Department of Computer Science

Master Thesis

Computational Aspects of Triangulations with Constant Dilation

Paul Seiferth

November 16, 2012

First Supervisor:
Prof. Dr. Wolfgang Mulzer

Second Supervisor:
Dr. Panos Giannopoulos

Let G be a plane geometric graph. For two distinct vertices u, v of G we can consider the ratio between the length $d_G(u, v)$ of the shortest path and the Euclidean distance $|u, v|$ between u and v. The dilation of G is the maximum over all these ratios, i.e. $\max_{u,v \in V(G)} \frac{d_G(u,v)}{|u,v|}$.

The aim of this Master Thesis is to examine the geometric properties of triangulations that have bounded dilation and to utilize them in an algorithmic way. Krznaric and Levcopoulos showed that given the Delaunay triangulation for a planar point set S we can compute a hierarchical clustering for S in linear time. We present a similar algorithm that does not insist on the Delaunay triangulation but uses triangulations that have constant dilation. Unfortunately, when analyzing the running time it turned out that the running time of the algorithm is not linear. We identified two properties that are necessary to achieve linear running time. It is left as an open question what kind of triangulations, except for the Delaunay triangulation, fulfill these properties. Furthermore, we state additional properties of triangulations with constant dilation that were encountered during the analysis of the algorithm.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben. Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskomission vorgelegt und auch nicht veröffentlicht.

Ort, Datum	Paul Seiferth

Danksagung

An dieser Stelle möchte ich Herrn Prof. Dr. Wolfgang Mulzer für die Überlassung dieses anspruchsvollen Themas, sowie der intensiven Betreuung während der Arbeit danken. Insbesondere für die Beantwortung meiner zahlreichen Fragen, egal ob sie fachlicher, technischer oder organisatorischer Natur waren. Ebenfalls bedanke ich mich bei Herrn Dr. Panos Giannopoulos für die Anfertigung des Zweitgutachtens.

Letztendlich möchte ich meiner Familie und meinen Freunden danken, die mich bis zum Ende hin immer wieder motiviert haben fokusiert zu arbeiten, und gleichzeitig für die nötige Zerstreuung in der Zwischenzeit sorgten.

Contents

List of Figures

List of Algorithms

1 Introduction

1.1 The Problem

One of the most examined structures in computational geometry are triangulations [BE95, SD95, DDMW94, MR08]. A triangulation of a finite set of planar points S is a division of S into triangles such that all corner points of the triangles are points from S. Triangulations can be interpreted as graphs and thus benefit from a rich set of graph theoretic results. For instance, since every triangulation is a planar division it belongs to the well studied class of planar graphs and all properties of planar graphs apply for triangulations as well.

A special kind of triangulation that maximizes the minimum angle of all the angles of the triangles is called Delaunay triangulation. Krznaric and Levcopoulos showed in a series of papers how the Delaunay triangulation, if given as additional input, can be exploited to obtain fast algorithms for the computation of quadtrees and solving various problems in cluster analysis [KL95a, KL95b, KL98]. Quadtrees are a commonly used datastructure in algorithmic geometry with many applications like collision detection, spatial indexing and range queries [SW88]. The goal of cluster analysis is to find similarities between objects. The objects are assigned to points in a metric space according to their properties, and sets of points that have a pairwise small distance are grouped together to form a cluster. Instead of just partitioning the whole point set into clusters, one way to identify clusters is to structure the points in a hierarchical tree-like way. The tree consist of a sequence of partitions containing the entire set of points on the top level, and a decreasing number of points on the levels below until each part of the partition consists of a single point only (see [ELL01], p.55). Cluster analysis and hierarchical clusterings have many applications, for example in market research, business intelligence, pattern recognition in images, information retrieval, artificial intelligence, machine learning, and biology [PS83, HKP12, JvR71]. Two of the most widely used and well studied hierarchical clustering methods in cluster analysis are the single and complete linkage clustering [JD88, ELL01].

The concrete results from Krznaric and Levcopoulos are that, given the Delaunay

Triangulation for a planar point set S with $|S| = n$ a quadtree for S can be computed in time $O(n)$, the complete linkage clustering can be computed in time $O(n \log^2 n)$ and the single and compute linkage clustering can be approximated in time $O(n)$. All these results rely on building an intermediate datastructure, the c-cluster tree, that speeds up the computation. Krznaric and Levcopoulos defined c-cluster trees for planar point sets and showed that they can be computed in time $O(n)$ if the Euclidean minimum spanning tree of the point set is known.

Since insisting on the Delaunay triangulation as additional input is a very strong restriction, the question that arises is whether there are other, more general geometric structures that can be exploited to speed up the computation of quadtrees or single and complete linkage clusterings. We could ask also in a more general way: What geometric structures can be exploited to solve which kind of problems how fast?

1.2 Goals

The dilation of a triangulation on a planar point set S describes the maximum detour we have to accept if we go from one point of S to another using the edges of the triangulation instead of going to it directly using the beeline. More formally, if we compare the length of the shortest path in the triangulation that connects two distinct points of S to their euclidean distance, the dilation is the maximum over all these ratios.

This thesis examines possibilities of using triangulations with dilation that is bounded by some constant as additional input in order to develop fast algorithms for different geometric problems. The goal is to obtain results similar to those of Krznaric and Levcopoulos by using triangulations with constant dilation instead of using the Delaunay triangulation. Since it is known that the dilation of the Delaunay triangulation is also bounded by some constant, this would be a generalization of the set of possible inputs that can be used to speed up the computation of a quadtree or the single and complete linkage clustering [DFS90]. The first step that was always applied by Krznaric and Levcopoulos is to compute a c-cluster tree from the Delaunay triangulation using a linear time algorithm. We will focus on reproducing this result using triangulations with constant dilation. For this purpose, in Chapter 3 we define all necessary graph theoretic notations and properties of triangulations that will be used. Afterwards, hierarchical clusterings and especially c-cluster trees are defined. It turned out that the definition of c-cluster trees used by Krznaric and Levcopoulos is to strict for our purposes and therefore we also define a relaxed version of these trees, the (c_1, c_2)-cluster trees (see Section 2.4.2). Since all the presented algorithms use the minimum spanning trees of

the triangulations, an algorithm that is able to extract these trees from the triangulations in linear time is explained in Section 2.6.

In the next chapter we recap the algorithm by Krznaric and Levcopoulos for the computation of a c-cluster tree from the Delaunay triangulation and explain as an example for an application of these trees how to approximate the single linkage clustering in a fast way.

In Chapter 4, we show how to extend the algorithm presented in Chapter 3 to work with triangulations with constant dilation instead of the Delaunay triangulation. Since some properties of triangulations with constant dilation are not as tight as for the Delaunay triangulation, the algorithm will not compute a c-cluster tree but the relaxed version, the (c_1, c_2)-cluster tree. When analyzing the running time of the algorithm in Section 4.4, it will turn out that the algorithm will not run in linear time. The main problem is that the minimum spanning tree of triangulations with constant dilation is not necessarily the Euclidean minimum spanning tree as it is for the Delaunay triangulation and therefore, it lacks some crucial properties. We give a counterexample to illustrate the problem and discuss possible solutions in Section 5.2. Additional properties of triangulations with constant dilation that where encountered during the research are stated in Section 4.5.

2 Definitions and Preliminaries

2.1 Graph-Theoretic Definitions

2.1.1 Basics and Special Graphs

A *graph* $G = (V, E)$ is a tuple of two disjoint sets V and E where $E \subseteq \binom{V}{2}$, i.e. the elements of E are two-element subsets of V. We call an element of V *vertex* and an element of E *edge*. If $uv \in E$ is an edge where $u \in V$ and $v \in V$ are vertices then we say u and v are *adjacent* to each other in G and u as well as v are *incident* to the edge uv. The *degree* $\deg(v)$ of a vertex $v \in V$ is the number of edges that are incident to v. A *directed* graph is a graph together with two functions $\text{init} : E \to V$ and $\text{ter} : E \to V$ which assign each edge a start vertex $\text{init}(e)$ and an end-vertex $\text{ter}(e)$, i.e. the edge e is directed and is going from $\text{init}(e)$ to $\text{ter}(e)$. The number of edges e with $\text{ter}(e) = v$ is called the *in-degree* of the vertex v and the number of edges e with $\text{init}(e) = v$ the *out-degree* of v.

For a set S we say that the graph $G = (S, F)$ is a graph *on* S. We denote by $V(G)$ the set of vertices of G and by $E(G)$ the set of edges. Another graph H is called *subgraph* of G (denoted by $H \subseteq G$) if $V(H) \subseteq V(G)$ and if every edge of H is also an edge of G, i.e $E(H) \subseteq (V(H) \times V(H)) \cap E(G)$. Now consider a subset $U \subseteq V(G)$ of the vertex set of G. A subgraph H of G is a graph *induced* by U in G if $V(H) = U$ and if for every two vertices $u, v \in V(H)$ the edge uv occurs in $E(H)$ if and only if it occurs in $E(G)$. We write $G[U]$ for the graph induced by U in G. For the subset $U \subseteq V(G)$ we denote by $I_G(U)$ the set of edges of G that are incident to exactly one vertex in U, i.e. the edges $vu \in E(G)$ with $v \in U$ and $u \in V(G) \setminus U$.

An important operation on graphs is the *edge contraction*. Given a graph G and an edge e of G we obtain a graph G' from G by contracting the edge e. Thereby, the edge e becomes a new vertex v' in G' and all the edges incident to the endpoints u and v of e will be incident to v' in G'. More formally, we say that $G' = (V', E')$ is a graph with vertex set

$$V' = (V \setminus \{u, v\}) \cup \{v'\}$$

and edge set

$$E' = \{xy \in E \mid \{x,y\} \cap \{u,v\} = \emptyset\} \cup \left\{v'w \;\; \mid vw \in E \text{ or } uw \in E\right\}.$$

Note that when applying the contraction operation repeatedly, the obtained graph G' can have self-loops or multiple edges. A self-loop is an edge vv for a vertex $v \in V(G)$, whereas multiple edges are two or more edges between the same pair of vertices.

A *path* $P = (v_1, \ldots, v_k)$ in a graph G is a sequence of pairwise distinct vertices such that $v_i v_{i+1} \in E(G)$ for $1 \leq i \leq k-1$. This means that we can walk along the edges of the graph starting from v_1 to reach the end-vertex v_k. If there is at least one path between every two vertices of G we call G *connected*. A connected graph with exactly $|V(G)| - 1$ edges is called a *tree*. If G is a graph and $H \subseteq G$ is a subgraph of G with $V(H) = V(G)$, and H is a tree, we call H *spanning tree* of G.

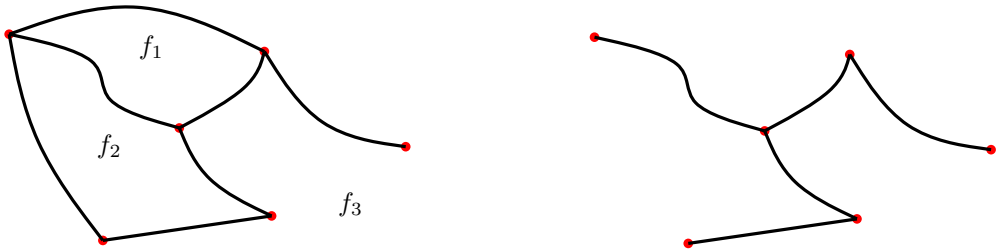

Figure 2.1: A planar graph with two bounded faces f_1, f_2, and one unbounded outer face f_3 (left), and a spanning tree of this graph (right).

2.1.2 Planar Graphs and Triangulations

Let $S \subseteq \mathbb{R}^2$ be a planar point set and G a graph on S. A *planar embedding* of G is an assignment of smooth curves to edges of G such that every edge $uv \in E(G)$ is assigned to a smooth curve $\pi(uv)$ that connects the endpoints of the edge. G is called *planar* if it can be embedded into the plane, such that the smoothed curves assigned to the edges do not cross each other except for their endpoints. In other words, the intersection of two or more smooth curves should either be empty or a single point $p \in S$. For a particular planar embedding of a graph G we call the connected subsets of the plane which we obtain when considering $\mathbb{R}^2 \setminus \bigcup_{uv \in E(G)} \pi(u,v)$ *faces*. That is to say, faces are the regions of the plane that remain if we remove the planar embedding. A relation between the number of vertices, the number of edges, and the number of faces for a planar graph is given by the well known Euler's formula ([Die06], p.99).

