_{max}$ and the two points $u', v' \in P$ with the minimum Euclidean distance $|u'v'|_{min}$.

In the remaining of the chapter it is assumed that $|u'v'|_{min} = 1$ so that $|u, v|_{max} = \alpha$. This assumption can be made without any loss of generality.

5.1.1 Definition of the Deformable Spanner

A set of *discrete centers* is the maximum subset P' of a point set P such that for a constant r every $p \in P$ is inside a circle (n-ball in higher dimension) with radius r centered at one point $p' \in P'$ such that the Euclidean distance between every pair of points $p, q \in P'$ is $|pq| \geq r$.

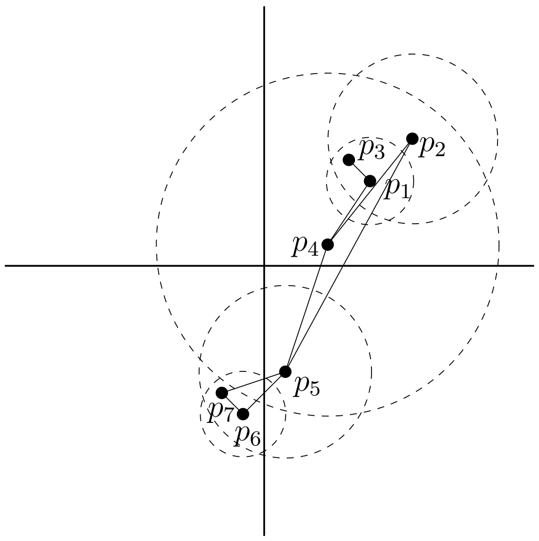


Figure 5.1: A hierarchy of discrete centers with four levels; $S_0 = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7\}$, $S_1 = \{p_1, p_2, p_4, p_5, p_6\}$, $S_2 = \{p_2, p_4, p_5\}$ and $S_3 = \{p_4\}$.

A *hierarchy of discrete centers* is a collection of of discrete centers P_i. The set of discrete centers P_0 is the original point set P and P_i is a subset of P_{i-1} such that the distance between every pair of point in P_i is greater than 2^i. The hierarchy stops when $|P_i| = 1$ such that $P_{i+1} = \emptyset$. It is worth to notice that a hierarchy of discrete centers of P is not unique. A hierarchy of discrete centers always exists and $P = P_0$ due to our assumption on the minimum distance between any two points.

The *deformable spanner* $S = (P, E)$ is constructed from the hierarchy of discrete centers. For every P_i in the hierarchy E contains an edge for any two points $p, q \in P_i$ if $|pq| \leq c \cdot 2^i$ where $c = 4 + \frac{16}{\varepsilon}$.

5.1.2 Terminology

We call the set P_i of the hierarchy of discrete centers for the i'th *level*. A point p on a given level i is denoted $p^{(i)}$.

At some level i in the hierarchy the distance between two points $p \in P_i$ and $q \in P_{i-1}$ may be lesser than 2^i such that $q \notin P_i$. The point p is then said to *cover* q. A point $p \in P_i$ may cover many points in P_{i-1} and likewise a point $q \in P_{i-1}$ may be covered by many points in P_i. When a point $p \in P_i$ are covered by one or more points it is called a *child* and one of the covering points are chosen as p's *parent* denoted $P(p)$. For a point p the maximum level i such that $p \in P_i$ and $p \notin P_{i+1}$ (or P_{i+1} does not exists) is denoted p_M.

A point $p \in P_i$ has a set of children in P_{i-1} denoted $C_{i-1}(p)$. This set

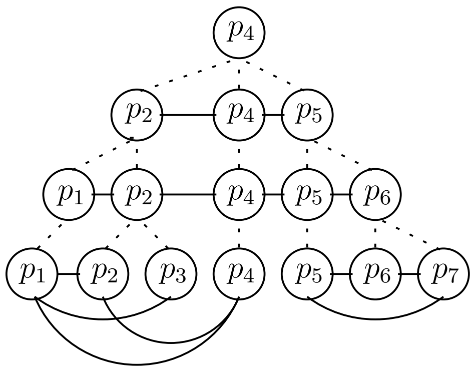


Figure 5.2: A tree like structure of the hierarchy from Figure 5.1. The dotted lines show parent-child relations and the solid lines shows the edges between points.

contains p itself and every point $q \in P_{i-1}$ such that $P(q) = p$.

We use $P^i(p)$ to denote the *ancestor* of p in level i. The *ancestor* of p in level i is

$$P^i(p) = \begin{cases} p & \text{if } p \in P_i \\ P(P^{i-1}(p)) & \text{if } i - 1 = P^{i-1}(p)_M \\ P^i(P^{i-1}(p)) & \text{else} \end{cases}$$

For a point $p \in P_i$ and every point $q \in P_i$ such that $|pq| \leq c \cdot 2^i$ and $p \neq q$ the point q is called a *neighbor* to p on level i. The set of neighbors for a point p on level i is denoted $N_i(p)$. The children of $N_i(p)$ are called *cousins* to the children of p. A point p may stop having neighbors from some level i. The level i with the highest number such that $N_i(p) = \emptyset$ and $N_{i+1}(p) \neq \emptyset$ is denoted P_m.

The level i where the size $|P_i| = 1$ and $P_{i+1} = \emptyset$ is called the *top level* of the hierarchy denoted M. The single point in the top level is called the *root* of the hierarchy. The level containing every point in P and allowing the shortest distance between the points is called the *bottom level* of the hierarchy and is denoted m

Using this terminology a hierarchy is a tree-like structure where each level of the hierarchy has parent-child relations to the level above it and the level under it. See figure 5.2 for an illustration.

5.1.3 Properties

This section will show some properties of the deformable spanner. The main purpose is to show that it is a $(1 + \varepsilon)$-spanner and to give a basis for the analyse of the kinetic properties in section 5.3.

Some Basic Properties

This section shows some basic properties of the deformable spanner.

Lemma 6. *1. $P_i \subseteq P_{i-1}$.*

2. *For every pair of points $p, q \in P_i$ the distance between the points is $|pq| \geq 2^i$.*

3. *A point q is always neighbor with its parent p on level i such that $q \in P_i$ and $q \notin P_{i+1}$.*

4. *The hierarchy of discrete centers has at most $\lceil \log \alpha \rceil$ levels.*

5. *For any point p and any ancestor $P^i(p)$ the distance $|pP^i(p)| \leq 2^{i+1}$.*

Proof. 1. This is obvious from the definition.

2. Obvious from the definition.

3. We know that $|pq| \leq 2^i$. Neighbors on level $i-1$ are at most $c \cdot 2^{i-1}$ distance away from each other. $2^i = 2 \cdot 2^{i-1} \leq c \cdot 2^{i-1}$ as $c \geq 2$.

4. At level α the radius is at least equal to the aspect ratio of the input such that the points in $P_{\alpha-1}$ cover each other in level α. From this we know that $|P_\alpha| = 1$ and it is the top level of the hierarchy.

5. An ancestor $P^i(p)$ is by definition at most distance $\sum_0^i 2^i \leq 2^{i+1}$ away from each other.

$\square$

From section 3.3 in [14] we can state the following properties of the spanner.

Lemma 7. *1. Each point in P_i covers at most 5^d points in P_{i-1}.*

2. *Any point $p \in P_i$ can at most have $(1 + 2 \cdot c)^d - 1$ neighbors in P_i ($O(\frac{1}{\varepsilon^d})$.*

3. *The maximum degree of a point is $O(\log \frac{\alpha}{\varepsilon^d})$.*

4. *The total number of edges in S is $O(\frac{n}{\varepsilon^d})$.*

Finally we show a property that the algorithm for constructing the spanner in section 5.2 will utilize.

Lemma 8. *If $q \in N_i(p)$ then $P^i(q) \in N_{i+1}(P^i(p))$.*

Proof. By definition we have that $|P(p)P(q)| \leq |pP(p)| + |pq| + |qP(q)|$. From lemma 6 we have

$$|pP(p)| + |pq| + |qP(q)| \leq 2^{i+1} + c \cdot 2^i + 2^{i+1}$$

We can derivate that

$$2^{i+1} + c \cdot 2^i + 2^{i+1} = 2 \cdot 2^i + c \cdot 2^i + 2 \cdot 2^i = (c+4) \cdot 2^i$$

and conclude

$$(c+4) \cdot 2^i \leq c \cdot 2^{i+1}$$

□

A $(1+\varepsilon)$-Spanner

From the properties above we can summarize and show that the deformable spanner is a $(1+\varepsilon)$-spanner.

Theorem 1. *The deformable spanner $S = (P, E)$ is a $(1+\varepsilon)$-spanner if α is bounded by the size of the input. Furthermore it has $O(\frac{n}{\varepsilon^d})$ number of edges and a maximum degree of $O(\log \frac{\alpha}{\varepsilon^d})$.*

Proof. The size and maximum degree of the spanner is given from lemma 7.

It remains to show that the spanner is a $(1+\varepsilon)$-spanner. For two points $p, q \in P$ we find the lowest level i such that the the ancestors $P^i(p)$ and $P^i(q)$ are neighbors. From this way of choosing i we have that

$$|P^i(p)P^i(q)| \leq c \cdot 2^i$$

and

$$P^{i-1}(p)P^{i-1}(q) > c \cdot 2^{i-1}$$

From lemma 6 we know that

$$|pP^{i-1}(p)| < 2^i \quad \text{and} \quad |qP^{i-1}(q)| < 2^i$$

We obtain a lower bound on the distance $|pq|$ by subtracting the distance from each point to its ancestor from the distance between the ancestors.

$$|pq| \geq |P^{i-1}(p)P^{i-1}(q)| - |pP^{i-1}(p)| - |qP^{i-1}(q)|$$

Which can be rewritten as follows.

$$|P^{i-1}(p)P^{i-1}(q)| - |pP^{i-1}(p)| - |qP^{i-1}(q)| > c \cdot 2^{i-1} - 2 \cdot 2^i$$

$$> c \cdot 2^{i-1} - 4 \cdot 2^{i-1} > (c-4) \cdot 2^{i-1}$$

Now we can show that there exists a path connecting p and q via $P^i(p)$ and $P^i(q)$ with a length lower than $(1+\varepsilon)\cdot|pq|$.

We know that the length of the path from p to q via $P^i(p)$ and $P^i(q)$ is $2^{i+1}+|P^i(p)P^i(q)|+2^{i+1}$ (or $|P^i(p)P^i(q)|$ if $i=0$). From the proof of theorem 3.2 in [14] we get a upper bound

$$2^{i+1}+|P^i(p)P^i(q)|+2^{i+1}\leq 8\cdot 2^i+|pq|$$

and

$$8\cdot 2^i+|pq|\leq \frac{1+16}{c-4}|pq|=(1+\varepsilon)|pq|$$

As such a path can be found for every pair of points in P the deformable spanner is a $(1+\varepsilon)$-spanner. □

5.2 Construction of the Deformable Spanner

The construction of the deformable spanner works in an online way inserting every point one by one. As a consequence of this we don not know the aspect ratio in advance so the algorithm has to be able to dynamic extend the hierarchy of discrete centers. If two points comes within distance $|pq|\leq 2^m$ the hierarchy has to be extended with another level $m-1$. Likewise if two points in the top level do not cover each other the hierarchy has to be extended with a level $M+1$.

5.2.1 Constructing the Spanner

The insertion of a point is is done in two phases; the first iteration that locates the neighbors of p and the second iteration that locates a parent to p. The algorithm assumes that the hierarchy has a root and that there exist at least one level so that m and M are set. We get around this by evade the algorithm when the first point is inserted. Instead this point is chosen as root and just inserted into the single initial level of the hierarchy.

The First Iteration

The first iteration works from the top level to the bottom level locating neighbors. We know from lemma 8 that when a point p has an empty neighbor set $N_i(p)$, then all sets $N_j(p)$ where $j<i$ are also empty. So when the first empty neighbor set is found the iteration terminates. The first iteration is sketched as pseudo code in algorithm 1.

If the iteration reaches the bottom level P_m the hierarchy is extended with a new level P_{m-1}. Any neighbor $N_m(p)$ are added to P_{m-1} and we look in P_{m-1} to see if p still has neighbors. If p has neighbors the hierarchy is extended with another level P_{m-2} and so in continue until p at some

Algorithm 1 First Iteration when inserting p

```
Insert $p$ into all levels $P_m$ to $P_M$.
$lvl \leftarrow M$ {Begin from top level of the hierarchy}
$p_M \leftarrow M$
repeat
  if in top level of hierarchy then
    if dist($p$, $root$) $\leq c \cdot 2^{lvl}$ then
      add $p$ and $root$ as neighbors in $lvl$.
    end if
  else
    for all cousins in $lvl$ do
      if dist($p$, $cousin$) $\leq c \cdot 2^{lvl}$ then
        add $p$ and $cousin$ as neighbors in $lvl$.
      end if
    end for
  end if
  $p_m \leftarrow lvl$
  $lvl \leftarrow (lvl - 1)$
until $N_{lvl}(p) = \emptyset$ or bottom level is reached.
```

new level P_i has no neighbors. This level is the new bottom level and m is updated accordingly.

The Second Iteration

The second iteration works from the bottom level to the top level locating a parent to p. It finds the first set of discrete centers P_i where a neighbor q covers p. By definition p and q cannot both be in P_i and every P_j such that $j > i$. So p is added as child to q on level $i - 1$ and the algorithm removes p from level i and every level above. The second iteration is sketched as pseudo code in algorithm 2.

If the top level of the hierarchy is reached before any parent is found the hierarchy is extended with another level. This is done by adding a new level P_{M+1} to the hierarchy. We know that there is only one root point in P_M. This point and p are added to the new level. If the root point covers p in P_{M+1}, p is deleted from the level and M is updated such that M corresponds to the new top level. If the points are not covering each other in P_{M+1} the hierarchy is updated with another level P_{M+2} and so it continues until the root covers p and M is updated according to the new top level.

Algorithm 2 Second Iteration when inserting p

```
$lvl \leftarrow m$ {Begin from bottom of the hierarchy}
$cleaning \leftarrow$ **false**
**repeat**
  **if not** $cleaning$ **then**
    **for all** neighbors to $p$ on $lvl$ **do**
      **if** dist($p$, $neighbor$) $\leq 2^{lvl}$ **then**
        $p_{parent} \leftarrow neighbor$
        $p_M \leftarrow (lvl - 1)$
        remove $p$ from $lvl$.
        $cleaning \leftarrow$ **true**
      **end if**
    **end for**
  **else**
    remove $p$ from $lvl$
  **end if**
  $lvl \leftarrow (lvl + 1)$
**until** top level $M$ of the hierarchy is reached.
```

5.3 Maintaining the Spanner

To maintain the spanner we need to keep the hierarchy of discrete centers valid.

To do this we need to remove points from a level if they are not the fixed distance away from each other and on the other hand add points to levels if they are no longer covered by points in the lower levels. For this we have to kinds of certificates; a *parent-child certificate* certifying that that a parent covers a child and a *separation certificate* that certifies that points is far enough away from each others.

Besides keeping the hierarchy valid we also need to maintain the edges of the spanner. We need to certify that neighbors in a level is within the proper distance of each other. We also need to ensure that if two points comes within a certain distance they are added as neighbors. For this we have two kinds of certificates; a *potential edge certificate* that add points as neighbors when needed and a *edge certificate* that removes points as neighbors if they are to far away from each others.

It may seem like the combination of *edge certificates* and *separation certificates* implies that points are involved in a linear number of certificates as we can have levels that contains all the points from the input set. This however is not the case as explained below.

The following sections explains the certificates and the corresponding events in more details. The sections analyses the kinetic quality of each certificate and event and the final section summarizes the overall quality of the kinetic structure.