Theorem 2.1. (Euler's formula) Let G be a connected planar graph with $n \geq 1$ vertices, m edges, and l faces. Then it holds that

$$n - m + l = 2.$$

From Euler's formula we can deduce the following lemma which shows that every planar graph is *sparse* in the sense that the number of edges is linear in the number of vertices ([Die06], p. 100).

Lemma 2.2. For every planar graph G with $n \geq 3$ vertices the number of edges is at most $3n - 6$.

Another important property, related to planar graphs, is that all graphs obtained from a planar graph by edge contractions are planar, too.

Property 2.3. Planar graphs are closed under the edge contraction operation, i.e. if G is a graph obtained by contracting an edge of a planar graph, G will be planar as well.

A *line segment* between two points x, y is defined as

$$\{\alpha x + (1 - \alpha)y \mid \alpha \in \mathbb{R} \text{ and } 0 \leq \alpha \leq 1\},$$

the set of all points which can be expressed as convex combination of x and y. According to Fary's theorem we know that for every graph G on a planar point set S that can be embedded in the plane we can find another embedding for G (not necessarily on S but on a point set S' with the same cardinality as S) with only line segments assigned to edges [Fár48]. Since we are interested in geometric structures and their properties, going forward, we assume that every graph has a planar embedding with only straight line segments between vertices. Furthermore, when referring to a vertex we mean the point in the plane that corresponds to that particular vertex, and by an edge uv we actually mean the line segment from u to v assigned to this edge.

Now let $S \subseteq \mathbb{R}^2$ be a planar point set in general position, where general position means that there are no three points collinear, and let G be a graph on S. We call G *maximally planar* if we cannot add an edge between any two vertices in $V(G)$ without crossing some other edge in $E(G)$. The graph we obtain when starting with a graph G with no edges and adding non-crossing edges one after another until the graph is maximally planar is a geometric *triangulation*. Note that every face except the infinite boundary face (or outer face) of a geometric triangulation is a triangle. A triangulation on a point set is not necessarily unique. For example, consider two adjacent triangles $\triangle abc$ and $\triangle bcd$

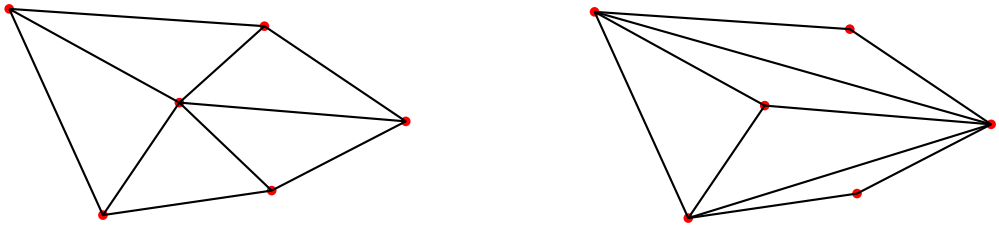

Figure 2.2: Two different triangulations on the same planar point set.

forming a convex quadrilateral in a particular triangulation. These two triangles share a common edge bc. Now we flip this edge inside the quadrilateral, i.e. substitute bc with the edge ad, in order to obtain a different triangulation. In general, we are interested in special kinds of triangulation. One example is the Delaunay triangulation which has many useful properties that will be considered in Section 2.3.

2.2 Distance Measurement in Graphs

The standard method to measure the distance between two points is the euclidean distance defined as $|ab| = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2}$, where $a = (a_x, a_y)$ and $b = (b_x, b_y)$ are two points in the plane. This distance measurement can be extended to point sets, say A and B, by computing the euclidean distance between all pairs of points $(a, b) \in A \times B$ and taking the minimum, that is,

$$d_e(A, B) = \min_{a \in A, b \in B} |ab|.$$

A *weighted graph* is a graph G and a weight function $f : E(G) \to \mathbb{R}^+$ that assigns to every edge a positive real value as weight. Since we are dealing with geometric graphs, we will simply assign to each edge the euclidean distance between its endpoints as weight. We can measure the weight of a graph by adding up the weights of all its edges. A special subgraph H of a graph G is the spanning tree with the minimum weight amongst all possible spanning trees of G. It is called the *minimum spanning tree* of G or mst(G) for short. Now let G be a complete graph on a point set in the plane, then we call mst(G) the *euclidean minimum spanning tree* G, or emst(G). If a graph is not a complete graph then there exists a pair of vertices u and v which do not share a common edge. To measure distances between u and v as well, we consider the shortest path P in G from u to v and sum up the weights of all edges contained in P. Let us denote this distance $d_G(u, v)$ and call it the *shortest path distance* between u and v. This leads

to the important notion of the *graph theoretic dilation* (also called *detour* or *distortion*) between two vertices u and v which is defined as

$$\delta_G(u,v) = \frac{d_G(u,v)}{|uv|}.$$

The dilation is a measure for the length of the detour we incur when going over the edges of the graph, instead of using the direct connection between two vertices. Graph theoretic dilation can be also defined for a whole graph G by considering the maximum dilation of all pairs of vertices of G, that is

$$\delta(G) := \max_{u,v \in V(G)} \frac{d_G(u,v)}{|uv|} = \max_{u,v \in G} \delta_G(u,v).$$

The notion of dilation can be further extended to a family of graphs $\mathcal{G}$. Then the dilation of $\mathcal{G}$ is the maximum dilation over all graphs $G \in \mathcal{G}$.

2.3 Triangulations with Constant Dilation

A family $\mathcal{T}$ of triangulations is a family of maximal planar graphs such that every finite planar point set $S \subseteq \mathbb{R}^2$ can be assigned to at least one maximal planar graph T on S with $T \in \mathcal{T}$. We write $T(S)$ for the triangulation on a particular point set S. By dilation of a triangulation $T(S)$, we mean the dilation of the family of triangulations, i.e. $\delta(\mathcal{T}) = \max_{S \subseteq \mathbb{R}^2} T(S)$, where $T(S) \in \mathcal{T}$ and $S \subseteq \mathbb{R}^2$ is a finite planar point set.

It is known that for some families of triangulations the dilation is bounded by some constant. The first example is the Delaunay triangulation, abbreviated as $DT(S)$ for a particular point set S. It is defined as follows: For a triangulation T with m triangles let $A(T) = (\alpha_1, \ldots, \alpha_{3m})$ be the *angle vector* of T, where $\alpha_1, \ldots, \alpha_{3m}$ are the internal angles of all m triangles of the triangulation sorted in ascending order. We can compare the angle vectors of two triangulations T and T' by comparing $A(T)$ and $A(T')$ lexicographically, i.e. $A(T) = (\alpha_1, \ldots, \alpha_{3m})$ is larger than $A(T') = (\alpha'_1, \ldots, \alpha'_{3m})$ if and only if there is an i with $1 \leq i \leq 3m$ such that

$$\alpha_i > \alpha'_i \text{ and } \alpha_j = \alpha'_j \text{ for } j < i.$$

The triangulation that has the minimum angle vector of all triangulations on S is called the Delaunay triangulation. An alternative characterization of the Delaunay triangulation is given by the following theorem ([dBCvKO00], Ch. 4, p. 190):

Theorem 2.4. Let $S \subseteq \mathbb{R}^2$ be a planar point set and T a triangulation on S. Then T equals $DT(S)$ if and only if the circumcircle of any triangle of T does not contain a point of S in its interior.

Assuming that S is a point set in general position, we can deduce that $DT(S)$ is unique. General position in this context means that there are no four cocircular points. Furthermore, the following important property of the Delaunay triangulation can be shown.

Property 2.5. Let $S \subseteq \mathbb{R}^2$ be a planar point set and $DT(S)$ the Delaunay triangulation of S. Then the minimum spanning tree of $DT(S)$ is the euclidean minimum spanning tree of S and therefore, the euclidean minimum spanning tree of S is a subgraph of $DT(S)$.

This property is exploited by Krznaric and Levcopoulos to compute a hierarchical cluster tree, given a Delaunay triangulation [KL95a]. A detailed explanation of their results will be given in Chapter 3. As previously mentioned, the Delaunay triangulation has constant dilation. Dobkin et al. showed that there is a universal constant $c \leq 5.08$ such that for every planar point set S the dilation of the Delaunay triangulation satisfies $\delta(DT(S)) \leq c$ [DFS90]. Keil and Gudwin improved this constant to $c \leq 2.42$, and Xia further improved it to $c \leq 1.998$ [KG92], [Xia11]. There are point sets S, found by Chew by placing points on a circle for which the dilation of $DT(S)$ is $\pi/2$ [Che86]. For a long time, this was the best known lower bound and therefore conjectured to be tight until 2010 when Bose et al. discovered point sets, for which the dilation of the Delaunay triangulation is strictly greater than $\pi/2$ [BDL$^+$10].

The next two examples of triangulations we are going to introduce are the greedy and the minimum weight triangulation. The greedy triangulation is obtained by starting with an empty graph on a planar point set and by iteratively adding the shortest non-crossing edge to this graph, as long as it is not maximal planar. The minimum weight triangulation minimizes the sum of the weights of its edges over all triangulations on a particular point set. While the computation of the greedy triangulation can be done in polynomial time, even with the naive approach, computing the minimum weight triangulation is NP-hard [MR08].

A famous result, related to the graph theoretic dilation of triangulations, is by Das and Joseph in [DJ89]. They proved that the greedy triangulation and the minimum weight triangulation both have constant dilation. More noteworthy is the framework Das and Joseph developed to obtain these results. They showed that an algorithm that computes planar graphs for any given set of points and ensures that the graphs have

two special properties, computes graphs with constant dilation only. Particularly, they proved the following theorem [DJ89]:

Theorem 2.6. Let $\mathcal{A}$ be an algorithm that computes a planar graph G on a point set S, α be an angle less then $\pi/2$, and d be a constant. If every graph G computed by $\mathcal{A}$ satisfies the following properties then there is a constant $c_{\alpha,d}$ dependent on α and d only such that G has dilation $\delta(G) \leq c_{\alpha,d}$.

i) **Diamond property**: For every edge $e \in E(G)$ one of the two isosceles triangles with base e and apex angle α contains no other point of S. This means that it cannot happen that e blocks the shortest path between two vertices that have a small euclidean distance.

ii) **Good polygon property**: For every face f of G and every two vertices u and v on the boundary of f with the line segment uv lying entirely within f, the following condition must hold: the distance $d_f(u, v)$ from u to v, with u and v lying on the the boundary of f, must be less than $d \cdot d_e(uv)$. In other words, the detour that we must take to reach v starting from u when taking the shortest path on the boundary of f instead of the line segment between u and v cannot get too large.

Note that for triangulations every face is a triangle and therefore property ii) holds trivially. This implies that every triangulation with the diamond property has constant dilation. Summarizing, some well known triangulations like the Delaunay triangulation,

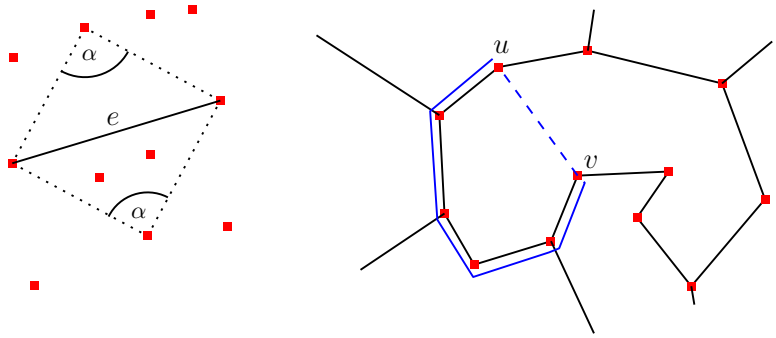

Figure 2.3: The diamond property for an edge (left) and the good polygon property for a face (right).

the greedy triangulation and the minimum weight triangulation, have constant dilation. Furthermore, we have the diamond property as a tool to show that a triangulation has constant dilation. This leads to the assumption that the class of triangulations with constant dilation is quite large and therefore needs further study.

2.4 Cluster and Cluster Trees

Points in a planar point set S must not be well distributed in general. There may be points that are far away from each other and points that are very close to each other. The ratio of the largest and smallest distance between any two distinct points in S is called the *spread* of S. Some geometric algorithms for planar point sets perform worse if the spread of the point set is to large. One example is the naive approach for computing a quadtree. A quadtree is a rooted tree whose nodes correspond to squares in the plane and whose internal nodes have exactly four children. Let B_v be the square that corresponds to the internal node v, and b the side length of B_v. Then the four children of v correspond to the four squares we obtain when partitioning B_v into four equally sized squares with side length $\frac{b}{2}$, each. We obtain a complete quadtree with the following procedure:

1. Find the smallest square B that encloses all points of S. Assign B to the root of the quadtree.
2. Partition B into four equally sized child squares B_1, B_2, B_3, B_4.
3. Recursively partition every child square B_i that contains more than one point of S.

At the end, there is a square for every point p of S such that only p is contained in this square (see. Figure 2.4). These squares correspond to leaf nodes of the tree. Clearly the size of the quadtree depends on the depth of the recursion, but the depth does not depend on the cardinality of S only, but on the spread of S (see. [dBCvKO00], Ch. 14). Therefore, if the spread of the point set is large, the size of the tree will be large, too. To be able to better deal with planar point sets that have a high spread, it is useful to find an additional structure that partitions the point set into smaller subsets that have a bounded spread, for instance, by grouping together points that have pairwise small distance. It turns out that the hierarchical c-clustering will have this property.

2.4.1 c-Cluster of Planar Point Sets

Krznaric and Levcopoulos defined the *diameter cluster* for a planar point set S [KL95a]. Let $U \subseteq S$ be a set of points. We define B_U to be the smallest axis-aligned rectangle that encloses U and call it *bounding box* of U, and denote by $\text{rdiam}(U)$ the diameter of B_U. U is a diameter cluster, if and only if $d_e(U, S \setminus U) \geq \text{rdiam}(U)$. The definition extends naturally to the following definition of a c-cluster.

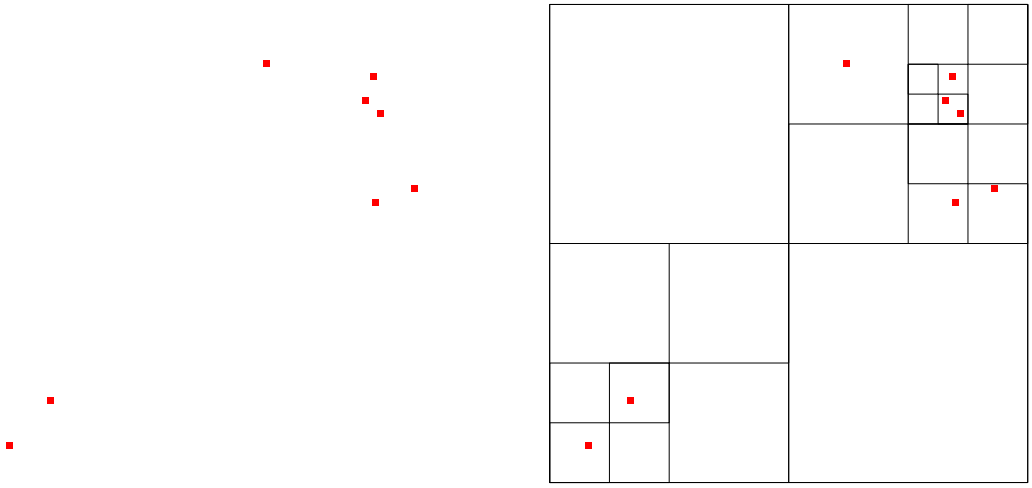

Figure 2.4: A point set with high spread and a quadtree on this point set.

Definition 2.7. Let $c \in \mathbb{R}$ with $c \geq 1$ be a constant and S a planar point set. A subset $U \subseteq S$ of S is a c-cluster if and only if $d_e(U, S \setminus U) \geq c \cdot \text{rdiam}(U)$.

This means that all points contained in U are far away from points not contained in U. Note that a diameter cluster is a c-cluster for $c = 1$ and that every point is a c-cluster itself for any c, because the diameter of the bounding box for a single point is zero.

2.4.2 Hierarchies of Clusters and Cluster Trees

We are interested in a hierarchical decomposition of point sets. This brings us to the notion of a *laminar family*. Let X be a set and $\mathcal{A}$ be a family of subsets of X. Then $\mathcal{A}$ is a laminar family, if and only if for every $A, B \in \mathcal{A}$, either A and B are disjoint or one is a subset of the other. Krznaric and Levcopoulos proved that a set of diameter clusters always forms a laminar family [KL95a]. We will show the same result for two different clusters, a c_1-cluster and a c_2-cluster, using the same proof technique.

Lemma 2.8. Let $1 \leq c_1 \leq c_2$ be two constants, $S \subseteq \mathbb{R}^2$ a planar point set, $A \subseteq S$ a c_1-cluster and $B \subseteq S$ a c_2-cluster. Then either A and B are disjoint, A is a subset of B or B is a subset of A.

Proof. Assume for the purpose of contradiction that none of the three conditions holds, i.e. neither $A \cap B = \emptyset$, nor $A \subseteq B$, nor $B \subseteq A$. Then A and B must intersect each other. Now we choose x, y, z as follows: $x \in A \setminus B$, $y \in A \cap B$ and $z \in B \setminus A$ (see Figure 2.5). Then by definition $d_e(y, z) > c_1 \cdot \text{rdiam}(A)$ because $y \in A$ and $z \notin A$

and A is a c_1-cluster. From $y \in B$ and $z \in B$ it follows that $d_e(y, z) \leq \text{rdiam}(B)$ and thus $\text{rdiam}(B) > c_1 \cdot \text{rdiam}(A)$. On the other hand, $d_e(x, y) > c_2 \cdot \text{rdiam}(B)$ since $y \in B$ and $x \notin B$ and due to $x, y \in A$ it holds that $d_e(x, y) \leq \text{rdiam}(A)$. Hence, $\text{rdiam}(A) > c_2 \cdot \text{rdiam}(B)$. With $\text{rdiam}(B) > c_1 \cdot \text{rdiam}(A)$ from above, it follows that $\text{rdiam}(A) > c_2 c_1 \cdot \text{rdiam}(A)$. This is a contradiction since we have $1 \leq c_1 \leq c_2$ and there must be at least two points, x and y, in A and thus we have $\text{rdiam}(A) > 0$. □

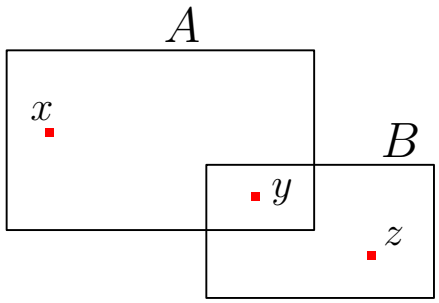

Figure 2.5: Assuming two c-clusters are not disjoint leads to a contradiction.

It follows that a set of c_1-clusters and c_2-clusters with $1 \leq c_1 \leq c_2$ always forms a laminar family. Krznaric and Levcopoulos introduced the *cluster tree* to represent a laminar family of c-clusters of a planar point set [KL95a]. Each leaf node of the cluster tree corresponds to a particular point of S and every c-cluster that contains more than one point of S is represented by an inner node of the tree. There is an edge in the tree from a c-cluster A to another c-cluster B if $B \subset A$ and if B is maximal with respect to this relation, i.e. there is no c-cluster C such that $B \subseteq C \subseteq A$. Löffler and Mulzer defined a more relaxed version of the cluster tree that in some cases can be computed more easily [LM12]. It is defined as follows:

Definition 2.9. Let $c_1, c_2 \in \mathbb{R}$ be two constants with $1 \leq c_1 \leq c_2$ and S a planar point set with $|S| = n$. A (c_1, c_2)-*cluster tree* $T_{(c_1,c_2)}$ is a rooted tree with exactly n leaf nodes, one for each point in S, such that every inner node v corresponds to a subset $S_v \subseteq S$, and with v having the following two properties:

1. If v is not the root, then $d_e(S_v, S \setminus S_v) \geq c_1 \cdot \text{rdiam}(S_v)$.
2. If S_v has a proper subset $Q \subset S_v$ such that $d_e(Q, S \setminus Q) \geq c_2 \cdot \text{rdiam}(Q)$, then there is a child w of v such that $Q \subseteq S_w$.

This means that instead of computing a c_2-cluster tree, where every inner node must correspond to a c_2-cluster, we are allowed to insert nodes which do not belong to a c_2-cluster but at least belong to a c_1-cluster in the cluster hierarchy as long as it is ensured that every c_2-cluster is represented as a node in the tree. This enables us to design less restricted algorithms to compute hierarchical clusterings for planar point sets. To assure that it is useful to compute a (c_1, c_2)-cluster tree, instead of a c_2-cluster tree, we show that, given a (c_1, c_2)-cluster tree where arbitrary many c_1-clusters were inserted in the hierarchy, we can transform it into a c_2-cluster tree.

Lemma 2.10. Given a (c_1, c_2)-cluster tree T for a planar point set S with $|S| = n$, we can contract $O(n)$ edges of T to obtain a c_2-cluster tree.

Proof. We denote by C_v the cluster that is assigned to some node v of the tree T. Assume there is at least one cluster in the hierarchy represented by T that is not a c_2-cluster. Let u be the first node that we find if we traverse T in level-order such that B_u is a c_1-cluster but not a c_2-cluster, and let w be the parent of u in T. Since the root of T is a c-cluster for any $c > 1$, and for $c = c_2$ in particular, the node w must exist in T. Let $u_1, \ldots, u_k$ be the k children of u and $w_1 = u, w_2, \ldots, w_l$ be the l children of w in T. Now we can use the contraction operation defined in Section 2.1.1 to contract the edge wu. We obtain a new tree T' with the node w', instead of the nodes u and w. Since $w_1 = u$, the node w' has the children $w_2, \ldots, w_l$ from w and $u_1, \ldots, u_k$ from u, we get that $C_{w'} = C_w$. Thus, the cluster assigned to the new node w' in T' is the same as for the old node w in T, but u does not exist anymore.

Next we show that T' is a (c_1, c_2)-cluster tree. All clusters of T' are still c_1-clusters and the only part of the tree that has changed is below the node w'. Thus, let $C \subset C_{w'}$ be a proper subset of $C_{w'}$ with $d_e(C, S \setminus C) > c_2 \cdot \text{rdiam}(C)$. Since w' in T' is equal to w in T, there was a child w_i of w in T with $C \subseteq C_{w_i}$. If $w_i \neq u$, the node w_i will be a child of w' in T' and we are done. Therefore, assume $w_i = u$. By definition is $C_{w_i} \subseteq C_u$ but since C_u is not a c_2-cluster it cannot be that $C_{w_i} = C_u$. Thus, by definition again, we know that there is a child u_j of u such that $C \subseteq C_{u_j}$. Hence, since all children of u are children of w' in T' the node u_j is a child of w' and T' is a (c_1, c_2)-cluster tree.

How many contractions do we need to remove all clusters that are not c_2-clusters? Consider a (c_1, c_2)-cluster tree for S. This tree is a rooted tree with n leaf nodes, one for each point in S. Furthermore, every inner node has degree at least two, as otherwise the cluster that is assigned to a node v with degree one is the same as the cluster assigned to the child of v. Thus, we could remove v or its child. Because a rooted tree with n leafs and with every inner node having degree at least two has a total of $O(n)$ nodes, we

can conclude that every (c_1, c_2)-cluster tree also has a total of $O(n)$ nodes. Hence, we can contract at most $O(n)$ edges.