5.3.1 Potential Edge Certificate

A *potential edge certificate* is a certificate that ensures that if two points $p, q \in P_i$ comes within distance $c \cdot 2^i$ of each other they are added as neighbors.

From lemma 8 we know that two points p, q in level i can only be neighbors if their ancestors are neighbors in level $i + 1$. This observation implies that we only create potential edge certificates for every point p and q in a level if they are cousins and thus avoiding that points are involved in a linear number of certificates.

By lemma 7 a point can have at most $O(\frac{1}{\varepsilon^d})$ neighbors in a level and can be parent to 5^d children in each P_i. So the total number of potential edge certificates a point can be involved in in a level is $O(\frac{1}{\varepsilon^d})$.

When a potential edge certificate expires an *add edge event* are triggered. This event adds an edge between the certified points p, q on the certified level i . When two points becomes neighbors the children of the points gets new cousins. For every pair of children of p and q on level $i - 1$ we thus create a new potential edge certificate.

Handling an add edge event can be done in constant time as adding the edge takes constant time and each point can at most have a constant number of children in a level.

5.3.2 Edge Certificate

A *edge certificate* is a certificate that maintains that two neighbors p, q in level i stays within distance $|pq| \leq 2^i$.

We have an edge certificate for every two points that are neighbors in a level. By lemma 7 we have at most $O(\frac{n}{\varepsilon^d})$ edges that are needed to be certified and each point can at most be involved in $O(\frac{1}{\varepsilon^d})$ edge certificates in a level.

When an edge certificate fails a *delete edge event* is triggered. This event removes the edge from the certified level i. When two points are no longer neighbors in level i the children of p and q in level $i-1$ may lose cousins. For every pair of children of p and q we remove the potential edge certificates that certifies the distance between them.

A delete edge event can be handled in constant time. The analyse is similar to handling an add edge event as described above.

5.3.3 Parent-child certificate

A parent-child certificate is a certificate that maintains that for any point $q \in P_i$ its parent $p \in P_{i+1}$ are within distance $|pq| \leq 2^{i+1}$.

We have a parent-child certificate for every parent-child relation in the hierarchy. As every point (except the root) has exactly one parent the total number of parent-child certificates is $O(n)$ and each point is involved in $O(1)$ parent-child certificates in a level as a point can only have a constant number of children (by lemma 7).

When a parent-child certificate expires a *promote node event* are triggered. This event locates a new parent to the child in the level i of the parent. If a new parent cannot be found the child is added to P_i and a new parent is located on level $i+1$. So it continues increasing i until a parent is found. If the top level is reached we extend the hierarchy with another level as explained in section 5.2.1.

Adding a point p to new levels implies locating neighbors and creating edge and potential edge certificates. First we insert p into all levels from the old $p_M + 1$ to the new p_M. Then we locate neighbors as in the first iteration of inserting a point. When the point is inserted into all relevant levels and all new neighbors are found we iterate through the levels creating new edge and potential edge certificates.

As described above adding two points as neighbors takes $O(1)$ time. By lemma 7 there can at most be $O(\frac{1}{\varepsilon^d})$ neighbors in a level each having a constant number of children. The hierarchy has at most $O(\log \alpha)$ levels, thus handling a promote node event takes $O(\log \frac{\alpha}{\varepsilon^d})$ time.

5.3.4 Separation certificate

A *separation certificate* is a certificate that maintains that two neighbors p, q in level i stays at least distance $|pq| > 2^i$ away from each other.

We have a separation certificate for every pair of points that are neighbors in a level, so the locality analyse is the same as for the edge certificate.

When a separation certificate expires we need to remove a point from a level and a *demote node event* event is triggered. This event removes one of the two neighbors and makes it a child of the other point.

When p is added as a child of q in level $i-1$ we may have to add p to that level. This is done as described above except that we do not iterate over multiple level but only work on level $i-1$. Making p a child of q implies that p gets new cousins on level $i-1$. For each new cousin a potential edge certificate is created.

To remove a point p from level $i-1$ we need to remove p as neighbor for all $q \in N_{i-1}(p)$ and remove the edge certificate and separation certificate certifying p, q on that level. Furthermore we need to remove all potential edge certificate certifying p, q for every cousin q of p in level $i-1$.

If p had any children in level $i-1$ they are now without a parent. For each child we locate a new parent as explained when handling a promote node event.

It takes constant time to remove a point from a level. The children of the point are handled as when handling a promote node event so it takes $O(\log \frac{\alpha}{\varepsilon^d})$ time to handle a child. As any point has a constant number of children it takes $O(\log \frac{\alpha}{\varepsilon^d})$ time to handle a demote node event.

5.3.5 Kinetic Quality

From the analyse of each type of certificate we see that each point at most participates in $O(\frac{1}{\varepsilon^d})$ certificates in any level and thus the the spanner is local. It also shows that the total number of each type of certificate is at most $O(\frac{n}{\varepsilon^d})$ making the spanner compact.

The analyse of the handling time for each type of event shows that if α is bounded by the input size the worst case time of handling an event is $O(\log \frac{n}{\varepsilon^d})$ making the spanner responsive.

By lemma 5.1 in [14] we know that the lower bound of topological updates is $\Omega(\frac{n^2}{(1+\varepsilon)^d})$. As the total number of event that the spanner is bounded by is $O(n^2 \log \alpha)$ when trajectories are bounded-degree polynomials (see page 195 in [14]) the deformable spanner is efficient.

Chapter 6

Implementation

This chapter concerns the implementation of the kinetic spanners. The implementations are based on a framework called CGAL which is introduced in Section 6.1. Section 6.2 explains how the implementations are tested. Section 6.3 and 6.4 describes and summarizes the status of the implementations of the two kinetic spanners.

6.1 CGAL

The Computational Geometry Algorithms Library (CGAL) is a open source library providing a wide variety of computational geometric algorithms and data structures. It also offers a framework for developing that includes geometric objects and predicates with exact precision.

Since version 3.2 (`http://www.cgal.org/releases.html`) of CGAL the framework has contained a kinetic package. This package provides kinetic algorithms and data structures but also a framework for developing kinetic data structures. The kinetic package was developed by Daniel Russel as a part of his Ph.D. Thesis [22] and details about the package can be found in the thesis.

The implementation of the kinetic spanners is based on the kinetic package of the CGAL. More precisely version 3.5.1 of the CGAL framework.

6.1.1 The Kinetic Package of CGAL

To understand the implementation a little knowledge of the kinetic package in CGAL is required. In the following the basic components and concepts are explained.

- **Active Object Table** (AOT) — This table is a global table containing all the primitives of a kinetic data structure i.e. the input points. Each point that is inserted into the active object table is mapped with a key such that a point can be extracted from the table. Changes such as insertion, deletion or motion plan updates in the input set are handled through the active object table such that when changes happens the kinetic data structure will receive a notification from the active object table.

- **Simulator** — The simulator handles the aspect of time in a simulation and triggers events from certificates that expires. To do this the simulator maintains an event queue sorted after the expiration time of the certificates. Thus as time progress events are triggered from this queue. A notable win by having a global event queue is that it is possible to find intervals in the simulation where no certificates expires. Such intervals can be used by the kinetic structure to safely verify its internal structure. This is called *auditing*.

- **Kinetic Kernel** — The kinetic kernel handles kinetic primitives and certificates. It provides a certificate generator that can generate certificates from predicates and trajectories of the primitives.

- **Instantaneous Kernel** — The instantaneous kernel is the mapping between the kinetic model and a static model. An instance of the instantaneous kernel is fixed at some time in the simulation and provides methods for evaluate kinetic primitives in this instant of time and thus makes kinetic objects usable in a static context.

- **Function Kernel** — A function kernel handles computations in the kinetic model e.g. it handle computations based on polynomials. This includes finding and comparing roots of the polynomials. It also provides computation for the instantaneous kernel so that primitives can be evaluated at a specific point in time.

- **Traits** — A traits is a common concept in CGAL not particular related to the kinetic package. A traits specifies data types, kernel types, table types etc. and stores global objects. It has the function of being a global mediator such that all part of a program can access central components and specifications.

6.1.2 Using CGAL To Run a Simulation

When using a kinetic data structure in CGAL one first sets up the base environment through a Traits-class. The kinetic data structure can then be instantiated using this traits and it will be registered by the components of the traits thus receiving notifications from the framework.

The point set can then be created e.g. from a file. The active object table will store the points and notify the kinetic data structure about the new points.

Then the simulation can begin. As the simulation progress events will be handled one by one and when the time of the simulation is in an interval suitable for auditing the kinetic data structure is notified. The simulation ends when no more certificates will expire.

6.2 Testing

The implementation is tested extensively using the audit functionality. Each time the kinetic spanner is allowed to audit it is tested for violation of definitions and properties but also combinatorial errors. E.g. the deformable spanner is tested to see if lemma 8 in Section 5.1.3 is satisfied but it is also checked that two points on a level only has one edge between them on that level.

Using the audit functionality in combination with assertions from the C++ language gives dynamic error detection such that an error is reported as soon as it occur (if it is tested for) thus making debugging a bit less cumbersome.

Besides running the spanner on auto generated point sets the implementations has also been tested on a few handmade sets. This has mainly been used to follow an implementation when it is running from start to the end thus verifying the internal structures by hand.

6.3 Kinetic Spanner in $\mathbb{R}^d$

The implementation of the kinetic spanner in $\mathbb{R}^d$ is based on the theory of Chapter 4. However as explained in Section 6.3.2 below certain things are done differently.

6.3.1 Overview of the Code

The implementation of the kinetic spanner in $\mathbb{R}^d$ is structured into the following components.

- **SortedList** — The SortedList is a wrapper to the kinetic sorted list from CGAL. It wraps the kinetic sorted list so that only selected one-dimensional points are inserted into the list. Namely the points that are projections onto the direction the list represents. This has to be done as whenever a new one-dimensional point is created it is inserted into the active object table and then all kinetic data structures are notified about the point. But we only want to add the relevant points to the kinetic sorted list so the wrapper is a filter sorting all non-relevant points away.

Besides handling insertion into the kinetic sorted list the wrapper also notifies the cone it is related to whenever two points swaps rank in the wrapped kinetic sorted list.

- **ConePoint** — A ConePoint is a wrapper to a point. Each point has a ConePoint for every cone the point is inserted into such that there exists a number of ConePoints for each point.

  A ConePoint stores the key to the original two-dimensional point, the key to the one-dimensional point projected onto the representative edge and the keys for each of the one-dimensional points projected onto the orthogonal of a cones directions.

  Besides keys a ConePoint stores the ranks of the point in the two directions and the representative edge.

  A ConePoint also has a list of *edge_ref_t* pointers that stores information about connections in the well-connected pairs of the CSPD. How this is implemented is explained below.

- **RBRT** — The RBRT structure is the implementation of the rank based range tree. The implementation is based on vectors as explained below. Each node in a RBRT stores a canonical subset and a range set.

- **Cone** — The Cone structure represents a cone.

  Besides containing a RBRT a Cone contains three SortedLists. One that sorts the points by the order of the cones representative edge and two that sorts the points by the order of the projection onto the orthogonal of the two directions that defines the cone.

  Each cone stores a pointer to the next and previous cone in the circle of cones (see Section 4.1 for a example of how to construct the cone set in $\mathbb{R}^d$). By this a cone only stores one direction which is used as its representative edge and the first direction. The second direction is the representative direction of the next cone. This ensures that the set of cones covers the whole space.

  When events are triggered in one of the SortedLists stored in the cone the cone is given control and handles the update its RBRT.

6.3.2 Notes About the Implementation

The implementation of the rank based rank tree is based on a vector implementation of binary trees as described in [15]. Each RBRT is a skeleton capable of storing 2^i points such that $2^i \geq n > 2^{i-1}$. As the the input set in this thesis never changes doing a simulation the RBRT is always "full" (as close as it gets) and no space is wasted by using a vector implementation.

To keep things simple the edge set has also been implemented as vectors. An edge is represented on a point with an *edge_ref_t* that stores all information relevant for locating the exact internal node and set in a pair (A_i, B_i) from where this edge occurs. From this information it is easy to located the points in the opposite set that the current point is connected to. Namely the points with index $2 \cdot i + 1$ and $2 \cdot i + 2$ (and index 0 if the point is the first in the set) if they exists. This however requires linear processing time as it assumes that points are always completely ordered by the order on the representative edge and does thus not use logarithmic grouping.

The status of the implementation is that both the initialization and the kinetic structure is working. However as the size complexity of the spanner has been ignored from the beginning the spanner consumes asymptotic polynomial memory with a high constant factor.

Due to problems implementing the deformable spanner (see Section 6.4.2) there were no time to implement the kinetic spanner fully as described in [2]. So the development stopped when the implementation had the same spanner properties and partially kinetic properties (compactness, locality and efficiency) as the theoretical kinetic spanner.

The implementation thus lacks RBRTs maintaining the canonical subset and range set of a node and completely misses the edge set (this is computed dynamically as explained above). As a consequence of this the implemented kinetic spanner is not responsive.

6.4 Deformable Spanner

The implementation of the deformable spanner is based on the theory of Chapter 5. But as explained below in Section 6.4.2 certain things are done in a different way.

6.4.1 Overview of the Code

The implementation of the deformable spanner is structured into the following components.

- **PointNode** — A PointNode wraps every input point. It contains a key to extract the point from the active object table and two integers m and M storing the lowest and highest level in the hierarchy of discrete centers that the point is in.
- **Edge, PotentialEdge and ParentEdge** — Represents the four types of relations between points and stores the certificate for a relation. As separation certificates do not exists unless an edge exists between two points the Edge class contains both a separation certificate and an edge certificate. It is worth to note that Parent-edge relation are stored in the level of the child (see below).

- **DiscreteCentersHierarchy** — A DiscreteCentersHierarchy stores the levels of the hierarchy of discrete centers. For each level a list of edges, of potential edges and of parent-edges are stored. In contrast to the theory this implementation does not have a dynamic lower level m but always has 0 as its lowest level (see below for more information about this).

- **Spanner** — This is the kinetic data structure receiving notifications from the CGAL framework. It handles insertions of points and events as explained in Section 5.2 and 5.3. It has a first in first out queue of expired certificate events so that that events are first processes when the kinetic data structure are ready to handle them. The queue ensures that events are first processed (in the correct order) when the time of the simulation can be represented as a rational number — a requirement by the calculation done in the distance predicate.

- **AddEdgeEvent, DeleteEdgeEvent, PromoteNodeEvent and DemoteNodeEvent** — Implementations of the event types explained in section 5.3. Each type of event calls a relevant method on the Spanner class that handles the update of the spanner.

6.4.2 Notes About the Implementation

The implementation only operates down to S_0. Such a limit on the lowest level in the hierarchy has to be set to avoid the hierarchy to extend downward infinitely. E.g. when two points approaches each other as the trajectories intersects the distance between the points continuously gets lesser and lesser. The consequences of this is that the upper bound on the maximum degree and number of edges does not hold when points are under the distance $|uv| \leq 1$ of each other.

There is an error in the implementation of the kinetic structure and the exact cause of the error has not been found. The error is related to handling a DemoteNodeEvent which either (presumably) do not remove all the relevant certificates or creates wrong certificates thus planting an error that will be triggered at a later time.

In an attempt to locate the error by doing things simpler the Discrete-CentersHierachy class and PointNode class were re-coded moving the implementation away from the theory. Instead of storing neighbor, child and parent information on each point it has been moved to the relevant level in the hierarchy. This have the consequence that each level stores at worst a linear size of information in a list thus adding a linear factor to the response time. Hence the responsiveness is changed from $O(\log \frac{n}{\varepsilon^2})$ to $O(n \log \frac{n}{\varepsilon^2})$.