Since a (c_1, c_2)-cluster tree contains every possible c_2-cluster of the point set by definition, we obtain a c_2-cluster tree after removing all clusters that are not c_2-clusters. □

We remark that this proof does not imply an efficient algorithm to transform a (c_1, c_2)-cluster tree into a c_2-cluster tree as it is not known if we can efficiently decide whether a node v belongs to a c_1- or a c_2-cluster. Furthermore, the proof shows that the number of nodes in a (c_1, c_2)-cluster tree is linear in the number of points since a point set with n points can have at most $O(n)$ c_1-clusters and $O(n)$ c_2-clusters.

Another useful property of cluster trees that Krznaric and Levcopoulos showed is that for every c-cluster C of a cluster tree, the spread of C is bounded by a function that depends on the number of children of C only. More precisely, they proved the following lemma [KL95a]:

Lemma 2.11. Let C be an arbitrary c-cluster with $m \geq 2$ children. Let l' be the longest distance between two distinct children of C, and let l be the shortest of those distances. Then the ratio l'/l is less than $(2 + c)^{m-1}/c$.

We will show that the same property also holds for a similar ratio between the longest and the shortest distance between the children of a (c_1, c_2)-cluster tree.

Lemma 2.12. Let C be an arbitrary cluster of a (c_1, c_2)-cluster tree with $m \geq 2$ children. Let l' be the longest distance between two distinct children of C, and let l be the shortest of those distances. Then the ratio l'/l is less than $(1 + c_2 + \frac{c_2}{c_1})^{m-2}(2 + c_1)/c_1$.

Proof. Let A_1 and A_2 be two children of C such that their distance is minimal over all pairs of children, and thus $d_e(A_1, A_2) = l$. Since A_1 and A_2 are at least c_1-cluster it holds that $\text{rdiam}(A_1) \leq l/c_1$ and $\text{rdiam}(A_2) \leq l/c_1$. Now consider their union $A_1 \cup A_2$. Then it must hold that

$$\text{rdiam}(A_1 \cup A_2) \leq \text{rdiam}(A_1) + d_e(A_1, A_2) + \text{rdiam}(A_2),$$

and by plugging in the bounds from above we obtain

$$\begin{aligned} \text{rdiam}(A_1 \cup A_2) &\leq l/c_1 + l + l/c_1 \\ &= l(2 + c_1)/c_1. \end{aligned}$$

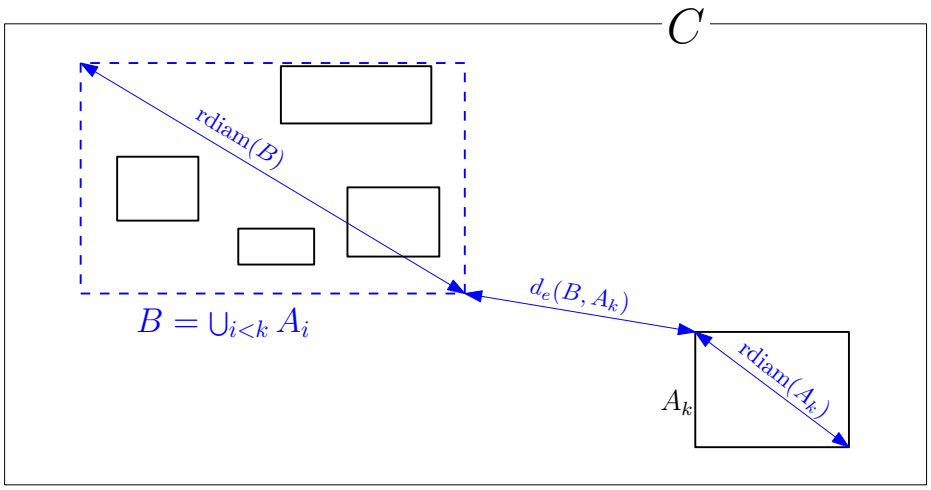

Figure 2.6: We chose A_k such that $d_e(B, A_k)$ is minimal.

Now assume that we have chosen $k-1$ distinct child clusters $A_1, \ldots, A_{k-1}$ of C such that for every $1 < i \leq k-1$ the distance $d_e(\bigcup_{j<i} A_j, A_i)$ is minimal. Let $B = \bigcup_{i<k} A_i$ be the union of the first $k-1$ child clusters of C that we have chosen, and choose A_k such that $d_e(B, A_k)$ is minimal. We then bind $\text{rdiam}(B \cup A_k)$ from above, where it must hold that

$$\text{rdiam}(B \cup A_k) \leq \text{rdiam}(B) + d_e(B, A_k) + \text{rdiam}(A_k).$$

Since A_k is at least a c_1-cluster, we know that $\text{rdiam}(A_k)$ is less than $d_e(B, A_k)/c_1$ and thus we get

$$\text{rdiam}(B \cup A_k) \leq \text{rdiam}(B) + d_e(B, A_k) + d_e(B, A_k)/c_1.$$

Furthermore, $A_1, \ldots, A_k$ are children of the same cluster C and therefore $d_e(B, A_k)$ must be less than $c_2 \cdot \text{rdiam}(B)$. Otherwise B be would be a c_2-cluster itself and thus must be a child cluster of C in the cluster tree. But then $A_1, \ldots, A_{k-1}$ would be child clusters of B. This leads to

$$\begin{aligned}\text{rdiam}(B \cup A_k) &\leq \text{rdiam}(B) + c_2 \cdot \text{rdiam}(B) + c_2 \cdot \text{rdiam}(B)/c_1 \\ &= \text{rdiam}(B)(1 + c_2 + c_2/c_1).\end{aligned}$$

Now we can devise the following recurrence relation that describes the maximum distance

l' between two children of C if C has m children

$$A(m) \leq (1 + c_2 + c_2/c_1)A(m-1),$$

with initial value

$$A(2) \leq l(2 + c_1)/c_1.$$

Solving this recurrence relation leads to $A(m) \leq l(1+c_2+c_2/c_1)^{m-2}(2+c_1)/c_1$ and thus the ratio l'/l is less than $(1 + c_2 + c_2/c_1)^{m-2}(2 + c_1)/c_1$. □

2.4.3 Linkage Clusterings

Another type of a hierarchical clustering is the linkage clustering. We start with a point set S and define that every point in S is a cluster itself. Then we merge the two clusters that have the smallest distance according to some distance measure as long as there is more than one cluster left, i.e. we construct the hierarchy in a bottom up manner. This kind of hierarchical clustering is called an *agglomerative* clustering. Depending on the distance measure that is used, we can distinguish between various kinds of linkage clusterings. One of the famous ones is the *single linkage* clustering where the distance between two clusters is just the euclidean distance between the point sets forming the clusters. In other words, we take the minimum distance between all pairs of points (a, b) with $a \in A$ and $b \in B$. Instead of the minimum we can use the maximum to measure the distance between two clusters. The clustering we obtain in this way is called the *complete linkage* clustering. In Section 2.4.3 we can observe that the single and complete linkage clustering method can end up in different clusterings for the same point set. In the first step, both will merge the points v_1 and v_2, because their distance is minimal. Let C be the cluster that arose from merging v_1 and v_2. In the single linkage clustering the next merging will be between C with v_3 as the distance between C and v_3 is the distance between v_2 and v_3 and this is less than the distance between v_3 and v_4. However, in the complete linkage clustering the distance between C and v_3 is the distance between v_1 and v_3 and this is greater than the distance between v_3 and v_4. Therefore, the second merging in the complete linkage clustering will merge the points v_3 and v_4.

There are algorithms known as SLINK and CLINK to compute the single linkage and the complete linkage clustering in time $O(n^2)$ [Sib73, Def77]. We will show in Section 3.2 how to obtain faster algorithms that approximate these clusterings using c-cluster trees.

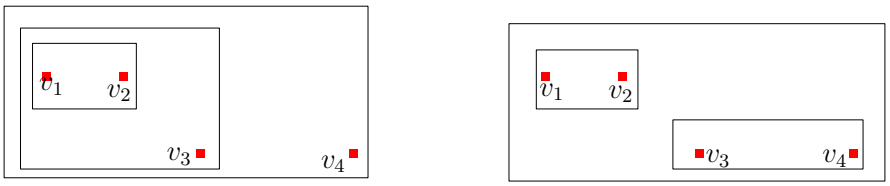

Figure 2.7: The single linkage clustering (left) differs from the complete linkage clustering (right) on the same point set.

2.5 Models of Computation

In theoretical computer science mathematical models that abstract actual computers are used to simplify the design and analysis of algorithms. As an advantage, we do not depend on architecture specific instruction sets that a particular computer can use. Thus, we can describe algorithms in an abstract way using an abstract language called *pseudocode*. Therefore, the model of computation must be designed to cover a wide range of computer architectures. When defining computational models, it is desired that all computations that can be done on the computational model can be also done on an actual computer in basically the same time using the same amount of space, i.e. the computational model should not have more computational power than real computers. Two commonly used models of computation are the *random access machine* and the pointer machine [AUH83, BKRW02].

2.5.1 RAM - Random Access Machine

A random access machine (RAM) consists of an infinite sequence of storage cells $C_1, C_2, C_3, \ldots$ storing arbitrary natural numbers, each and the following instructions to modify the content of the storage cells:

$A := B$ op C	Assign the value of B op C to A, where op is one of the following operations: addition, subtraction, multiplication, division.
$A := B$	Assign the value of B to A.
goto L	Jump to line L in the program.
GGZ B, L	Jump to line L if $B > 0$.
GLZ B, L	Jump to line L if $B < 0$.
GZ B, L	Jump to line L if $B = 0$.
HALT	Terminate program.

In this instruction set A can be either C_i or (C_i) and B, C and L can be $C_i, (C_i)$ or some constant $k \in \mathbb{N}_0$. Thereby C_i refers to the content of the cell, whereby (C_i) refers to the content the cell C_j with j being the content of cell C_i, i.e. we can use indirect addressing. A program of a RAM is a numbered sequence of these instruction. We assume that each of these instructions can be executed in constant time. Note that we can alter the content of a cell just by knowing its number, that is, we have random access to the storage cells.

In computational geometry we extend the notion of a RAM to a Real RAM by allowing arbitrary real values, instead of just natural numbers as content of the storage cells. Furthermore, in the Real RAM model we assume that the standard operations multiplication and addition of two real values and commonly used operations in geometric computation (like trigonometric function, floor and ceiling functions and computation of the square root) can be done in constant time for arbitrary real numbers. However, the Real RAM model, as defined above, is too strong. If we are allowed to use the floor function for arbitrary real values, we can even decide PSPACE complete problems [Sch79]. To address this problem, we either allow only a restricted version of the floor function or simply forbid it completely. Another computational model that uses the second approach and does not support the floor function is the *pointer machine*.

2.5.2 Pointer Machine

In the pointer machine model the input data is represented as a directed graph where every vertex of the graph has a constant out degree. The actual data is represented in *records*, where each record contains a constant number of real or integer values. A record is assigned to a vertex of the graph. For the computation we get a constant number of pointers to vertices of the graph and are allowed to traverse the graph to access a record of a particular vertex. We can alter records in constant time using the same operations as in the Real RAM model except for the floor and ceiling functions. Note that we cannot access records or vertices in constant time using indirected addressing, like in the RAM model, since finding a particular vertex requires a traversal of the graph. Therefore, we cannot use powerful computational methods that insist on random access, like hashing. Thus, the pointer machine model is weaker compared to the Real RAM model, and it is usually better to design algorithms that run on a pointer machine and do not insist on the floor function and indirect addressing.

2.6 Computing the Minimum Spanning Tree

One of the most famous greedy algorithms for computing the minimum spanning tree of a graph G is Kruskal's algorithm [Kru56]. Kruskal's algorithm sorts the edges of the graph G according to their weights and adds them in increasing order to form a spanning tree of G. The running time is $O(m \log n)$ where m is the number of edges and n the number of vertices of G [CLRS01]. Even if we restrict G to be a planar graph and use the sparsity lemma (Lemma 2.2) to bound the number of edges, we have a running time for Kruskal's algorithm of $O(n \log n)$. Since we need to sort the edges according to their weights and sorting n items requires $\Omega(n \log n)$ time (see [CLRS01], p.192), this is the best we can achieve using Kruskal's algorithm. However, using the idea suggested by Borůvka, the minimum spanning tree computation for planar graphs can be done in linear running time [Bor26]. A concrete algorithm that can be implemented on a pointer machine is given by Mareš [Mar04].

Algorithm 2.6.1 Computation of the minimum spanning tree of a graph G.

1: Let $E = \emptyset$ be the set of MST edges.
2: **while** G has more than one vertex **do**
3: For every vertex $v \in V(G)$ find the shortest edge vw incident to v and add it to E'.
4: Add all edges from E' to E.
5: Contract all edges in E' to obtain a smaller graph G'
6: Clean up G'.
7: Set $G = G'$.
8: **end while**

Note that after contracting all selected edges there may be self-loops or multiple edges in G', e.g. there may be a vertex u and an edge uu or two vertices v, w together with an edge vw that occurs two times as shown in Figure 2.8. Therefore, we need to cleanup the graph G' obtained after the contraction in step 6. Below, we will explain how such a cleanup can be done in time $O(m + n)$, where m is the number of edges and n is the number of vertices of G' before the cleanup. Note that Algorithm 2.6.1 requires all weights to be distinct or otherwise self-loops may be produced making the output not be a tree anymore. We can force distinct edge weights by numbering all edges and breaking ties using the number of the edge, i.e. lexicographically compare edges using the weight and the edge number. For subsequent considerations, we assume that all edge weights are pairwise distinct.

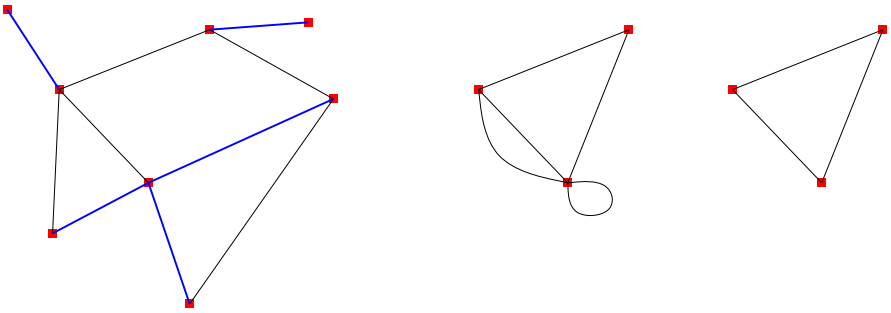

Figure 2.8: One iteration of the algorithm. The blue edges are selected by the algorithm (left). After contracting them there may be self loops and double edges (middle) that will be removed in the cleanup step to obtain a smaller graph (right).

Since in each iteration of this procedure the graph gets smaller and since the initial graph G is finite, this procedure must terminate. Like for most of the minimum spanning tree algorithms the correctness follows from the following lemma.

Lemma 2.13. Let G be a graph and $A, B \subseteq V(G)$ sets of vertices from G such that $B = V(G) \setminus A$. If $I_G(A, B) = \{ab \in E(G) \mid a \in A \text{ and } b \in B\}$ is the set of vertices with exactly one endpoint in A and one in B, the shortest edge in $I_G(A, B)$ is an edge of $\text{mst}(G)$.

Proof. For the sake of contradiction assume that the shortest edge $uv \in I_G(A, B)$, with $u \in A$ and $v \in B$, is not an edge of the minimum spanning tree $\text{mst}(G)$. Since $\text{mst}(G)$ is a tree there is a unique path $P = (u = p_1, p2, \ldots, p_l = v)$ between u and w and since it is connected there are vertices p_i, p_{i+1} in P with $p_i \in A$ and $p_{i+1} \in B$. Adding the edge uv to $mst(G)$ forms a cycle in with p_ip_{i+1} lying on this cycle. Because uv is the shortest edge in $I_G(uv)$ and all edge weights are distinct we can obtain a spanning tree with weight less than $\text{mst}(G)$ by removing p_ip_{i+1}. That contradicts the minimality of $\text{mst}(G)$. □

This lemma enables us to prove that the above algorithm indeed computes the minimum spanning tree.

Lemma 2.14. Algorithm 2.6.1 computes correctly the minimum spanning tree for a planar graph G.

Proof. Let G_i be the graph after the i-th iteration of the while loop. G_0 will be the initial graph G. By using Lemma 2.13 with $A = \{v\}$ for $v \in V(G_i)$ we know that all

edges that will be selected during the algorithm are edges of mst(G). It remains to show that the graph, that is obtained at the end of the algorithm, is a tree.

We show by induction that after each iteration of the while loop all vertices of G_i represent pairwise vertex disjoint subtrees in G_0 and that these trees cover all vertices in G_0. For G_0 this is clear, since a single vertex is a tree by definition. Now assume that the vertices of G_i represent subtrees in G_0 that cover all vertices of G_0, and let E_i' be the edges of mst(G) selected in iteration i. Since all edges have distinct weights, the edges E_i' do not form a cycle in G_i. Assume there is a cycle $K = (v_1, \ldots, v_k, v_1)$ and let $v_l v_{l+1}$ be the longest edge of K. Since all edge weights are distinct the edges $v_{l-1}v_l$ and $v_{l+1}v_{l+2}$ that are adjacent to $v_l v_{l+1}$ must be shorter than $v_l v_{l+1}$. But, subsequently neither for v_l nor for v_{l+1} Algorithm 2.6.1 would have selected the edge $v_l v_{l+1}$ to be part of the minimum spanning tree.

Now let $C_1, \ldots, C_k$ be the connected components of G_i induced by E_i'. Then for all j the component C_j must be a subtree in G_i, because the edges in E_i' do not form a cycle. But all vertices of C_j represent subtrees in G_0 and therefore C_j represents a subtree T_j in G_0. T_j consists of the subtrees represented by the vertices of C_j and the edges in E_i' that connect these subtrees. After the contraction step each connected component becomes a single vertex of the graph G_{i+1} and thus all vertices of G_{i+1} represent subtrees in G_0 covering all vertices of G_0. When the algorithm stops after, say k, iterations, the graph G_k consists only of a single vertex. This vertex represents a subtree in G_0 and covers all vertices. Therefore, it is a spanning tree and because each of its edges is an edge of mst(G) by Lemma 2.13 it is the minimum spanning tree. □

The next step is to prove that the running time of the algorithm is linear in the number of vertices if the graph is planar. To do this we will first argue that the contraction and the cleanup in each step of the algorithm can be implemented efficiently. Therefore, let $G_i = (V_i, E_i)$ be the graph in iteration i.

For the contraction we first identify the connected components of G_i induced by the edge set E_i' that the algorithm selected in step i. This can be done in time $O(n_i + m_i)$, where $n_i = |V_i|$ and $m_i = |E_i|$ using the breadth first search or depth first search algorithm. Then we remove all edges in E_i' and substitute the vertices of each connected component C with a single vertex v_C. Afterwards, we update the end-vertices of each edge that is left in E_i according to which connected component the end-vertex belonged to. These steps can be done in time $O(m_i)$ and thus the whole contraction step takes time $O(n_i + m_i)$.

The cleanup step has two purposes: Removing self-loops and removing multiple edges

between two vertices. Removing self-loops is easy since we just need to iterate through all edges in E_i and determine if their end-vertices are the same which can be done in time $O(m_i)$. For the multiple edges we remove all but the shortest edge. To detect multiple edges we label the vertices of G_i with numbers from 1 to n_i, and an edge between i and j with the pair (i, j) if $i < j$ or (j, i) otherwise. Such a labeling can be assigned to the graph in time $O(n_i + m_i)$. Then traditional radix sort can be used to sort the edges lexicographically. To do this, we create n_i buckets $b_1, \ldots, b_{n_i}$ and assign to each vertex i (with $1 \leq i \leq n_i$) a pointer to bucket b_i. In the first round we put each edge (i, j) in bucket b_j, i.e. we sort according to the second vertex of the edge. Then we collect all edges from the buckets in ascending order, starting from bucket b_1. In the second round we sort the edges in the same way but using the first vertex of the edge. While doing this we maintain the order of the edges we obtained from collecting them after the first round. At the end of the second round all edges will be lexicographically sorted and thus multiple edges will be grouped together. Now we need to iterate a last time through all edges, in order to remove all multiple edges except for the shortest one for each pair of vertices. The clean up step can be implemented to run in time $O(n_i + m_i)$ and since it uses only pointers within the data structures it can be implemented to run on a pointer machine. Because all other operations from Algorithm 2.6.1 can be done also on a pointer machine, the algorithm can run on a pointer machine as well.

Now we prove that the running time of the algorithm is linear in the number of vertices if the graph G is planar.

Lemma 2.15. Algorithm 2.6.1 runs for a planar graph G with n vertices in time $O(n)$.

Proof. Let $G_i = (V_i, E_i)$ be the graph that is created by the algorithm after the i-th iteration, $|V(G_i)| = n_i$ be the number of vertices, and $|E(G_i)| = m_i$ the number of edges of this graph. Selecting the shortest incident edge for each vertex v_i of G_i in iteration i takes time $O(m_i)$ by simply iterating through all vertices of G_i(every edge e will be checked exactly two times by each of its incident vertices). As argued above the cleanup step can be done in time $O(n_i + m_i)$ and thus the total time in iteration i is $O(n_i + m_i)$. By Property 2.3 we know that because G_0 is planar every graph G_i that arose from a sequence of edge contractions and deletions from G_0 must be planar, too. Hence, by Lemma 2.2 the number of edges m_i in G_i is $O(n_i)$ and thus the running time in iteration i is $O(n_i)$ for planar graphs.

In each iteration we select one edge for every vertex but the same edge can be selected by at most two distinct vertices. Therefore, at least $\lceil \frac{n_i}{2} \rceil$ edges are selected and

afterwards contracted in iteration i. Because every contraction operation reduces the number of vertices by exactly one, the graph G_{i+1} has at most $n_{i+1} = \lfloor \frac{n_i}{2} \rfloor$ vertices. Starting with G_0 and n_0 vertices, we obtain after at most $k = \log n_0 + 1$ steps a graph G_k with only one vertex and every graph G_l with $l \leq k$ has at most $\lfloor \frac{n_0}{2^l} \rfloor$ vertices. The total running time $T(n)$ can be bounded by

$$T(n) \leq \sum_{l=0}^{\log n_0 + 1} \frac{n_0}{2^l} \in O(n),$$

using that $\sum_{l=0}^{n} \frac{1}{2^l}$ is the geometric series and less than 2 for every n. □

Lemma 2.14 together with Lemma 2.15 shows that we can compute the minimum spanning tree for a planar graph, and especially for a triangulation, in linear time. We state this in the next theorem.

Theorem 2.16. The minimum spanning tree of a planar graph G with n vertices can be computed in $O(n)$ time using Algorithm 2.6.1.

3 Previous Results

In this chapter we will be concerned with the algorithm presented by Krznaric and Levcopoulos for the computation of a cluster tree for a planar point set S given the Euclidean minimum spanning emst(S) of the point set [KL95a]. To understand how this algorithm works is crucial for understanding why the adaptation of this algorithm to triangulations with constant dilation instead of Euclidean minimum spanning trees (given in Chapter 4) will work, too. We closely follow the way Krznaric and Levcopoulos presented the algorithm by first showing how to compute for a particular vertex v the parent cluster of v in the cluster tree and afterwards extend this idea to compute the entire cluster tree.

3.1 Computing a Cluster Tree from the Delaunay Triangulation

3.1.1 Finding the Parent Cluster

Let $S \subseteq \mathbb{R}^2$ be a planar point set and $DT(S)$ be the Delaunay triangulation of S. Since $DT(S)$ is planar, we can obtain the minimum spanning tree G of $DT(S)$ in linear time using Algorithm 2.6.1 described in Section 2.6. According to Theorem 2.5, G is the Euclidean minimum spanning tree of S. G can be used to compute the diameter cluster tree for S, where the following observation will be extensively used [KL95a].