The status of the implementation is that the initialization works but the error in the kinetic part makes the spanner incapable of completing a simulation.

Part III

Experiments

Chapter 7

Experimental Results on Spanner Properties

This chapter presents the experimental results on the spanner properties. Section 7.1 introduces the different distribution used for the data sets of the experiments. Section 7.2 gives a brief overview of the theoretical bounds on the properties of the spanners mentioned in the previous chapters. Section 7.3 contains a few notes relevant for the experiments and finally in Section 7.4 the results of the experiments are presented and analyzed.

7.1 Data Sets

The data for the experiments has been chosen similar to that of [13]. The input size of each data set has been chosen so that the number of points increases steadily. For every input size there exists 10 versions of the data set e.g. there are ten instances of 100 uniform distributed points. The measurements are be based on the average of the output from all ten sets thus minimizing the impact of an atypical data set. All data sets contains points in two dimensions and are defined as first degree polynomials with coefficients in the range [0;1000] for both axes.

The points of the data sets are distributed in three different ways:

- Uniformly distributed points.
- Uniformly distributed points inside uniformly distributed unit squares such that a unit has size 10 and each square contains 10 points.
- Multivariate normal distributed points in two dimensions with a mean on 500 for both dimensions and a covariance matrix:

$$cov = \begin{bmatrix} 200 & 0 \\ 200 & 0 \end{bmatrix}$$

Algorithm	edges	max. degree	weight	diameter
Greedy Algorithm [4]	$O(n)$	$O(1)$	$O(\text{wt}(\text{MST}))$	$\Theta(n)$
Θ-Graph [21]	$O(\frac{n}{\theta})$	$\Theta(n)$	$\Theta(n{\cdot}\text{wt}(\text{MST}))$	$\Theta(n)$
Ordered Θ-Graph [7]	$\Theta(\frac{n}{\theta})$	$O(\frac{1}{\theta} \cdot \log n)$	$O(n{\cdot}\text{wt}(\text{MST}))$	$\Theta(n)$
WSPD-Graph [8]	$O((\frac{t}{t-1})^2 \cdot n)$	$\Theta(n)$	$O(\log n{\cdot}\text{wt}(\text{MST}))$	$\Theta(n)$
Delaunay triangulation	$O(n)$	$O(n)$	-	-
Kinetic Spanner in R^d [2]	$O(\frac{n}{\varepsilon^d})$	$O(\frac{log^d n}{\varepsilon})$	-	$\Theta(n)$
Deformable Spanner [14]	$O(\frac{n}{\varepsilon^d})$	$O(\log \frac{\alpha}{\varepsilon^d})$	-	$\Theta(n)$

Table 7.1: Properties of spanners (the upper bounds for kinetic spanner in $\mathbb{R}^d$ are those from lemma 5 in Section 4.3.4).

7.2 Theoretical Bounds on Properties

Before continuing with the results of the experiments we will see that the asymptotic bound of the diameter of the implemented spanners is $\Theta(n)$.

For the deformable spanner this is seen by placing n points on a line where the left most is denoted 0 and the right most is denoted n. For a level in the hierarchy of discrete centers we place two points with distance such that that $i+1$ covers i and no other points are neighbors on that level. By recursively placing the points like this such that n is the root of the hierarchy we have to visit n points to come from 0 to n (0 and n included).

The diameter for the kinetic spanner in $\mathbb{R}^d$ is easily seen by placing all the points on a line such that they are all connected via the same cone. Then all points are only connected with the next point on the line (and the previous) and thus the diameter is $\Theta(n)$.

Table 7.1 summarize the properties of the spanners constructed by the different algorithms.

7.3 Notes About the Experiments

The experiments are done with stretch-factors $t = 1.12$, $t = 1.27$, $t = 1.49$, $t = 1.85$ and $t = 2.42$.

Besides the two implemented kinetic spanners the experiments also includes the results from a kinetic Delaunay triangulation having a stretch-factor $t \approx 2.42$. The kinetic Delaunay triangulation is part of the the kinetic package of the CGAL framework and is described in [22].

The results of the experiments will be compared with the other spanners and the results of Farshi and Gudmundsson in *Experimental Study of Geometric t-Spanners* [13].

One problem with the spanners are that they may contain multiple edges between two points u, v. In practice this does not make sence and being mathematical stringent a set can only contain unique elements (the spanners have undirected edges thus $(u, v) = (v, u)$). So the edge set of the kinetic spanners has been pruned so that it only contains unique edges. Doing the experiments there are two results for each spanner one based on the original edge set and one on the pruned edge set. The asymptotic bounds for the original and pruned spanner are the same so if nothing else is noted the spanner that is referred to in the remaining of this Section is the pruned spanner.

7.4 Experimental Results

In the following sections the results from the experiments are presented and analyzed with respect to the spanner properties from Section 2.2. All the results of the experiments can be found in Appendix A.

7.4.1 Size

The deformable spanner in general seems to have a good ratio between the stretch-factor and the size of the input. The difference between the number of edges with a stretch-factor on $t = 1.12$ and $t = 2.42$ is slightly under a factor 2 at 200 points on uniform and multivariate normal distributed points. It is clearly that the deformable spanner performs well on clustered input which was expected as the points will cover each other at fairly low distances and hence the number of points in the levels of the hierarchy will decrease rapidly.

The kinetic spanner in $\mathbb{R}^d$ performs in general a bit better than the deformable spanner. It has about the same ratio between the different stretch-factors as the deformable spanner. Compared with the other distributions it has a lower size on the clustered data set. This may be explained by that points are grouped together such that the probability of empty ranges are higher and hence no edges are created for those ranges.

The kinetic Delaunay triangulation outperforms both of the kinetic spanners only having approximately $3n$ number of edges while the deformable spanner has approximately $5.5n$ edges at 1000 points (with $t = 2.42$) on clustered data and the kinetic spanner in $\mathbb{R}^d$ performs even worse.

7.4.2 Degree

As with the size the result of the deformable spanner varies a lot between the clustered data and the other distributions.

On uniform and multivariate normal distributed data the maximum degree is almost equal to the size of the input when the input is small. This stops from around a input of 200 points where the ratio begins to decrease. With $t = 1.85$ the maximum degree is around 150 on a input of 200 points and with a input of 600 points the maximum degree is only around 300. There is a difference between the maximum degree of the different stretch-

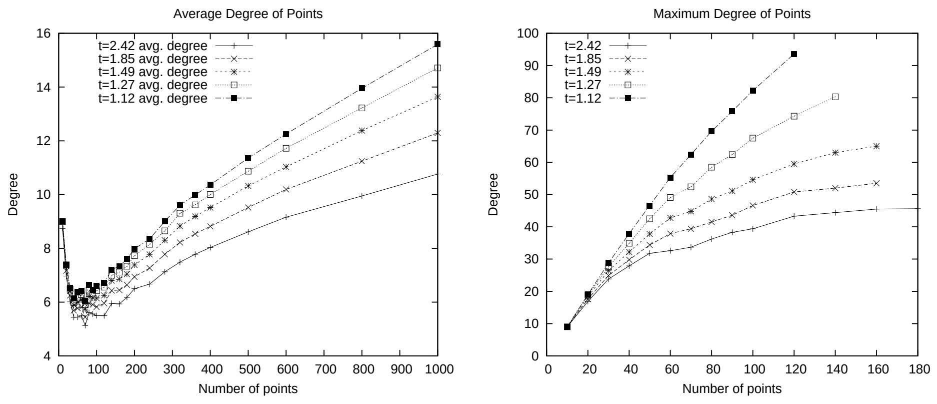


Figure 7.1: (a) Shows the average degree of the deformable spanner on clustered data. (b) Shows the maximum degree of the kinetic spanner in $\mathbb{R}^d$ on multivariate normal distributed data.

factors but the progress for the different stretch-factors are all decreasing. The average degree has the same characteristics as the maximum degree.

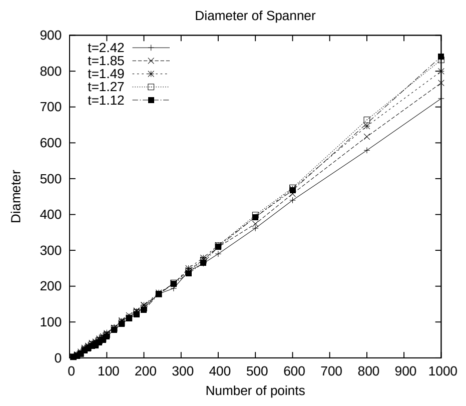
On clustered data the maximum degree is less than 100 for a input of 1000 points. The progress however seems to be similar to that of the other distributions. The average degree however is rather low and is under a degree of 16 per point on a input of 1000 points. Compared with the Delaunay triangulation which has a average degree slightly under 6 per point on the same input size it is quite good. However while the Delaunay triangulation has almost reached a constant average degree on a input of 500 points the average degree of the deformable spanner keeps increasing.

The maximum degree of the kinetic spanner in $\mathbb{R}^d$ has a constant difference between the stretch-factor and it seems like the maximum degree follows a logarithmic progress in the input size. For $t = 2.42$ it has already flatten out at around an input of 160-180 points with a maximum degree on approximately 50. This was expected as the maximum degree depends on the size of the cone set and only logarithmic on the number of input points. Compared with the Θ-graph variants from [13] the kinetic spanner in $\mathbb{R}^d$ constructs spanners with a maximum degree of only a small factor larger.

The average degree of points has the same tendencies but is rather high compared with the maximum degree. For $t = 2.42$ the average degree is almost 30 and the maximum degree is only slightly under 50 on the uniform and multivariate normal distributed data set. This comes as no surprise as the degree of each point is dominated by the number of cones.

7.4.3 Diameter

The stretch-factor of the deformable spanner hardly has any effect on the diameter on uniform and multivariate normal distributed data. This was



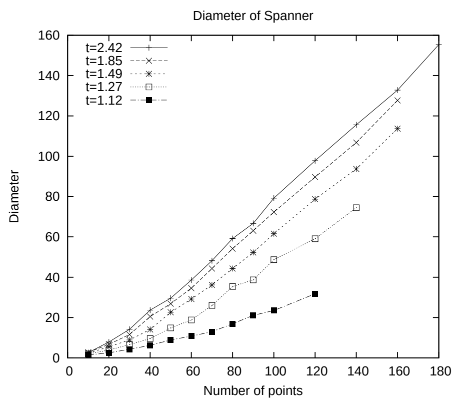


Figure 7.2: (a) Shows the diameter of the deformable spanner on clustered data. Note that the stretch-factor hardly influences the result. (b) Shows the diameter of the kinetic spanner in R^d on multivariate normal distributed data.

expected as the diameter do not depend on the stretch-factor of the spanner but depends on the aspect ratio of the input set. The diameter seems to be linear in the size of the input set and is approximately n on the uniform distributed data and $0.7n$ on the multivariate normal distributed data. This ratio between the diameter and the input set is about the same as for the kinetic Delaunay triangulation.

On clustered data the stretch-factor has an impact on the diameter but in a way such that a lower stretch-factor implies a higher diameter. This is a bit surprising and an explanation for this has not been found . On clustered data the diameter is lower than on the other data sets and it seems like it is increasing linearly from input of 500 points at about $0.9n$.

The diameter of the kinetic spanner in $\mathbb{R}^d$ has a high dependency on the stretch-factor. With the nature of the spanner in mind this makes sence as more cones implies that more points are connected directly. With a low stretch-factor the diameter is rather low, at $t = 1.12$ it seems as low as $\frac{n}{4}$ but due to the lack of testing on higher input sizes this estimates is a bit unsure. The diameter is still high compared with the results for Θ-graph variants in [13] where the best has a diameter of 5 when $t = 2$ on input of 100 points which increases to 15 on input of 2000 points. The kinetic spanner in $\mathbb{R}^d$ has a diameter ranging from 70-80 with $t = 1.85$ and $t = 2.42$ on input of 100 points. The diameter however do not seem to depend on the distribution. Something it has in common with the Θ-graphs variant from [13].

If we look at the average diameter between every pair of points in the spanners the tendencies are the same for both kinetic spanners.

7.4.4 Weight

The weight of the deformable spanner seems to have a linear progress on the higher values of t. It is reasonable to assume that this tendency is the same with the lower values of t when the size of the input set is large enough. The ratio between the different stretch-factors is about a factor 2 on clustered data at input of 1000 points between $t = 1.12$ and $t = 2.42$ but this factor seems to increase on the other distributions.

The weight of the spanner on the uniform distributed data is much higher than on the other data sets. Considering the nature of a uniform distribution and the deformable spanner this is however not surprising. The uniform distribution may imply that points are spread all over the defined area and as the deformable spanner has a tendency to create a lot of long edges between points on a level in the hierarchy the weight quickly increases when the points are not grouped.

For the kinetic spanner in $\mathbb{R}^d$ the stretch-factor has influence on the weight of the spanner and the gap between a low and high stretch-factor is rather high. With $t = 1.85$ and $t = 1.12$ the weight is approximately 200000 and 500000 on a input of 50 uniform distributed points and on input of 100 uniform distributed points the weight is approximately 400000 and 1600000.

Compared with the kinetic Delaunay triangulation the kinetic spanners performs worse. On uniform distributed points (where the Delaunay triangulation performs at worst) it has a weight of approximately 32500 on input of 100 points.

Chapter 8

Experimental Results on Kinetic Properties

This chapter presents the experimental results of the kinetic properties of the kinetic spanners. Section 8.1 describes the set-up of the experiments and Section 8.2 presents and analyses the results.

8.1 Notes About the Experiments

The experiments is based on the uniform and clustered data set explained in Section 7.1.

The size of the input is limited to 10, 30, 50, 70 and 90 as running a full simulation requires a lot more computation than just computing the initial spanner.

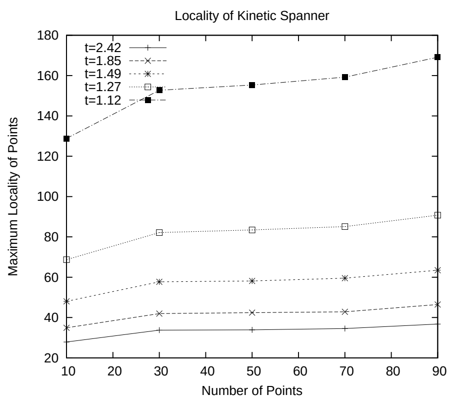
The simulation runs from [0;1] and the time is increased by 0.1 for each step in the simulation. The short interval of the simulation has been chosen since the coefficients c_i of the polynomials in the data sets are $0 < c_i \leq 1000$ meaning that almost all of the events are handled in the chosen interval.

It is worth to notice that the simulator of the kinetic package of CGAL do not contain certificates that are never going to fail. So as the simulation progress and events are handled the simulator will contain fewer and fewer certificates.

In Section 6.4 it is explained that the kinetic structure of the deformable spanner contains errors. So the kinetic experiments are only done on the kinetic spanner in $\mathbb{R}^d$ and the kinetic Delaunay triangulation.

8.2 Experimental Results

In the following sections the results from the experiments on the kinetic properties are presented. The properties experimented with are the ones described in Section 3.3 except for responsiveness. The responsiveness of



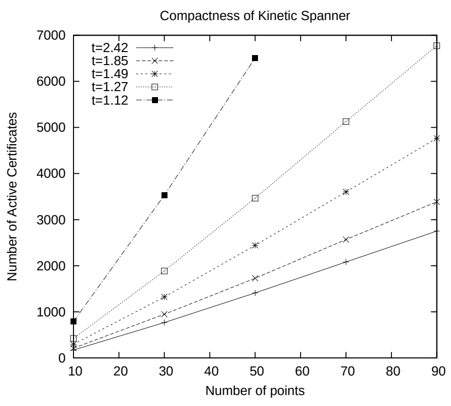


Figure 8.1: (a) Locality of the kinetic spanner in $\mathbb{R}^d$ on clustered data. (b) Compactness of the kinetic spanner in $\mathbb{R}^d$ on normal distributed data.

the implementations are not the same as in the theory so experimenting with this property would not bring any interesting results. All the results from the experiments can be found in Appendix B.