Observation 3.1. Let $U \subseteq S$ be a proper subset of S and let $G[U]$ be the graph in the Euclidean minimum spanning tree G of S, induced by U. Then U is a diameter cluster if and only if $G[U]$ is connected and all edges in $I_G(U)$ have length greater than rdiam(U).

We now describe how to find the parent cluster of a vertex $v \in S$. The parent cluster of v is the smallest cluster that contains v and at least one additional point of S. The idea is to maintain a subset $U \subseteq S$ that initially contains only v and augment it until Observation 3.1 can be applied.

During the execution the algorithm will maintain the following four data structures.

- A set U of vertices of S describing the partially computed parent diameter cluster of v.
- A queue Q that contains edges of $I_G(U)$ that have length less than $\text{rdiam}(U)$.
- A set P that contains the remaining edges in $I_G(U)$, i.e. those edges with length at least $\text{rdiam}(U)$.
- A set D that contains the (at most four) xy-extreme points of U which define the rectangular bounding box of U.

At any point of the algorithm we will ensure that the subgraph $G[U]$ of G induced by U is connected. This enables us to apply Observation 3.1 at some point in order to identify U as a diameter cluster. The purpose of the queue Q is to efficiently determine which vertices can be used to augment U. To be able to quickly determine the value of $\text{rdiam}(U)$ at any time, we save the extreme points of U which define the size of the rectangular bounding box of U in the set D. The set P contains edges that are incident to vertices that currently do not belong to the diameter cluster, but eventually will do later when U is growing during the algorithm. We call the 4-tuple (U, Q, P, D) the computation status of the parent diameter cluster.

To initialize these data structures at the beginning of the algorithm, we will use the following procedure.

Algorithm 3.1.1 Initialization of the computation status for a vertex $v \in G$.

1: Initialize(v):
2: $U \leftarrow \{v\}$
3: $Q \leftarrow$ the shortest edge incident upon v
4: $P \leftarrow$ all the edges incident to v that are not in Q
5: $D \leftarrow \{v\}$
6: return (U, Q, P, D)

Algorithm 3.1.1 initializes the set U with the vertex v for which the parent cluster will be computed. The shortest edge incident to v is inserted into the queue Q and all other edges incident to v are added to P. Since U initially only contains a single vertex v, v is the only candidate for the xy-extreme points. Thus, D is initialized with v, too. Algorithm 3.1.2 will find the parent diameter cluster of a particular vertex $v \in S$, given the Euclidean minimum spanning tree G of S.

Algorithm 3.1.2 Computing the parent diameter cluster for a vertex v.

```
1: ParentDiameterCluster(G, v):
2: (U, Q, P, D) ← Initialize(v)
3: while Q is non-empty do
4:     remove the first edge vu from Q (with v ∈ U)
5:     add u to U
6:     update the xy-extremes in D by comparing them with u
7:     put each edge uw that is incident to u (except for vu) in P
8:     move all edges in P of length < rdiam(U) to Q
9: end while
10: return current cluster
```

In the main loop in line 3-9 we augment U by exactly one vertex at each step. We choose this vertex u such that $d_e(U, u) < \text{rdiam}(U)$. Since we maintain the queue Q in a way that it always contains all edges in $I_G(U)$ that have length less than rdiam(U) (lines 7,8), we can use the first edge vu in Q with $v \in U$ to find the desired vertex u (line 4). Note that in case of adding u to U there must be an edge $vu \in I_G(U)$ and thus the subgraph of G induced by U is always connected. After augmenting U by some vertex the size of rdiam(U) may be increased and the xy-extreme points in D must be updated (line 6). As a consequence of the increased value of rdiam(U) there now may be edges in P that have length less than rdiam(U) but have had length at least rdiam(U) before. Therefore, when adding u to U, we first move all edges incident to u to P and afterwards move all edges in P of length less than rdiam(U) to Q. If Q is empty at some point, all edges in $I_G(U)$ must have length greater than rdiam(U) and by Observation 3.1 we know that U is a diameter cluster.

3.1.2 The Complete Algorithm and its Correctness

Algorithm 3.1.2 can be used for an extended algorithm that computes the whole cluster tree. The basic idea is that every time a cluster C with vertex set U_C is found, we contract all the edges in subgraph $G[U_C]$ induced by U_C such that $G[U_C]$ becomes a single vertex v' and we obtain a smaller spanning tree G' (see Figure 3.2). We call v' a *shrunken* vertex. Afterwards, we can apply the same algorithm to G' starting from v' to obtain the parent diameter cluster of v'. Since the goal is to compute a complete diameter cluster tree, the computation must be organized in a hierarchical way meaning that before U_C will be shrunken to a single vertex v' all child clusters of C must be already computed. For this reason we introduce an additional data structure, a stack

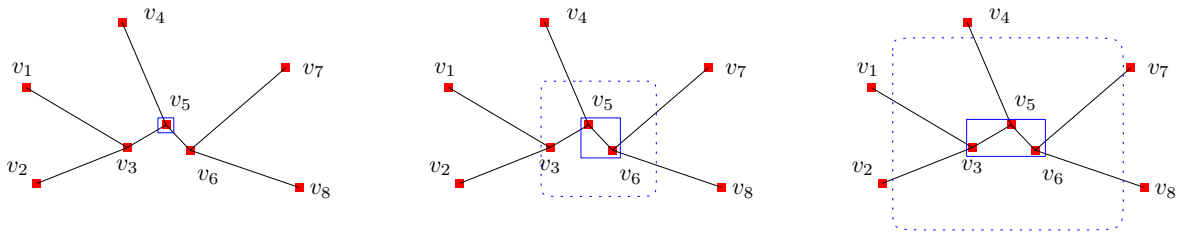

Figure 3.1: Computation of the parent diameter cluster for v_5. During the Algorithm 3.1.2 the set U (points within the blue square) increases until there are no more points with distance less than rdiam(U) left (points within the blue dashed line).

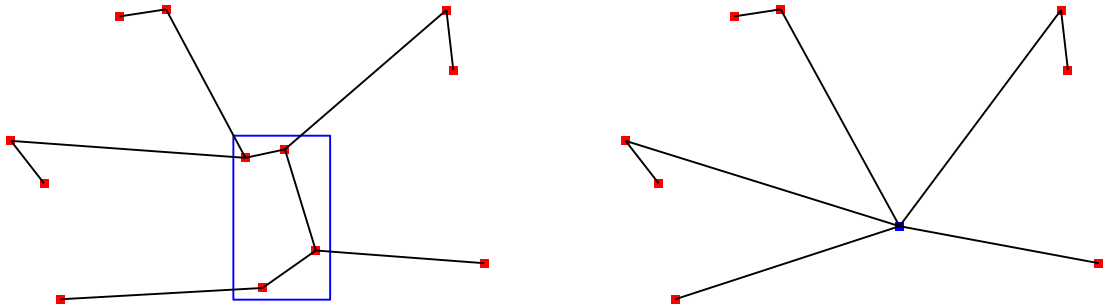

Figure 3.2: Contracting the by U induced subgraph (left) to a single vertex (right).

S. The stack is used to temporarily suspend the computation of a diameter cluster C by pushing the computation status (U_C, Q_C, P_C, D_C) for C onto the stack and start computing some child cluster C' of C to ensure that all child clusters of C will be completely computed before C itself.

Algorithm 3.1.3 Computation of the diameter cluster tree.

```
1: DiameterClusterTree(G)
2: let v be an arbitrary vertex of G
3: attach to v a pointer to Initialize(v) and push v on the stack
4: while Stack is not empty (loop1) do
5:     pop v from the stack and let (U, Q, P, D) the computation status of v
6:     while Q is nonempty (loop2) do
7:         remove the first edge uw from Q (with u ∈ U)
8:         if w is on top of stack then
9:             (U', Q', P', D') ← computation status of w
10:            U' ← U' ∪ U
11:            Q' ← Q' ∪ Q
12:            P' ← P' ∪ P
13:            D' ← xy-extremes of D' ∪ D
14:            exit loop2
15:        else if there is a shorter edge than uw incident to w then
16:            attach to u a pointer to (U, Q, P, D) and push u onto the stack
17:            attach to w a pointer to Initialize(w) and push w onto the stack
18:            exit loop2
19:        else
20:            add w to U
21:            update the xy-extremes in D by comparing them with w
22:            put each edge except uw that is incident to w in P
23:            move all edges in P of length ≤ rdiam(U) to Q
24:        end if
25:    end while
26:    contract G[U] to a single vertex v' of G
27:    if v' has some incident edges then
28:        attach to v' a pointer to Initiate(v') and push v' onto the stack
29:    end if
30: end while
```

Algorithm 3.1.3 computes the complete cluster tree. The outer loop (loop1) maintains the stack and runs as long as there is some vertex on the stack. Therefore, an arbitrary vertex is initialized using the initialization procedure from Section 3.1.1 and pushed onto the stack at the beginning of the algorithm (line 2-3). At each step of loop1 we pop

a vertex v from the stack and execute the inner loop (loop2) for v. Loop2 is similar to Algorithm 3.1.2 and its purpose is to compute the parent diameter cluster C for the vertex v. The main difference is that before we can add some vertex w to vertex set U_C of the partially computed cluster C we have do the following two sanity checks with w:

1. Is w part of another partially computed diameter cluster and lies therefore on top of the stack? (lines 8-14) This can happen during the execution of the algorithm and will be explained below.

2. Does w may belong to a child cluster of the currently computed diameter cluster C? (lines 15-18)

We first explain the second sanity check. Let C be the cluster we are currently computing and w the vertex that should be added to U_C. Then w may belong to a child cluster $\bar{C}$ of C. If there is such a child cluster $\bar{C}$ by Observation 3.1, we know that the subgraph of G induced by the vertices of $\bar{C}$ must be connected. Since every inner node of the cluster tree has at least two children, there must be a vertex w' of G that is adjacent to w such that w' is contained in $\bar{C}$. Now let uw (with $u \in U_C$) be the edge that was removed from Q_C and is responsible that w should be added to U_C. Because u is a cluster itself and also child of the cluster C, it cannot belong to $\bar{C}$, since two clusters do not intersect each other. Thus, it must hold that $d_e(w, w')$ is less than $d_e(u, w)$. If there is an edge adjacent to w that has length less than the length of edge uw, it could be that w belong to a child cluster of C. To handle this case we stop computing C by adding a pointer to the computation status $(U_C, Q_C, P_C,)$ to u, pushing u onto the stack and initialize the computation of the parent diameter cluster for w.

Now we explain the first sanity check (lines 8-14). Let $u \in U_C$ be a vertex pushed onto the stack because of the second sanity check while computing some cluster C, i.e. the computation of C was suspended and the computation of some potential child cluster of C was initialized. Let w be the vertex and uw be the edge that caused u to be pushed onto the stack so that the computation of the parent cluster of w was initialized afterwards. Later in the execution of the algorithm, we will consider u again and try to add u to some diameter cluster C' we are currently computing. Since u is also part of C and two diameter cluster do not intersect each other (see Lemma 2.8), the clusters C' and C must be the same. Therefore, their computation status' will be merged together (lines 9-13). Since the graph G is a tree, the same edge uw that caused u to be pushed onto the stack now is responsible for trying to add u to C'. Additionally, u must be on top of the stack. Note that at this point the end-vertex w of the edge uw must not necessarily be the same as at the point where u was pushed on the stack but can also

be some shrunken vertex w'. This can happen because there was some parent diameter cluster for w being a proper child cluster of C which was found by the algorithm in the meantime.

If none of the two sanity checks are true for some vertex w, this vertex can be safely added to the vertex set U_C of the cluster C we are currently computing. U_C will be augmented by vertices of G as long as all edges in the set $I_G(U_C)$ have length at least $\text{rdiam}(U_C)$. At this point we know because of Observation 3.1 that U_C is the complete vertex set of a diameter cluster C and due to the two sanity checks it is ensured that all child clusters are already computed correctly. Therefore, the subgraph $G[U_C]$ induced by U_C will be contracted to a single vertex v' and the computation of the parent cluster for v' is initialized in order to compute the next bigger cluster in the hierarchy (lines 26-28). The algorithm terminates if there is only on vertex left, i.e. all edges of the original graph G are contracted.