8.2.1 Locality

It is expected of the kinetic spanner in $\mathbb{R}^d$ that at most a poly-logarithmic number of certificates depends on a given point. From the experiments this can be confirmed. The stretch-factor of the spanner has an impact on the locality of the spanner. Going from approximately 30 certificates when $t = 2.42$ to approximately 150 certificates when $t = 1.12$ on a input of 30 clustered points. The difference between the locality on the clustered data and the uniform distributed data is small. The number of certificates a point is involved in for $t = 1.12$ with input of 30 uniform distributed points is approximately 170 (compared to 150 on clustered data).

Compared with the kinetic Delaunay triangulation the kinetic spanner in $\mathbb{R}^d$ has a lot more certificates. This was expected as a point in the kinetic spanner in $\mathbb{R}^d$ is involved in certificates per lists of every cone. The kinetic Delaunay implementation only has a certificate per edge so its locality follows the maximum degree of the spanner from Section 7.4.2 which is quite low.

8.2.2 Compactness

The compactness of the kinetic spanner in $\mathbb{R}^d$ is expected to be linear in the input size. This is confirmed when looking at the experiments on both uniform distributed data and clustered data. Again the stretch-factor has an impact on the total number of certificates which was expected due to the nature of the spanner.

There is a slightly increase in the total number of certificates when running the kinetic spanner with a uniform distributed data set. This can be explained by that it is less likely that two points swaps position in some sorted list in the clustered data set. This is also confirmed below when we look at the total number of handled events.

The kinetic Delaunay triangulation clearly outperforms the kinetic spanner in $\mathbb{R}^d$. Considering the kinetic Delaunay triangulation low number of total edges from Section 7.4.1 this comes as no surprise.

8.2.3 Efficiency

For the kinetic spanner in $\mathbb{R}^d$ there is a rather larger difference between the number of handled events when the input is uniform distributed points compared to clustered points. With $t = 2.42$ and a input of 90 points the number of handled events for uniform distributed points is approximately 45000 events and on clustered points it is slightly under 5000 events. As with locality and compactness the stretch-factor also influences the number of triggered events. The kinetic Delaunay triangulation only handles approximately 425 and 120 respectively. Considering the locality and compactness and the nature of the kinetic spanner in $\mathbb{R}^d$ it comes as no surprise that the Delaunay triangulation performs much better. On the clustered data sets the number of handled events of both spanners seems to be linear while on the uniform distributed sets the progress of the curve increases.

Part IV

Summary

Chapter 9

Future Work

An obvious thing to look into is the error in the kinetic structure of the deformable spanner. Finding and fixing this error will allow experimenting with the kinetic properties of the deformable spanner. This can hopefully give some interesting results that can put the kinetic properties of the kinetic spanner in $\mathbb{R}^d$ in a broader perspective.

When we have a fully working implementations the focus can be changed to optimization with respect to running time and size. To get responsive kinetic spanners the last bits from the theory needs to be implemented such that events are handled in poly-logarithmic time. Optimizing with respect to memory consumption and running time will also allow experiments with larger data size sets.

As mentioned in Section 3.4 Abam and Berg [3] has introduced another kinetic spanner running in $\mathbb{R}^2$. It is of interest to implement and experiment with this kinetic spanner so that it can be compared with the two spanners implemented in this thesis.

Chapter 10

Conclusion

This thesis have presented two algorithms that constructs and maintains geometric t-spanners. The kinetic spanner in $\mathbb{R}^d$ and the deformable spanner. The algorithms have been implemented in $\mathbb{R}^2$ with the purpose of testing them on different input sets.

The experiments have considered the properties of the spanners constructed by the algorithms. The results of these experiments showed that the spanners constructed by the kinetic algorithms in general is of worse quality than spanners constructed from non-kinetic algorithms.

If we only consider the kinetic algorithms some interesting results of the experiments are that:

- The deformable spanner performs well when the input is clustered into groups.
- The diameter of the deformable spanner do not depend on the stretch-factor t.
- The diameter of the kinetic spanner in $\mathbb{R}^d$ is rather low.
- The maximum degree the kinetic spanner in $\mathbb{R}^d$ highly depends on the stretch-factor.

Another group of experiments focused on the kinetic properties of the kinetic spanners. However due to an error in the implementation of the kinetic structure of the deformable spanner these experiments could only be done on the kinetic spanner in $\mathbb{R}^d$. To compare the spanner with another kinetic structure experiments were also done on a kinetic Delaunay triangulation

Some interesting results of the experiments on the kinetic properties are that the stretch-factor has a high influence on the number of certificates and that maintaining the kinetic spanner in $\mathbb{R}^d$ is a lot more expensive than maintaining the kinetic Delaunay triangulation.

Bibliography

[1] M. A. Abam. *New Data Structures and Algorithms for Mobile Data*. Eindhoven Univesity of Technology, 2007. Ph.D. Thesis.

[2] M. A. Abam and M. de Berg. Kinetic spanners in rd. In *SCG '09: Proceedings of the 25th annual symposium on Computational geometry*, pages 43–50, New York, NY, USA, 2009. ACM.

[3] M. A. Abam, M. de Berg, and J. Gudmundsson. A simple and efficient kinetic spanner. In *SCG '08: Proceedings of the twenty-fourth annual symposium on Computational geometry*, pages 306–310, New York, NY, USA, 2008. ACM.

[4] I. Althöfer, G. Das, D. P. Dobkin, D. Joseph, and J. Soares. On sparse spanners of weighted graphs. *Discrete Comput. Geom.*, 9(1):81–100, 1993.

[5] M. J. Atallah. Some dynamic computational geometry problems. *Computers & Mathematics with Applications*, 11(12):1171 – 1181, 1985.

[6] J. Basch. *Kinetic Data Structures*. Stanford University, 1999. Ph.D. Thesis.

[7] P. Bose, J. Gudmundsson, and P. Morin. Ordered theta graphs. *Comput. Geom. Theory Appl.*, 28(1):11–18, 2004.

[8] P. B. Callahan and S. R. Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *J. ACM*, 42(1):67–90, 1995.

[9] L. P. Chew. There are planar graphs almost as good as the complete graph. *Journal of Computer and System Sciences*, 39(2):205 – 219, 1989.

[10] K. Clarkson. Approximation algorithms for shortest path motion planning. In *STOC '87: Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 56–65, New York, NY, USA, 1987. ACM.

[11] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications.* Springer-Verlag, second edition, 2000.

[12] D. P. Dobkin, S. J. Friedman, and K. J. Supowit. Delaunay graphs are almost as good as complete graphs. In *SFCS '87: Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, pages 20–26, Washington, DC, USA, 1987. IEEE Computer Society.

[13] M. Farshi and J. Gudmundsson. Experimental study of geometric t-spanners. In *ESA 2005. LNCS*, volume 3669, pages 556–567, Berlin, 2005. Springer-Verlag.

[14] J. Gao, L. J. Guibas, and A. Nguyen. Deformable spanners and applications. In *SCG '04: Proceedings of the twentieth annual symposium on Computational geometry*, pages 190–199, New York, NY, USA, 2004. ACM.

[15] M. T. Goodrich and R. Tamassia. *Algorithm Design: Foundations, Analysis, and Internet Examples.* Wiley, 2001.

[16] L. J. Guibas. Kinetic data structures: a state of the art report. In *WAFR '98: Proceedings of the third workshop on the algorithmic foundations of robotics on Robotics : the algorithmic perspective*, pages 191–209, Natick, MA, USA, 1998. A. K. Peters, Ltd.

[17] L. J. Guibas J. Basch and J. Hershberger. Data structures for mobile data. In *SODA '97: Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, pages 747–756, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.

[18] J. M. Keil. Approximating the complete euclidean graph. In *No. 318 on SWAT 88: 1st Scandinavian workshop on algorithm theory*, pages 208–213, London, UK, 1988. Springer-Verlag.

[19] J. M. Keil and C. A. Gutwin. Classes of graphs which approximate the complete euclidean graph. *Discrete Comput. Geom.*, 7(1):13–28, 1992.

[20] G. Narasimhan and M .Smid. *Geometric Spanner Networks.* Cambridge University Press, New York, NY, USA, 2007.

[21] J. Ruppert and R. Seidel. Approximating the d-dimensional complete euclidean graph. In *In Proceedings of the 3rd Canadian Conference on Computational Geometry (CCCG'91)*, pages 207–210, 1991.

[22] D. Russel. *Kinetic Data Structures in Practice.* Stanford University, 2007. Ph.D. Thesis.

Index

Part V

Appendix

Appendix A

Experiments on Spanner Qualities

Figure A.1: Showing results for *Kinetic Spanner in* $\mathbb{R}^2$ on uniform distributed data.

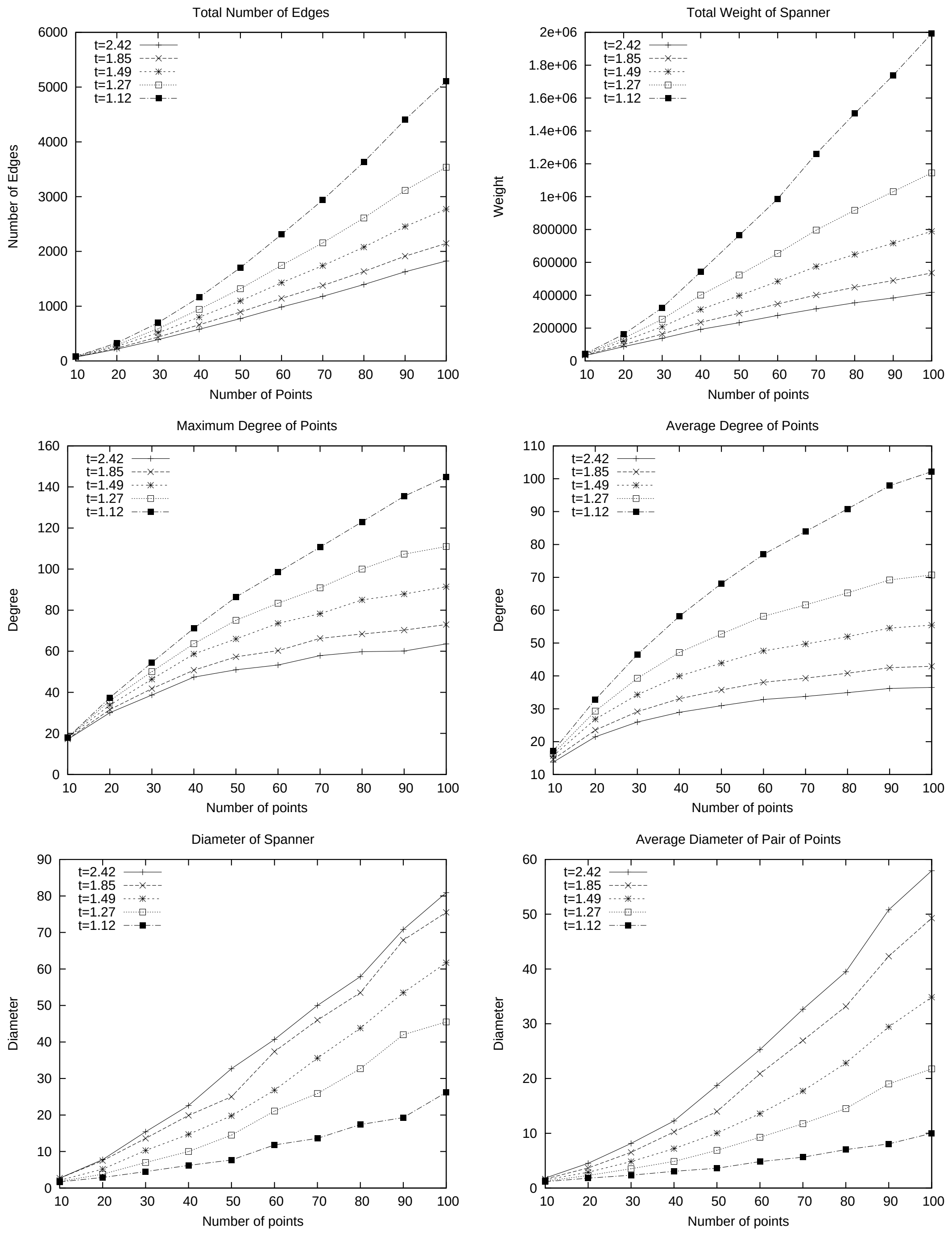


Figure A.2: Showing results for the *Kinetic Spanner in* $\mathbb{R}^2$ only containing unique edges on uniform distributed data.

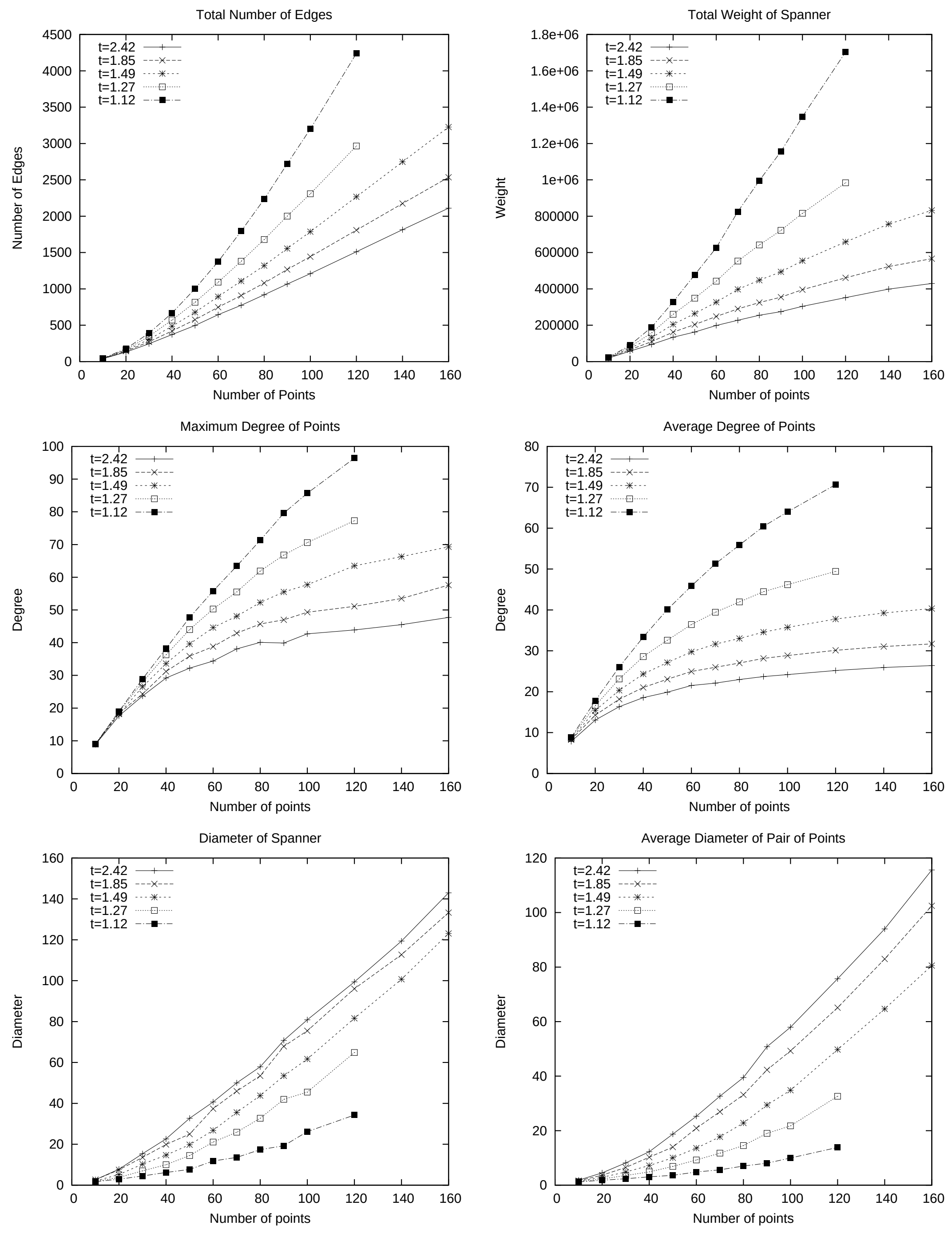


Figure A.3: Showing results for *Deformable Spanner* on uniform distributed data.