We remark that the algorithm not only computes the diameter cluster tree but a $c0$-cluster tree for an arbitrary $c \geq 1$. To do this we simply stop augmenting the vertex set U_C if all edges have length at least $c \cdot \text{rdiam}(U_C)$ instead of $\text{rdiam}(U_C)$ for every cluster C we are computing. Furthermore, this algorithm does not depend on random access and thus can be implemented to run on a pointer machine.

3.1.3 Running Time

A well known property of the Euclidean minimum spanning tree of a planar point set is the following:

Property 3.2. Every vertex of a Euclidean minimum spanning tree has degree at most 6.

To prove that we can use Algorithm 3.1.3 to compute a cluster tree for a planar point set S with $|S| = n$ in time $O(n)$ Krznaric and Levcopoulos showed a generalization of Property 3.2 to connected subgraphs of the Euclidean minimum spanning tree.

Lemma 3.3. Let S be a planar pointset, G the Euclidean minimum spanning tree of S and $U \subseteq S$ such that $G[U]$ is connected. Then there is at most a constant number of edges in $I_G(U)$ that have length greater than $\text{rdiam}(U)$.

Let us consider Algorithm 3.1.3. First we argue that every call of the initialize procedure takes constant time. Either the initialize procedure is called with an vertex v of the original tree G. Then, since G is the Euclidean minimum spanning for the underlying point set, we know by Property 3.2 that v has at most a constant number, namely 6,

of incident edges and all steps in the initialize procedure will take constant time. Or the initialize procedure is called with some shrunken vertex v'. Then v' corresponds to some connected subgraph of G with vertex set U. By Lemma 3.3 the number of edges in $I_G(U)$ is constant and thus initialize(v') will only take constant time.

Next we show that lines 7-24 (the inner while loop) take constant time on each execution. If we have to merge the computation status of two partially computed clusters in lines 9-13, we can merge U and U', Q and Q', P and P' in constant time. The xy-extreme points are at most four per cluster and thus merging D and D' needs at most 16 comparisons. Determining if there is an edge incident to some vertex w of a particular length can be done in constant time because of Property 3.2 and Lemma 3.3. Thus, line 16 takes constant time as well. In line 22 we put at most a constant number of edges in P. After moving all edges of length less than rdiam(U) from P to Q in line 23 by Lemma 3.3 we know that there is only a constant number of edges left in P. Hence, since after each execution of the inner loop there are only a constant number of edges left in P, we start the algorithm with a constant number of edges in P and add only a constant number of edges in each iteration we can find the edges in P of length less than rdiam(U) in constant time (line 23). Thus lines 7-24 need together only constant time for each execution.

It remains to show that the inner loop is executed only $O(n)$ times. To see this, it is easier to count the number of edges that are responsible for an execution of the inner loop since during the algorithm new vertices (shrunken vertices) will be created. Edges will only change the two vertices they are incident to because these vertices may become part of a shrunken vertex. Now let uw be some edge in the queue Q that is dequeued in line 5. Assume that uw is dequeued the first time during the algorithm and that $u \in U$ and $w \notin U$. There are two cases that can happen. First, w is added to U because it belongs to same cluster as all the vertices in U. Then, as soon as U is a complete cluster, uw gets contracted. Or second, w is not added to u. Then u will be pushed on the stack, because w may belong to a child cluster of the cluster that is currently computed. Now consider the second time uw is dequeued from some other queue Q'. At this point the endvertex w of the edge uw may be already a shrunken vertex w' and not the original w. However, when uw is dequeued the second time u must be on top of stack and the computation status attached to u is merged with the current one. Thus, w and u belong to the same cluster after the merging and the edge wu will be contracted as soon as the computation of the current cluster has finished. Therefore, each edge will be at most two times pushed on the stack and cause at most two executions of the inner loop. Since G is a tree there are exactly $n-1$ edges and the inner loop is executed only $O(n)$ times.

This argumentation leads to the following theorem from Krznaric and Levcopoulos [KL95a]:

Theorem 3.4. Given the Euclidean minimum spanning tree for a planar point set S we can compute a hierarchical cluster tree for S in linear time.

3.2 Applications of Cluster Trees

One of the applications of c-cluster trees is in cluster analysis. Usually, one is interested in more general hierarchical clustering methods like the single- or complete linkage clustering described in Section 2.4.3 instead of c-cluster trees. The reason is that these clustering methods are older and more examined and thus, there is a better understanding of their properties from a cluster-analytical point of view. However, c-cluster trees can be used to speed up for instance the computation of the complete linkage clustering resulting in an algorithm with running time $O(n \log^2 n)$ where n is the number of points of the underlying point set [KL95b]. c-cluster trees can also be used to compute approximated versions of the single and complete linkage clustering of a point set S with $|S| = n$ in time $O(n \log n)$ or even $O(n)$ if the Delaunay triangulation for S is given as additional input [KL95a, KL95b]. Another application that uses the Delaunay triangulation is to compute the quadtree for a point set S. Given the Delaunay triangulation $DT(S)$ we can do this in linear time using c-cluster trees as intermediate structures [KL98].

All these applications use the same technique. First, the c-cluster tree T is computed and then T is traversed in post order to compute the desired structure for every inner node. Thereby, when computing some inner node v of T it is assured that all children of v are already processed. For computing some inner node v the property that the clusters assigned to children of v have bounded spread, i.e. that Lemma 2.11 holds for every c-cluster tree. We will show how to compute an approximated single linkage clustering as an example for an application of c-cluster trees.

First we have do define what an approximation of a single linkage clustering is [KL95a].

Definition 3.5. A hierarchy of clusters is an $\varepsilon-$*approximation* of a single linkage clustering if its clusters can be produced by sequence of mergings in the following way. After an arbitrary number of mergings, let d be the distance between the two closest clusters and d' the distance between the two clusters that will be merged in the next step, then it holds that $\frac{d'}{d} \leq 1 + \varepsilon$.

Let T be the diameter cluster tree for a point set S with $|S| = n$ and denote by C_v the set of points of the cluster that is assigned to the inner node v of T. Note that for

every inner node v the pointset C_v is a single linkage cluster because all points in C_v have distance less then $\text{rdiam}(C_v)$ to each other and distance greater then $\text{rdiam}(C_v)$ to points in $S \setminus C_v$. Therefore, every single linkage cluster algorithm has to merge all the points in C_v before merging them with a point in $S \setminus C_v$.

The algorithm for computing c-cluster trees produces besides T for every node v of T a set of edges E_v such that every edge in E_v has endpoints in two distinct child clusters of v and every edge in E_v is an edge of the Euclidean minimum spanning tree of the underlying point set S. By Lemma 2.11 we know that the ratio between the length of the longest and the shortest edge in E_v is at most 3^{m-1}. To compute an approximation of the single linkage clustering for each inner node v of T the following steps are done. First we determine the length l of the shortest edge in E_v, create $m-1$ buckets and put all edges of E_v in the buckets such that bucket b_i contains the edges with length in $[3^{i-1}l, 3^i l)$. Then we choose a constant a such that $n^{-a} \leq \varepsilon$ and assign each edge e its order number depending on the bucket we put e in. The order number for an edge e in bucket b_i is $\lceil \frac{|e|n^a}{3^{i-1}l} \rceil$ where $|e|$ denotes the length of e. In the next step we collect all edges from all nodes of T and put them together in a set L. Then we sort all elements in L according to their order number with the radix sort algorithm (see. [CLRS01], p. 174). This can be done in time $O(n)$ since by definition the order number is less or equal than $3n$. Afterwards, we put all edges back to the same bucket we took them out but in ascending order respective to their order number. To obtain the approximation of the single linkage clustering we traverse T in post order and merge for each inner vertex v all the child clusters of v by considering every edge e in E_v and merging the two clusters that have endpoints of e. The merging order is defined through the buckets, starting with bucket b_1 and within each bucket through the order number of the edges. To keep track of the mergings we can use the Union-Find data structure [Tar75]. This leads to an overall running time of $O(n\alpha(n))$ where $\alpha(n)$ is the inverse of the Ackermann function.

Note that we only sorted the edges approximately but indeed this is sufficient for our sequence of mergings to produce an $(1+\varepsilon)$ approximation of the single linkage clustering. Suppose that we have merged an edge e_1 but there was an edge e_2 in the same bucket b_i with greater or equal order number as e_1 but length less than e_1, i.e. it holds that $|e_2| < |e_1|$. Then we have $\frac{|e_2|n^a}{3^{i-1}l} - \frac{|e_1|n^a}{3^{i-1}l} < 1$ or equivalently $|e_2| < |e_1| + 3^{i-1}l/n^a$. Now consider the ratio between $|e_2|$ and $|e_1|$:

$$\frac{|e_2|}{|e_1|} < \frac{|e_1| + 3^{i-1}l/n^a}{|e_1|}$$

and since $|e_1|$ must be greater than $3^{i-1}l$ we have

$$\frac{|e_2|}{|e_1|} < 1 + \frac{1}{n^a} \leq 1 + \varepsilon$$

because $\frac{1}{n^a}$ was chosen to be less than ε.

4 Algorithm

In this chapter we will use Algorithm 3.1.3 from Chapter 3 to compute a (c_1, c_2)-cluster tree from a triangulation with constant dilation. The (c_1, c_2)-cluster tree is a relaxed version of the cluster tree used by Krznaric and Levcopoulos introduced in Section 2.4. Unlike the Delaunay triangulation, a triangulation with constant dilation does not necessarily posses the Euclidean minimum spanning tree as subgraph. However, we can nevertheless extract the minimum spanning tree from a triangulation with constant dilation and use it as input for the algorithm from Krznaric and Levcopoulos.

4.1 What can go wrong?

Let $\mathcal{T}$ be a family of triangulations with constant dilation $d = \delta(\mathcal{T})$ and $T \in \mathcal{T}$ a triangulation on a planar point set S. Let the graph G be the minimum spanning tree with respect to Euclidean weights of T (not necessarily the Euclidean minimum spanning tree for S). Now consider an invocation of the Krznaric-Levcopoulos algorithm (Algorithm 3.1.3) from an arbitrary vertex $v \in S$ and let $U \subseteq S$ be a set of vertices for the parent diameter-cluster of v that the algorithm discovered, i.e. for all edges uw in $I_G(U)$ it holds that $d_e(u, w) > \text{rdiam}(U)$. Since we cannot assure that the minimum spanning tree of T is the Euclidean minimum spanning tree of S, there may be a vertex $a \in S \setminus U$ with $d_e(U, a) \leq \text{rdiam}(U)$ (see Figure 4.1) and there is no edge $ua \in E(G)$ with $u \in U$. Then the graph $G[U]$ induced by U will be contracted to a single vertex although it is not a valid diameter-cluster. This situation can occur when there is an edge xy in the triangulation T that prevent the existence of the edge ua because ua would have to cross xy. This is valid as long as the shortest path distance $d_T(u, a)$ is not longer than $d \cdot d_e(u, a)$, i.e. the detour we have to take to circumvent the edge xy when going from u to a is consistent with the dilation of the triangulation.

This leads to worry that maybe there is a c-cluster that does not correspond to a connected subgraph of $\text{mst}(T)$ at all. To assure that it is useful to consider connected subgraphs of $\text{mst}(T)$ when searching for c-clusters in triangulations with constant dilation we will prove the following structural lemma.

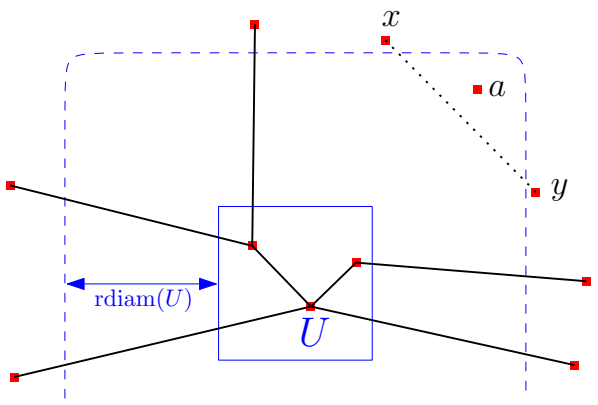

Figure 4.1: A situation where the Krznaric-Levcopoulos algorithm fails. The by U induced graph $G[U]$ will be contracted but is not a valid cluster because of a.