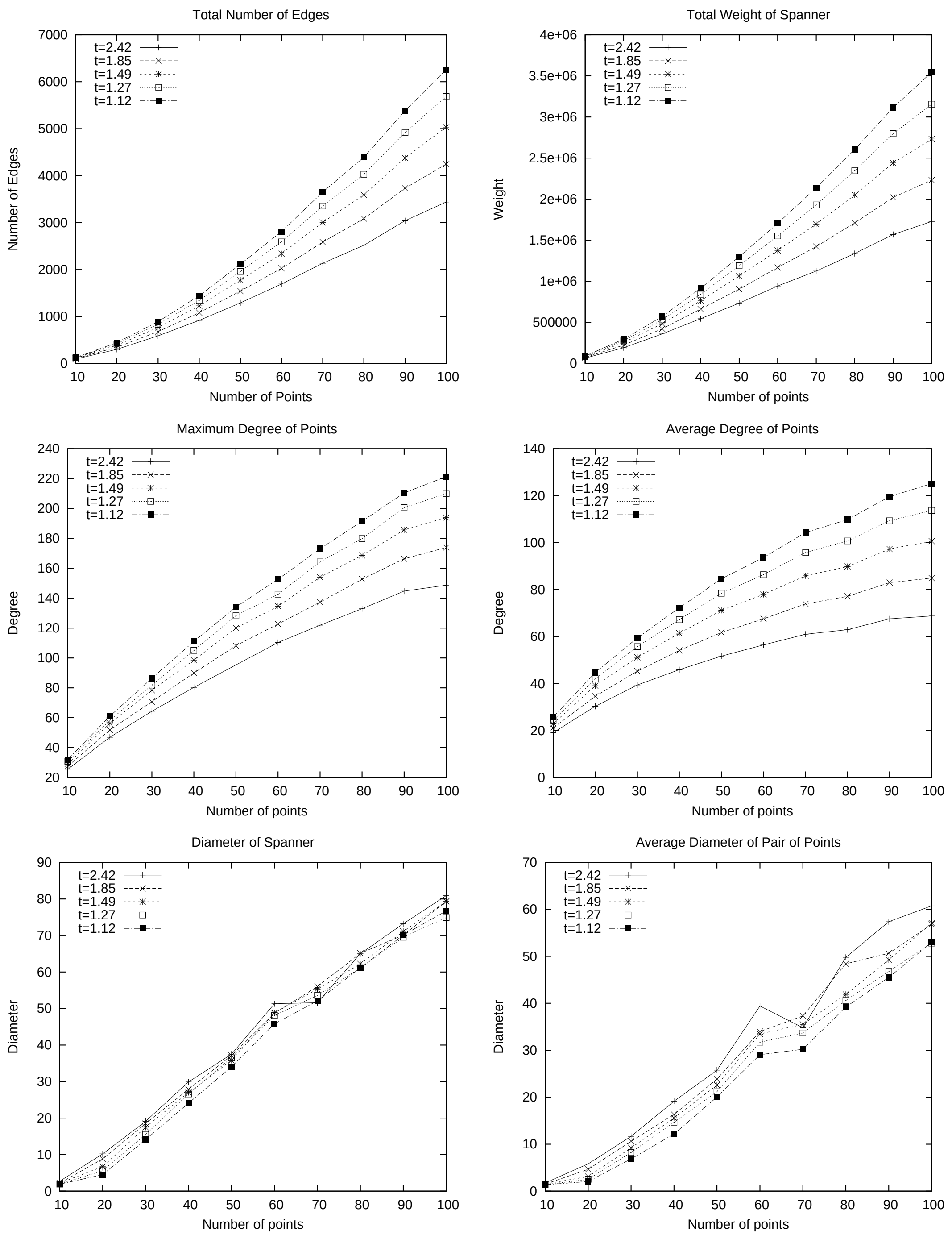


Figure A.4: Showing results for the *Deformable Spanner* only containing unique edges on uniform distributed data.

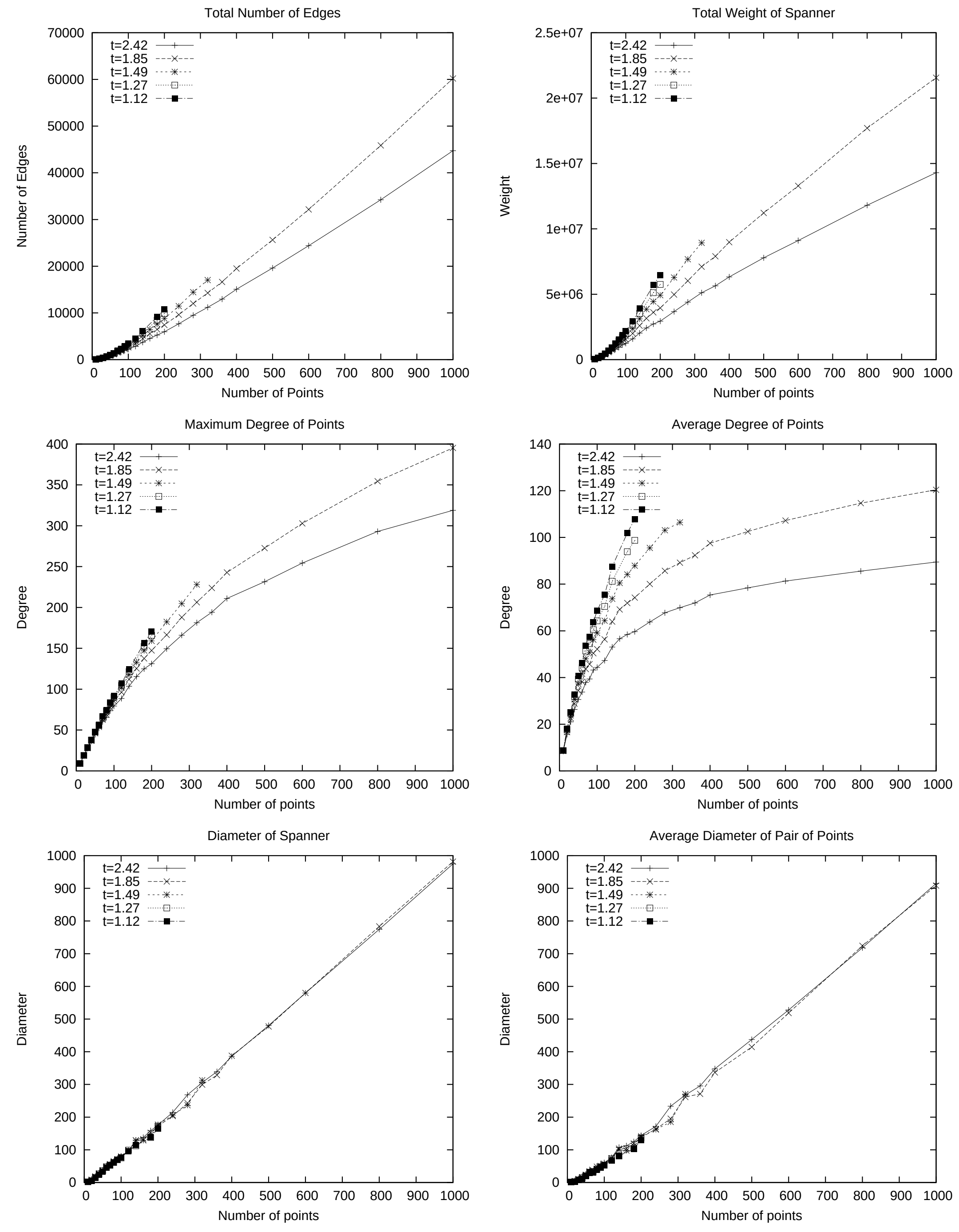


Figure A.5: Showing results for *kinetic Delaunay triangulation* on uniform distributed data.

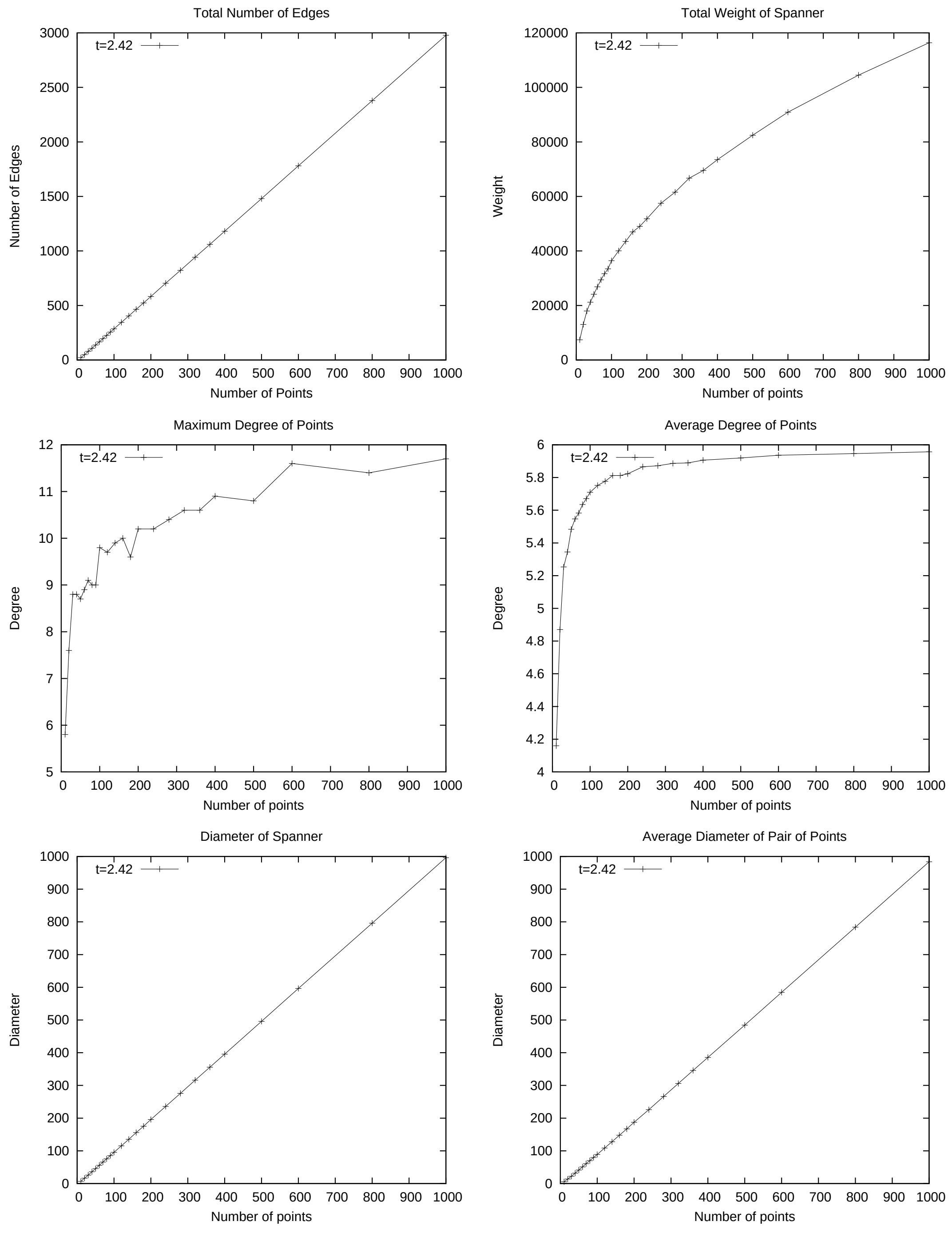


Figure A.6: Showing results for *Kinetic Spanner in* $\mathbb{R}^2$ on multivariate normal distributed data.

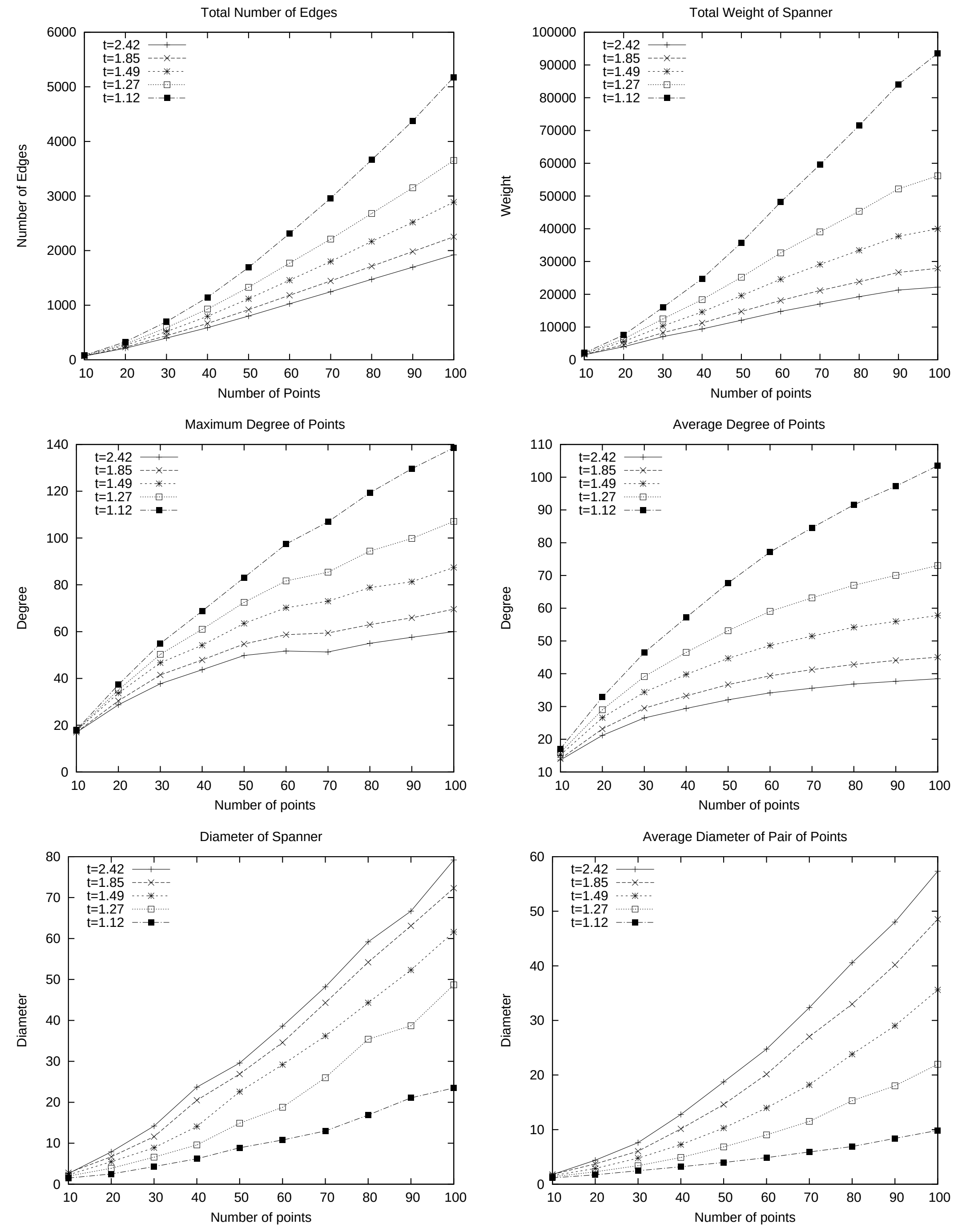


Figure A.7: Showing results for the *Kinetic Spanner in* $\mathbb{R}^2$ only containing unique edges on multivariate normal distributed data.

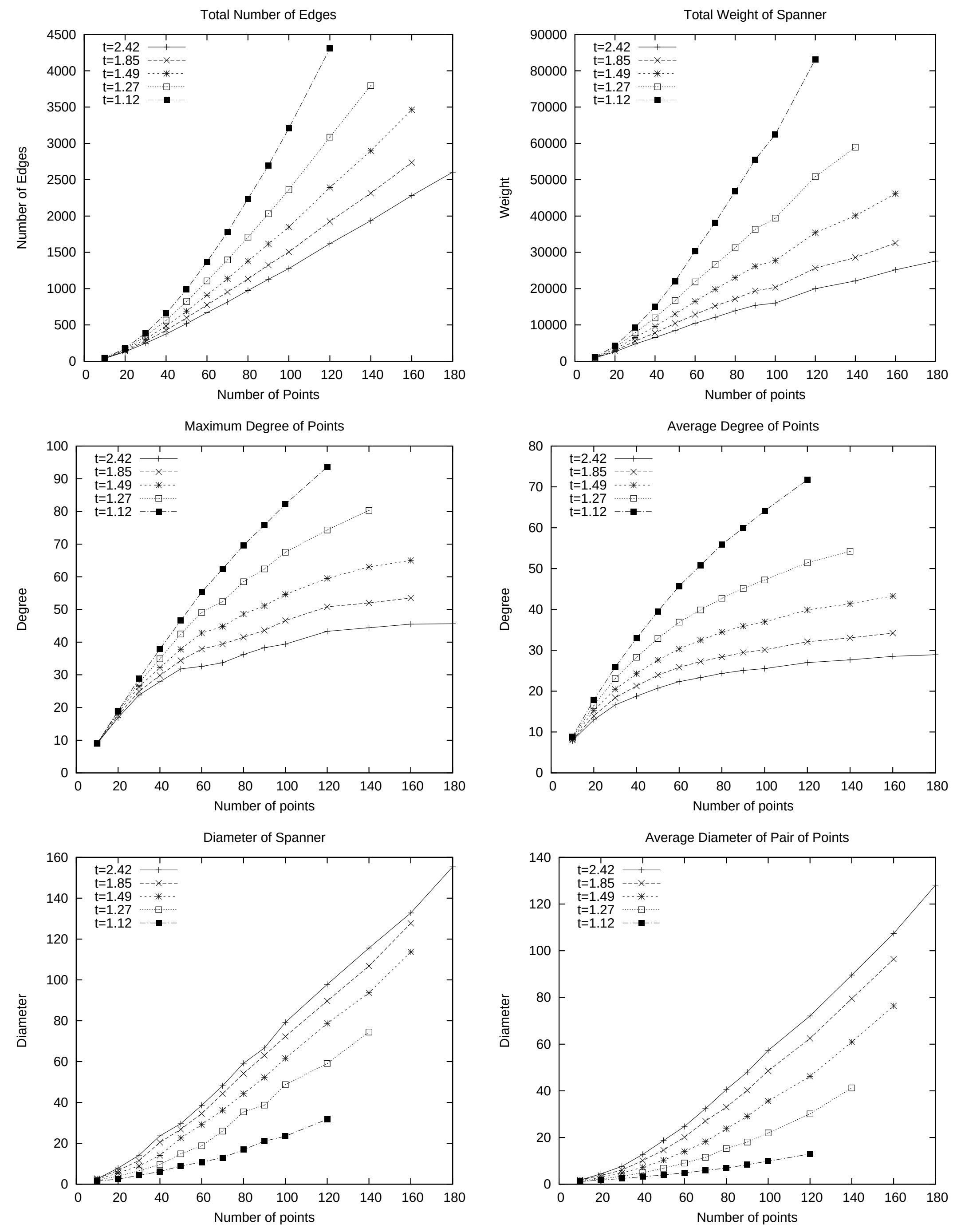


Figure A.8: Showing results for *Deformable Spanner* on multivariate normal distributed data.

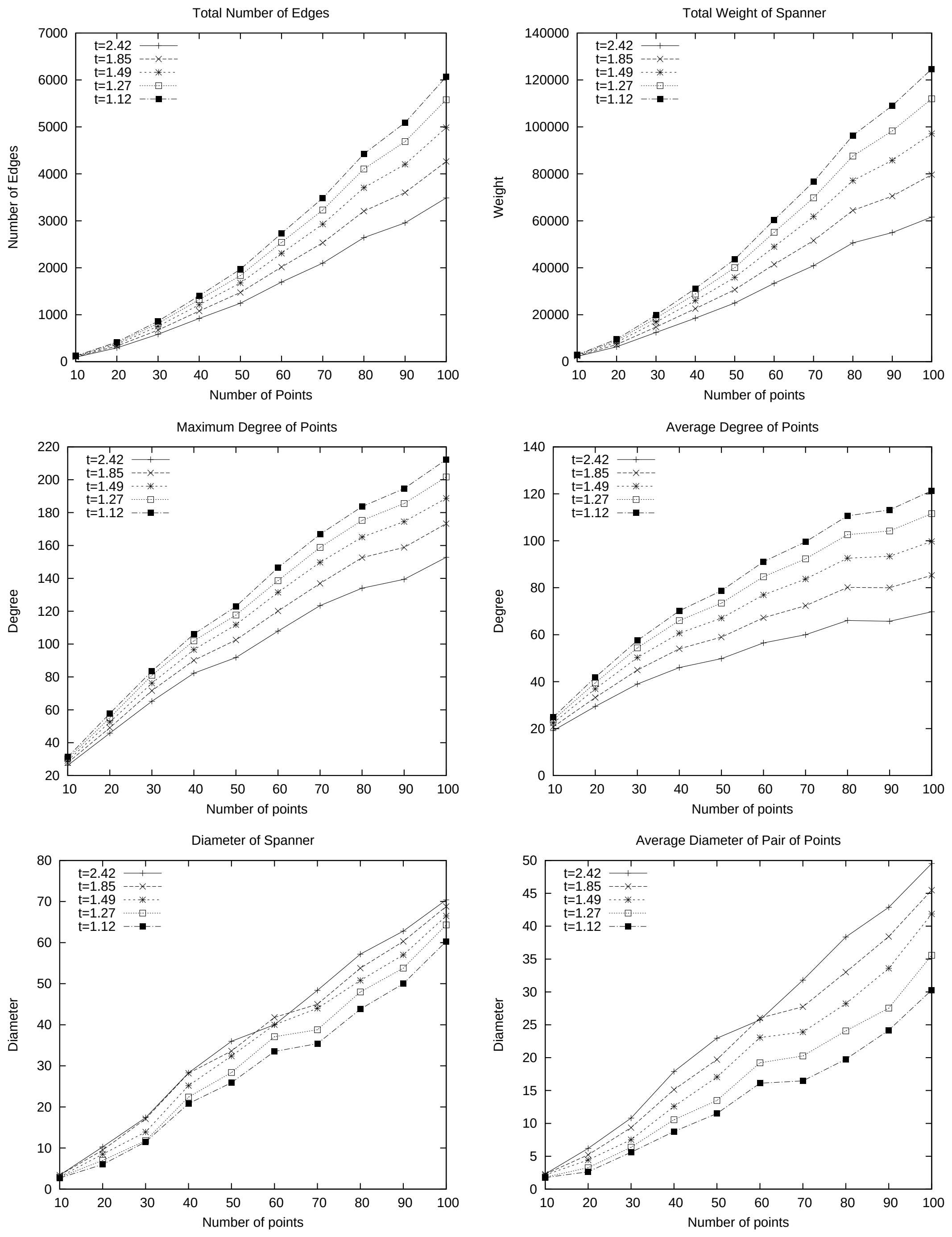


Figure A.9: Showing results for *Deformable Spanner* only containing unique edges on multivariate normal distributed data.

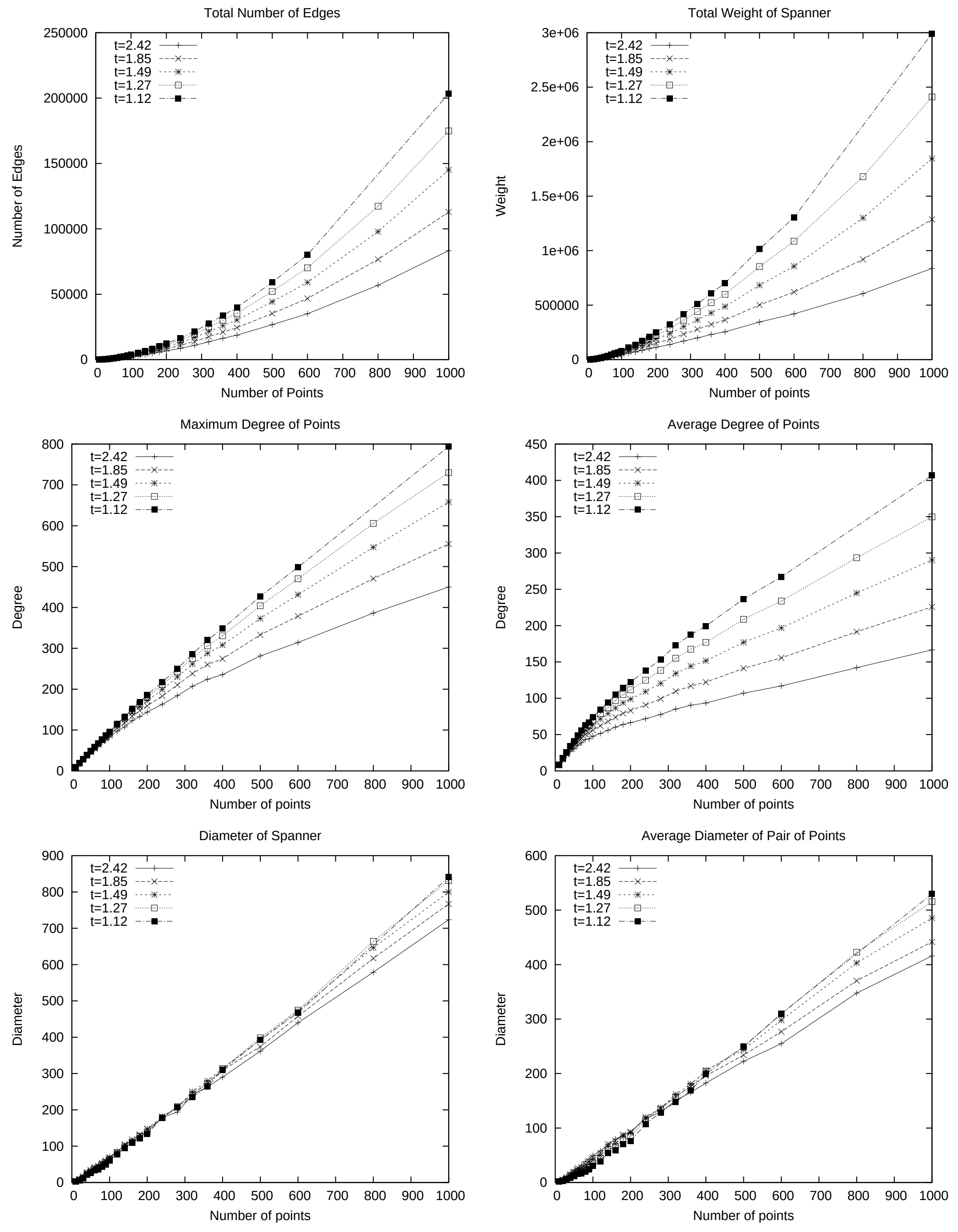
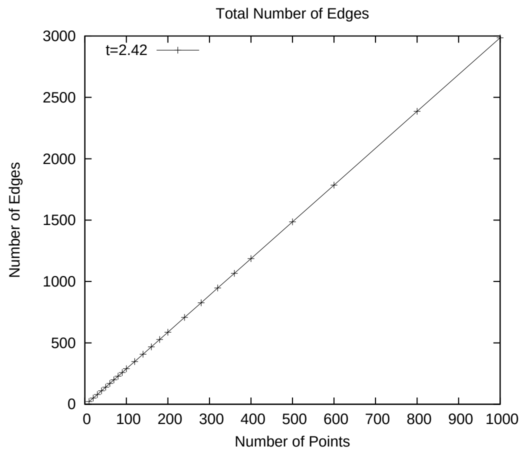
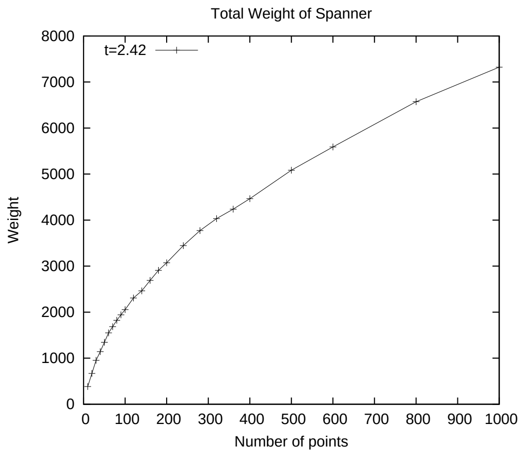
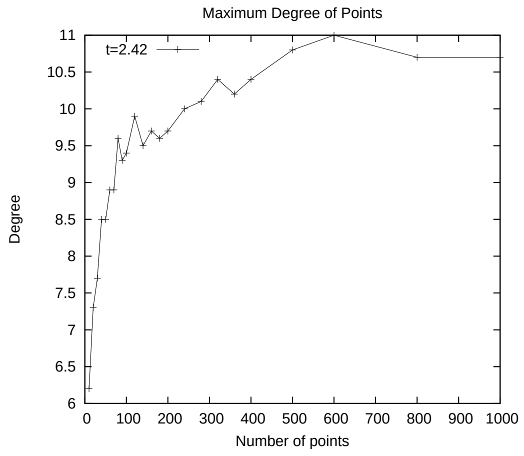
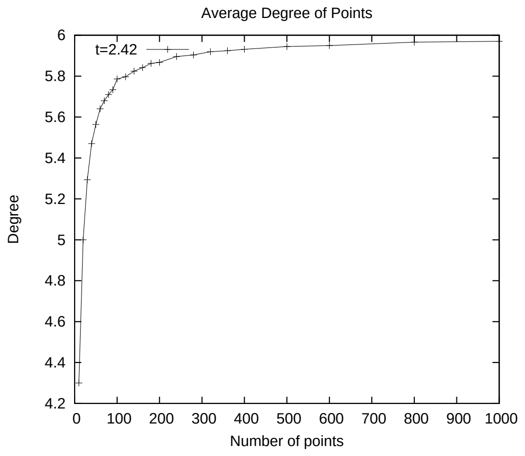


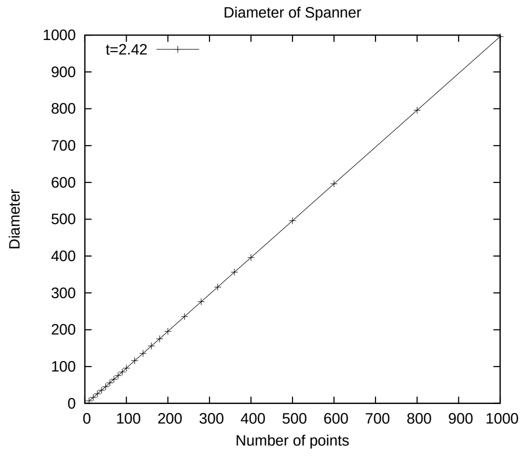
Figure A.10: Showing results for *kinetic Delaunay triangulation* on multivariate normal distributed data.

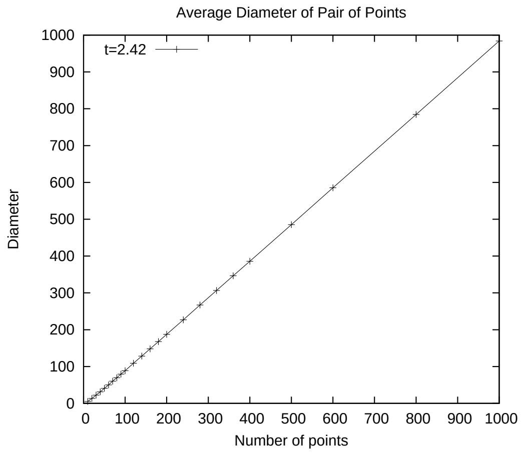












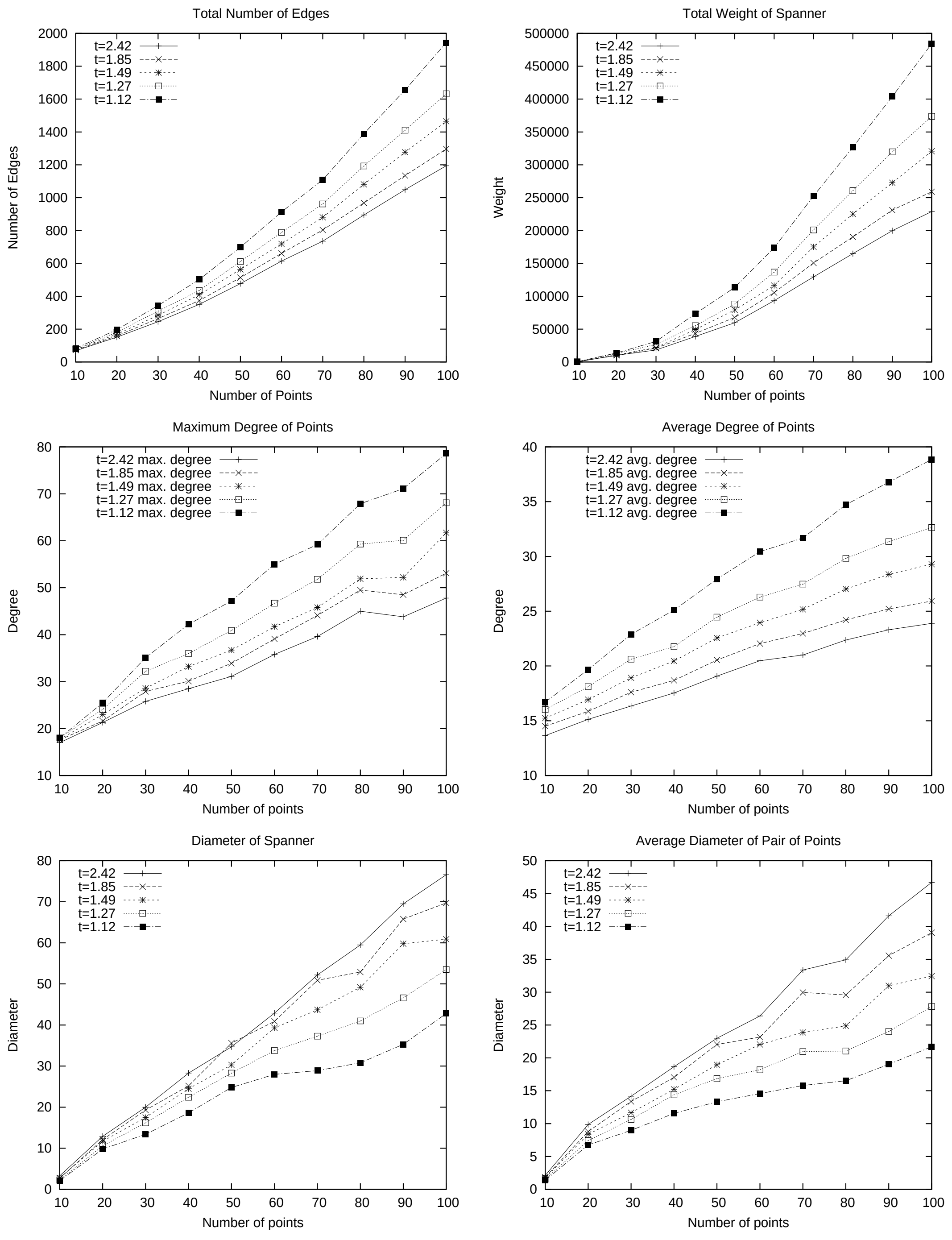


Figure A.11: Showing results for *Kinetic Spanner in* $\mathbb{R}^2$ on clustered data.

Figure A.12: Showing results for the *Kinetic Spanner in* $\mathbb{R}^2$ only containing unique edges on clustered data.

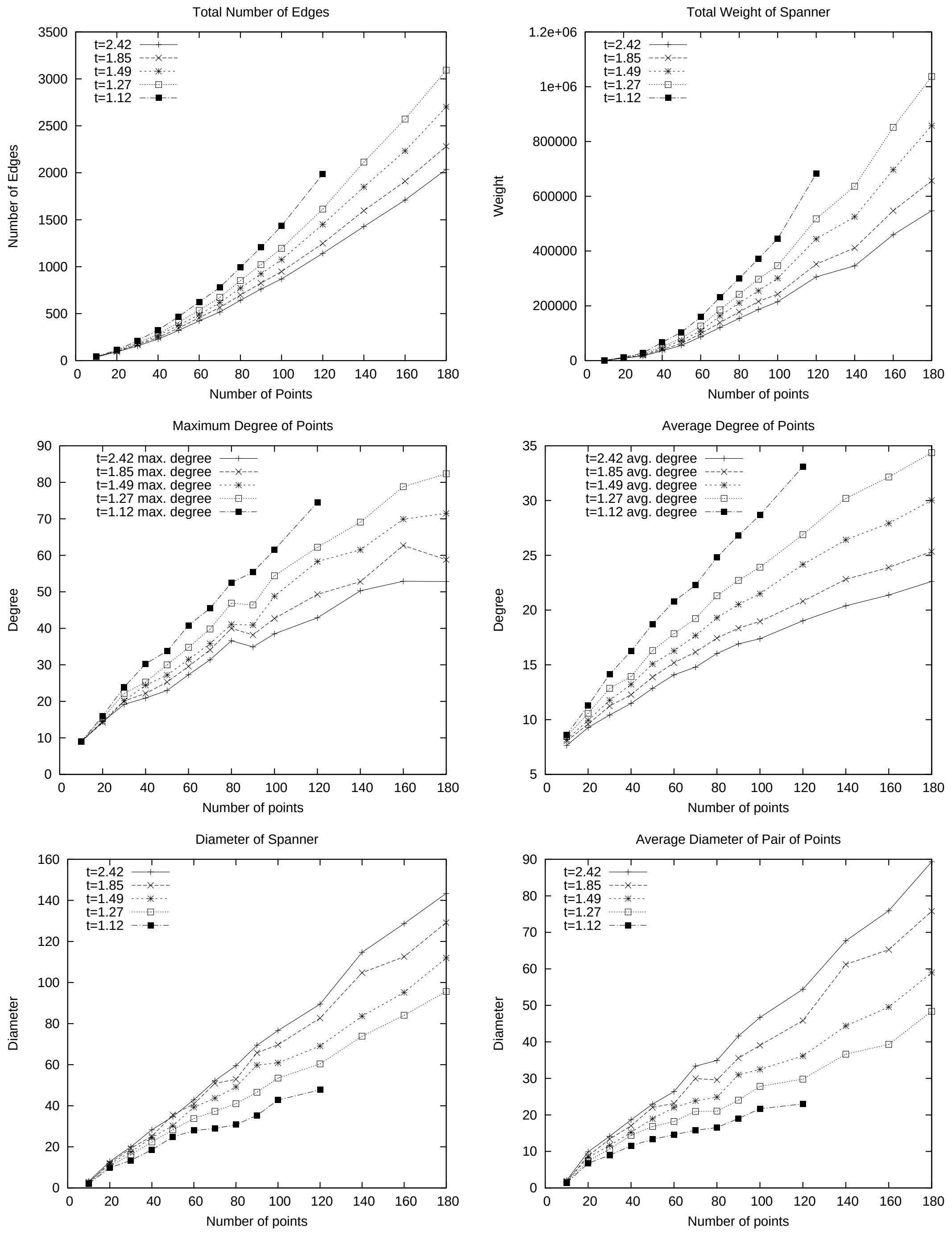


Figure A.13: Showing results for *Deformable Spanner* on clustered data.

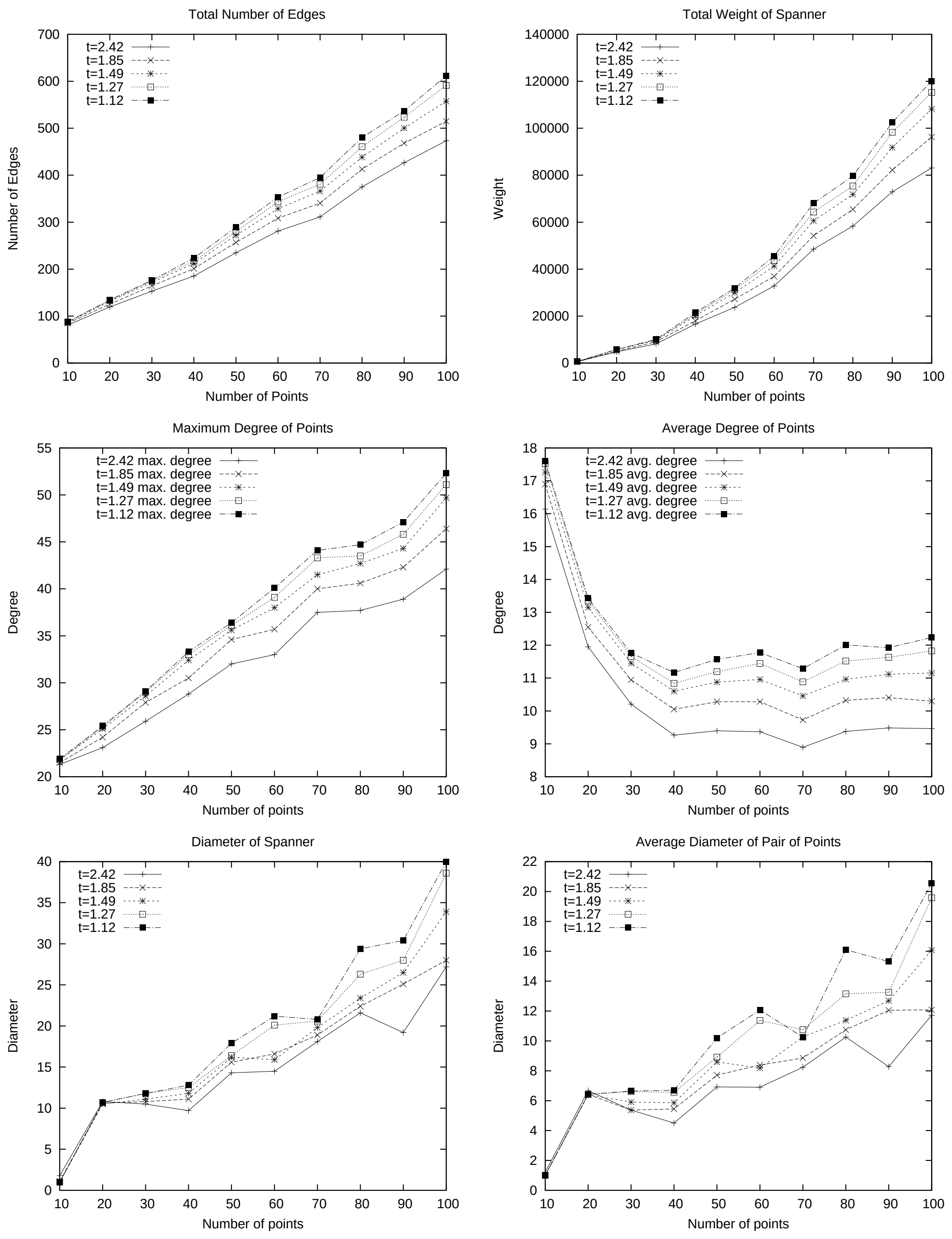


Figure A.14: Showing results for the *Deformable Spanner* only containing unique edges on clustered data.

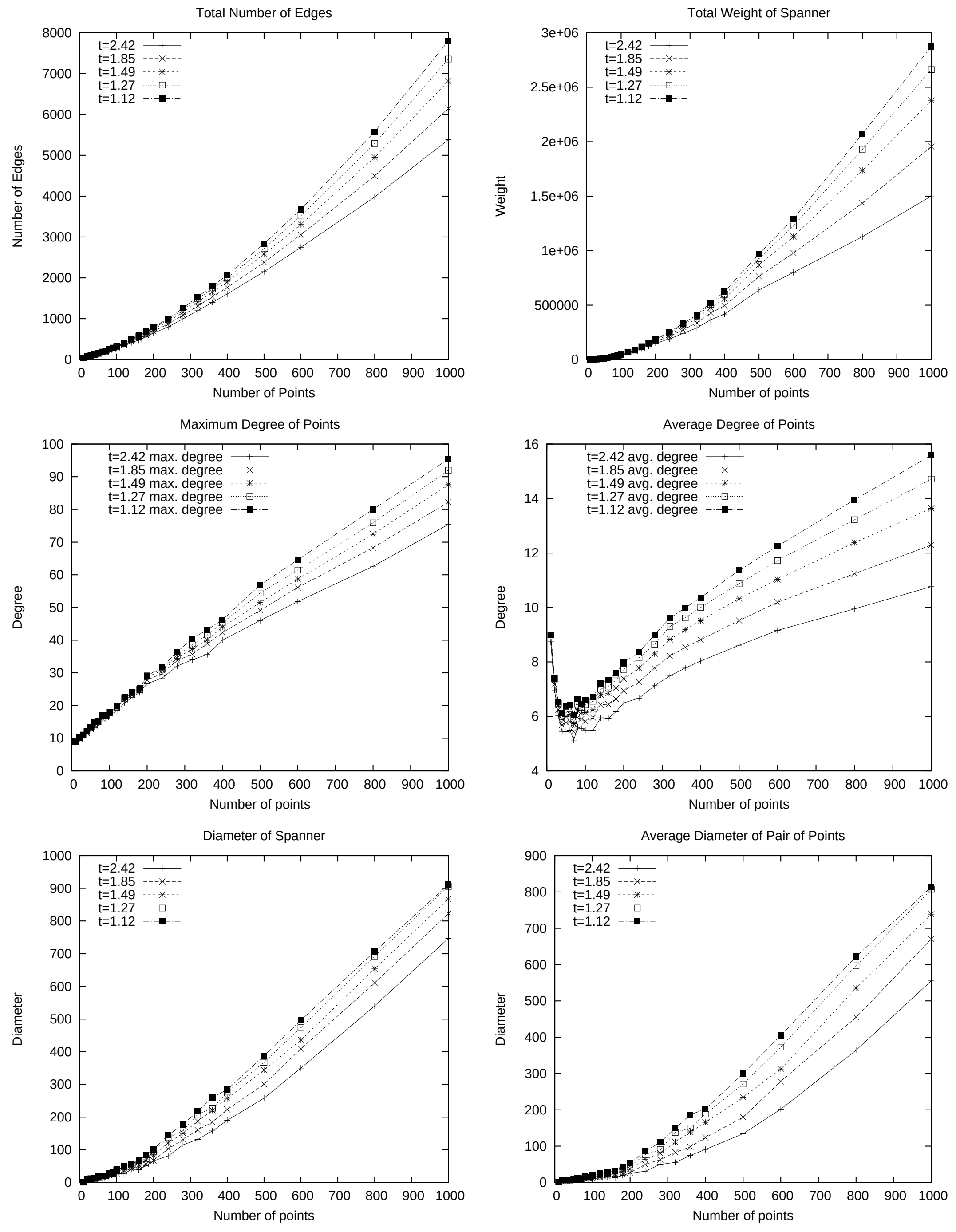
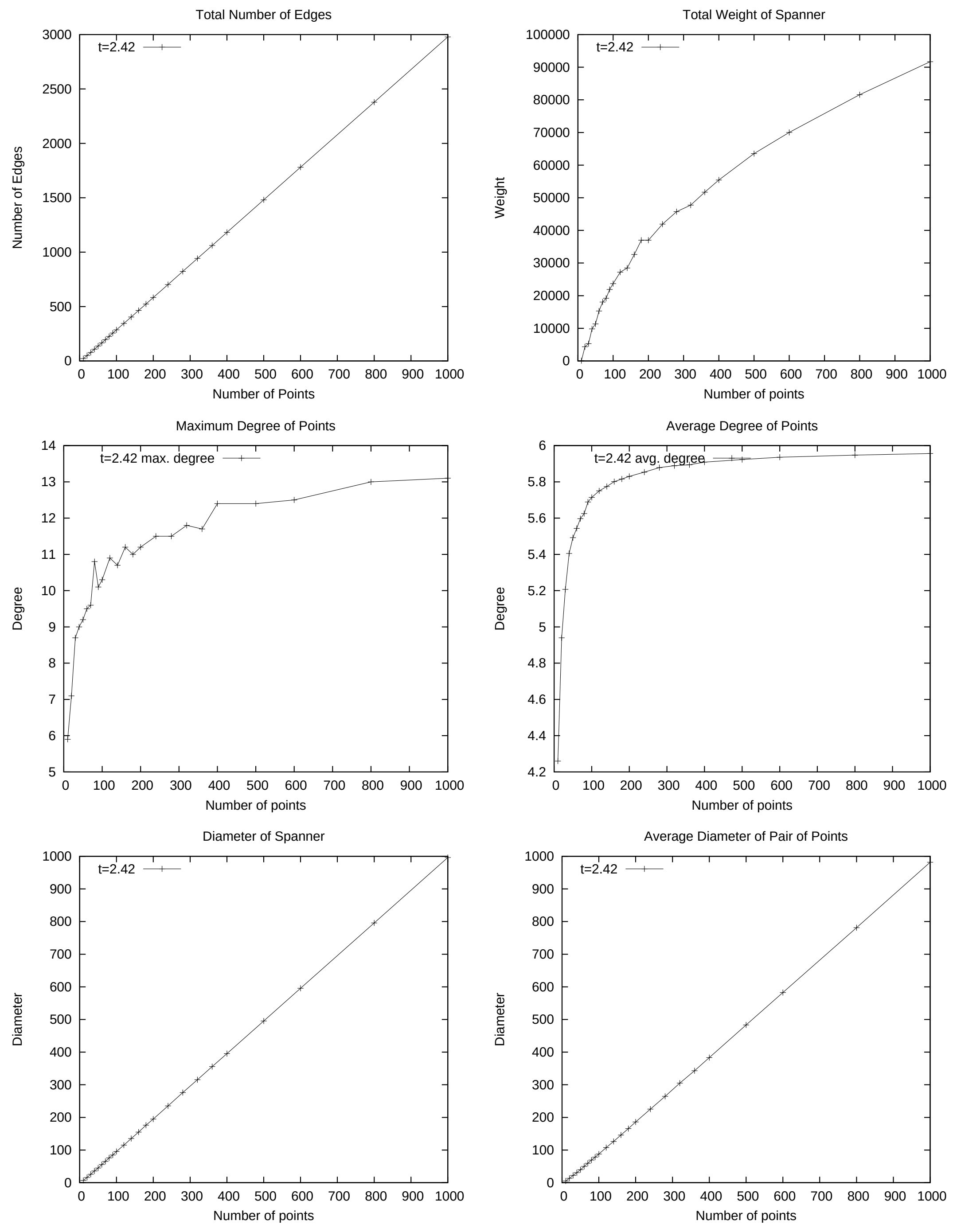


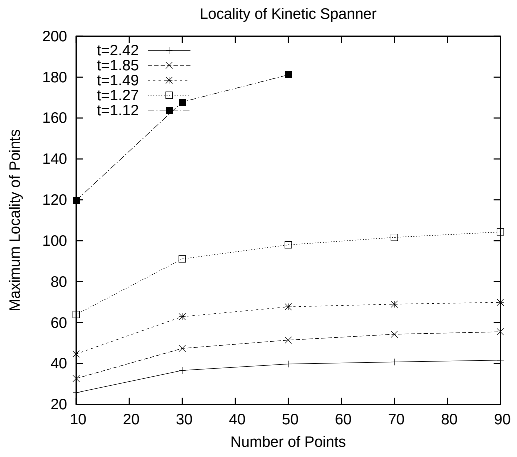
Figure A.15: Showing results for *kinetic Delaunay triangulation* on clustered data.

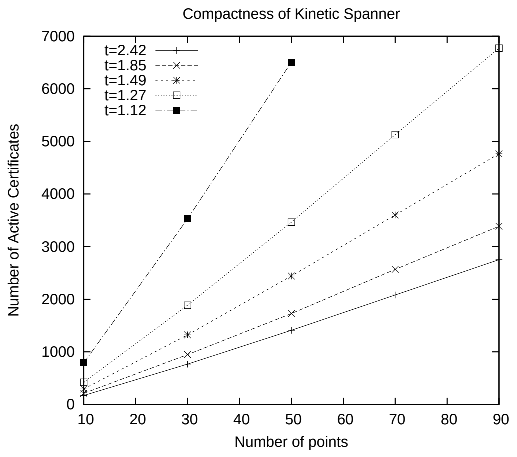


Appendix B

Experiments on Kinetic Qualitites

Figure B.1: Showing results for *Kinetic Spanner in* $\mathbb{R}^2$ on uniform distributed data.





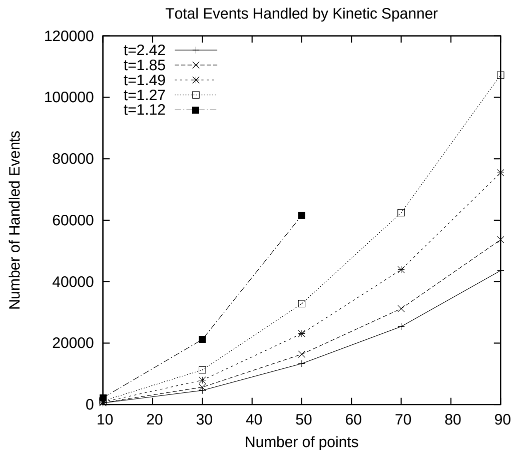


Figure B.2: Showing results for *kinetic Delaunay triangulation* on uniform distributed data.

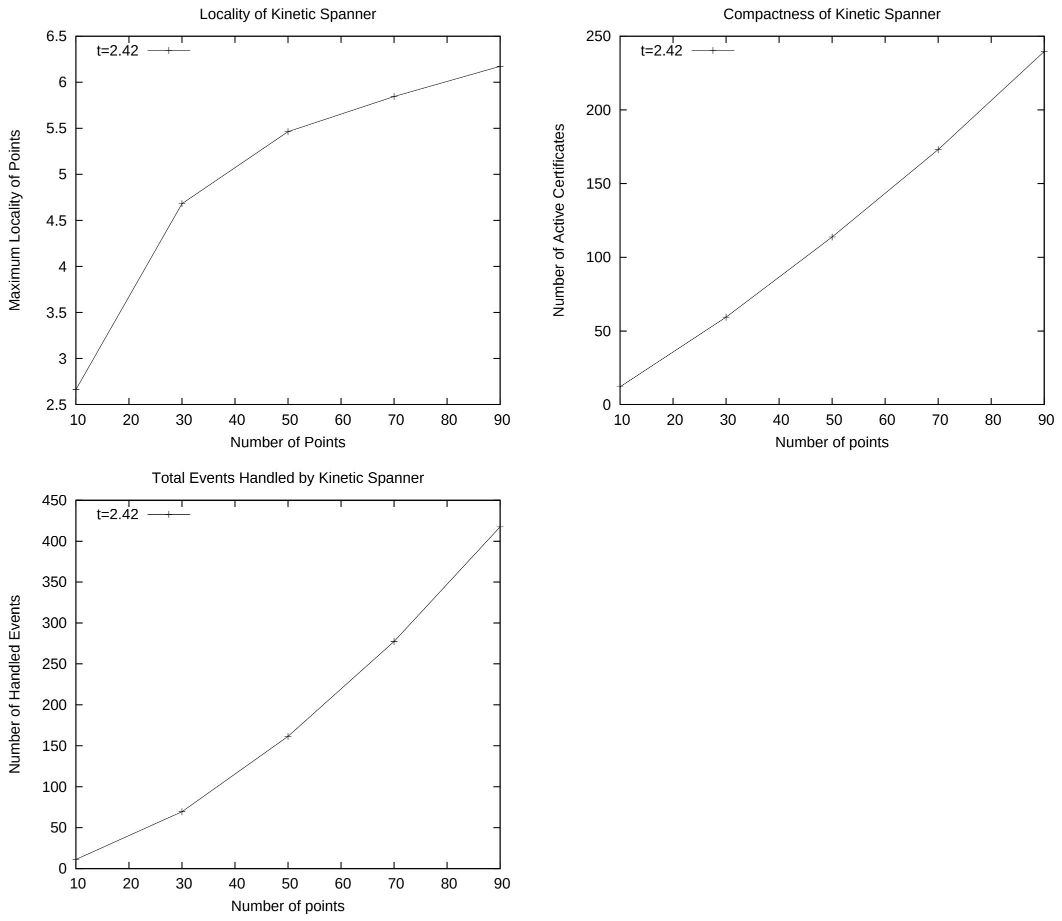
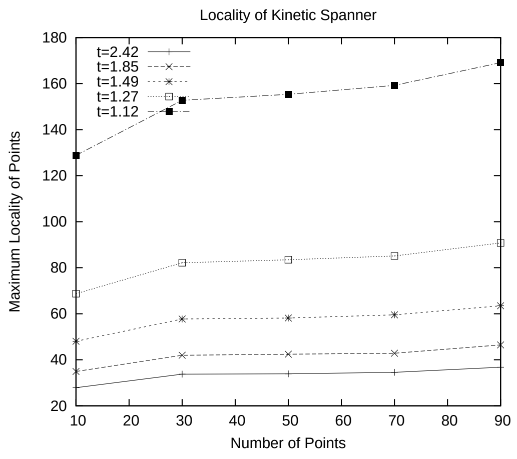
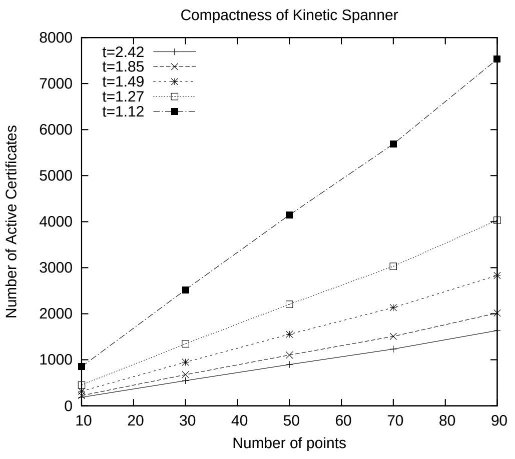


Figure B.3: Showing results for *Kinetic Spanner in* $\mathbb{R}^2$ on clustered data.





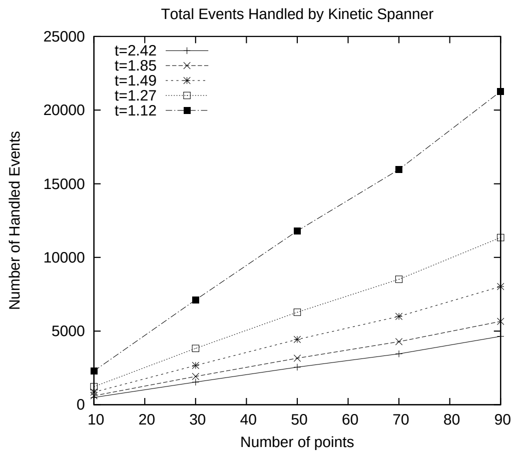
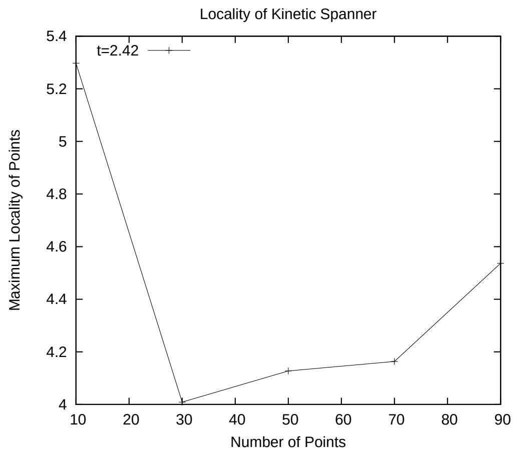
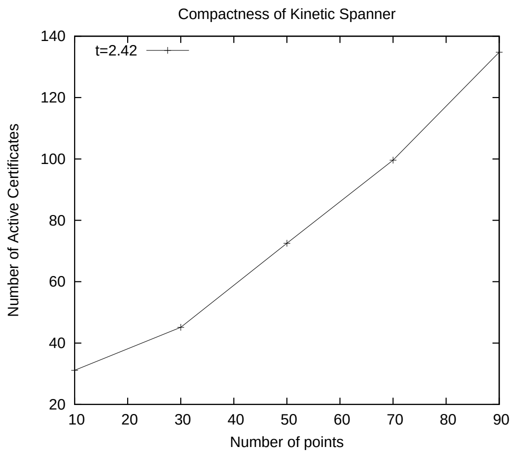
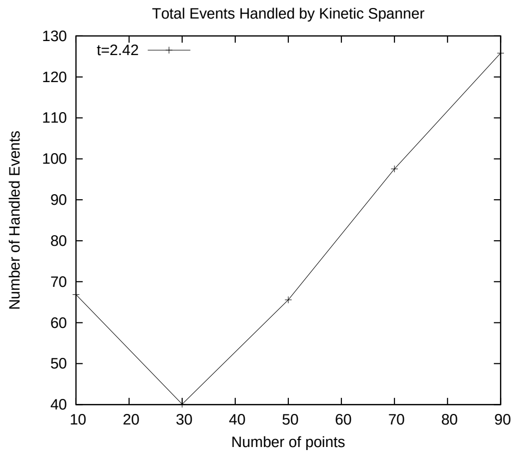


Figure B.4: Showing results for *kinetic Delaunay triangulation* on clustered data.


Locality of Kinetic Spanner
t=2.42
Maximum Locality of Points
Number of Points


Compactness of Kinetic Spanner
t=2.42
Number of Active Certificates
Number of points


Total Events Handled by Kinetic Spanner
t=2.42
Number of Handled Events
Number of points