Lemma 4.1. Let $\mathcal{T}$ be a family of triangulations with constant dilation $d = \delta(\mathcal{T})$ and $T \in \mathcal{T}$ a triangulation on a planar point set S with minimum spanning tree G. If $U \subseteq S$ is a c-cluster with $c \geq d$ then the graph $G[U]$ induced by U is connected.

Proof. Assume for the purpose of contradiction that $G[U]$ is not connected. Then there exist two vertices u, v such that there is no path between u and v in $G[U]$. But there must be such a path in G because G is connected. Let $P = (p_1, \ldots, p_l)$ with $p_1 = u$ and $p_l = v$ this path. Since P is not completely contained in $G[U]$ there exists p_i, p_{i+1} such that $p_i \in U$ and $p_{i+1} \in S \setminus U$ (see Figure 4.2). From the definition of a c-cluster it follows that

$$d_e(p_i, p_{i+1}) > c \cdot \text{rdiam}(U) \geq c \cdot d_e(u, v)$$

where the last inequality comes from the fact that u and v are contained in U. Now consider the shortest path $L = (l_1 = u, \ldots, l_k = v)$ between u and v in the triangulation T. Since T has dilation d, the length of L is $d_T(u, v) \leq d \cdot d_e(u, v)$ and therefore every edge e of L has length less than $d \cdot d_e(u, v)$ as well. Because we choose $c > d$ it follows that $c \cdot d_e(u, v) > d \cdot d_e(u, v)$ and thus every edge in L is shorter than $p_i p_{i+1}$. If we now remove the edge $p_i p_{i+1}$ we split G into two parts, one containing v and the other one containing u. Since L is a path from u to v, we can find an edge $l_j l_{j+1}$ in L such that adding this edge leads to a new spanning tree G'. But the length of $l_j l_{j+1}$ is less then the length of $p_i p_{i+1}$ and therefore the weight of G' is less than the weight of G. This contradicts the minimality of G. □

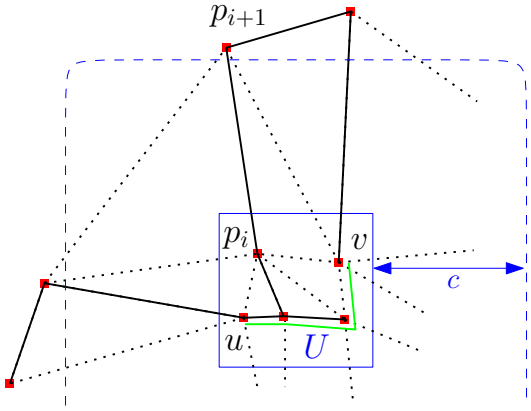

Figure 4.2: A triangulation (black dotted) of a point set and its spanning tree (black) and the shortest path L between the vertices u and v in the triangulation (green).

4.2 The Algorithm

The algorithm for computing a (c_1, c_2)-cluster tree from a triangulation with constant dilation is basically the same as the one we saw in Chapter 3 for computing a cluster tree from the Delaunay triangulation. Nevertheless we will state the algorithm for completeness.

Algorithm 4.2.1 Computation of the (c_1, c_2)-cluster tree.

```
1: (c1, c2)-ClusterTree(G)
2: let v be an arbitrary vertex of G
3: attach to v a pointer to Initialize(v) and push v on the stack
4: while Stack is not empty (loop1) do
5:     pop v from the stack and let (U, Q, P, D) the computation status of v
6:     while Q is nonempty (loop2) do
7:         remove the first edge uw from Q (with u ∈ U)
8:         if w is on top of stack then
9:             (U', Q', P', D') ← computation status of w
10:            U' ← U' ∪ U
11:            Q' ← Q' ∪ Q
12:            P' ← P' ∪ P
13:            D' ← xy-extremes of D' ∪ D
14:            exit loop2
15:        else if there is a shorter edge than uw incident to w then
16:            attach to u a pointer to (U, Q, P, D) and push u onto the stack
17:            attach to w a pointer to Initialize(w) and push w onto the stack
18:            exit loop2
19:        else
20:            add w to U
21:            update the xy-extremes in D by comparing them with w
22:            put each edge except uw that is incident to w in P
23:            move all edges in P of length ≤ c2 · rdiam(U) to Q
24:        end if
25:    end while
26:    contract G[U] to a single vertex v' of G
27:    if v' has some incident edges then
28:        attach to v' a pointer to Initialize(v') and push v' onto the stack
29:    end if
30: end while
```

The Algorithm 4.2.1 works similar to the algorithm described in Chapter 3. Let T be a triangulation on a planar point set S and G the minimum spanning tree of T. As in the Krznaric-Levcopoulos algorithm we take a vertex $v \in S$ and augment a subset $U \subseteq S$ containing v with vertices that are incident to edges in $I_G(U)$ but are not contained in

U to obtain the parent cluster of v. But instead of stopping the augmentation when every edge in $I_G(U)$ has length greater than rdiam(U), we stop if each of those edges has length at least $c_2 \cdot \text{rdiam}(U)$ for some $c_2 > 1$. Then we contract $G[U]$ to a single vertex as done in the Krznaric-Levcopoulos algorithm as well. As explained in Section 4.1 we cannot be sure that U is a c_2-cluster at this time. The idea is to prove that U is at least a c_1-cluster for some $1 \leq c_1 \leq c_2$ and afterwards show that the algorithm will find every possible c_2-cluster and thus computes correctly a (c_1, c_2)-cluster tree. Thereby we want to exploit the fact that T is a triangulation from a family of triangulations with constant dilation.

4.3 Correctness of the Algorithm

Let $\mathcal{T}$ be a family of triangulations with constant dilation $d = \delta(\mathcal{T})$ and $T \in \mathcal{T}$ a triangulation on a planar point set S. Furthermore, let G be the minimum spanning tree of T, $U \subseteq S$ the subset of vertices the algorithm maintains during its execution and $G[U]$ the subgraph induced by U in G. Note that by the way the algorithm augments U it follows that $G[U]$ is always connected. The first step in proving the correctness will be to show that every time the algorithm contracts $G[U]$ to a single vertex, i.e. find some cluster in the hierarchy, this is at least a c_1-cluster. In order to do this we need the following claim.

Claim 4.2. Let $c > 1$ be a constant, T a triangulation on a planar point set S with minimum spanning tree G and let $U \subseteq S$ be a subset of S such that $G[U]$ is connected. If all edges in $I_G(U)$ have length greater or equal than $c \cdot \text{rdiam}(U)$ then for every edge $xy \in I_T(U)$ with $x \in U$ and $y \in S \setminus U$ it holds that $d_e(U, y) > (c-1)\text{rdiam}(U)$.

Proof. We define the region $A = \{b \in \mathbb{R}^2 \mid d_e(U, b) < (c-1)\text{rdiam}(U)\}$ as the set of points that have distance less or equal $(c-1)\text{rdiam}(U)$ from U (the region within the blue dotted line in Figure 4.3). Now assume there is a vertex $a \in S \setminus U$ that lie within A and there is an edge $va \in E(T)$ with $v \in U$. Because $d_e(U, a) < (c-1)\text{rdiam}(U)$ the edge va has length at most $c \cdot \text{rdiam}(U)$. Namely from v to the vertex $u \in U$ that minimizes $d_e(U, a)$ which is at most rdiam(U) and from u to a which is at most $(c-1)\text{rdiam}(U)$. Since the length of av is less then $c \cdot \text{rdiam}(U)$ it is not an edge of G because $a \notin U$ and every edge in $I_G(U)$ is longer than $c \cdot \text{rdiam}(U)$. Because G is connected, there is a path $P = (p_1 = v, \ldots, p_l = a)$ from v to a in G. This path must use an edge $p_i p_{i+1}$ from $I_G(U)$ since $a \notin U$. Adding av to G forms a cycle containing $p_i p_{i+1}$. If we now remove $p_i p_{i+1}$ we obtain a spanning tree with weight less than the weight of G because av is shorter than $p_i p_{i+1}$. This contradicts the minimality of G. □

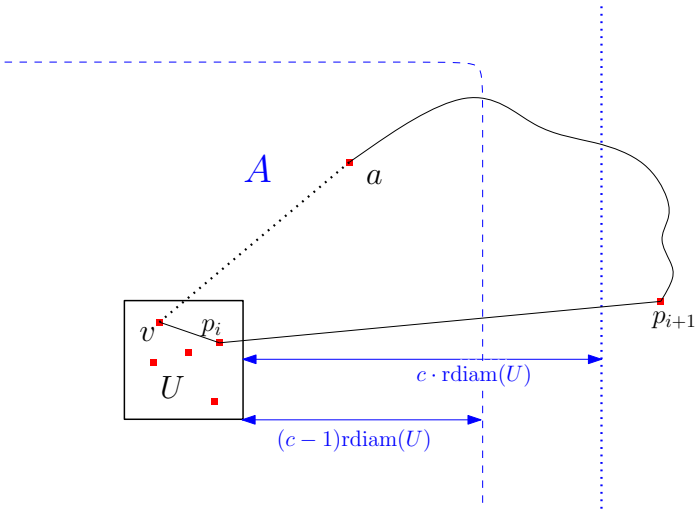

Figure 4.3: Assuming there is an edge in T (the black dotted line) that connects vertex a inside the blue dashed area A with a vertex in U leads to a contradiction.

Using this claim we can show the following lemma.

Lemma 4.3. Let $\mathcal{T}$ be a family of triangulations with constant dilation $d = \delta(\mathcal{T})$, $c_2 > d$ some constant and $T \in \mathcal{T}$ a triangulation of a planar point set S. Furthermore, let G be the minimum spanning tree of T and $U \subseteq S$ a subset of S such that $G[U]$ is connected. If every edge in $I_G(U)$ has length greater than $c_2 \cdot \text{rdiam}(U)$ then U is a c_1-cluster for $c_1 = 2(c2-1)/(d+1)$.

Proof. Let $A = \{b \in \mathbb{R}^2 \mid d_e(U,b) < (c_2-1)\text{rdiam}(U)\}$ be defined as the region where all points can lie that are less than $(c_2-1)\text{rdiam}(U)$ apart from U. Let $a \in S$ be a point within A. By Claim 4.2 we know that there is no edge $av \in E(T)$ with $v \in U$ because all edges in $I_G(U)$ have length greater than $(c_2-1)\text{rdiam}(U)$ and thus all edges in $I_T(U)$ must have length greater than $(c_2-1)\text{rdiam}(U)$. Furthermore the shortest path P from v to a in the triangulation T must use one edge of $I_T(U)$ and thus $d_T(v,a) > (c_2-1)$. But now P has left the region A and since $a \in A$ the path P must reenter A. Say h is the distance between the point where P reenters A to a (see Figure 4.4). Then by the triangle inequality it follows that $(c_2-1)\text{rdiam}(U) \leq h + d_e(v,a)$ and therefore $h \geq (c_2-1)\text{rdiam}(U) - d_e(v,a)$. Hence the total length of the shortest path between v and a must be at least

$$d_T(v,a) \geq (c_2-1)\text{rdiam}(U) + h$$

by plugging in the lower bound for h we have

$$d_T(v,a) \geq (c_2-1)\text{rdiam}(U) + (c_2-1)\text{rdiam}(U) - d_e(v,a)$$
$$d_T(v,a) \geq 2(c_2-1)\text{rdiam}(U) - d_e(v,a)$$

and since the triangulation T has dilation d it follows

$$d \cdot d_e(v,a) \geq 2(c_2-1)\text{rdiam}(U) - d_e(v,a)$$
$$(d+1) \cdot d_e(v,a) \geq 2(c_2-1)\text{rdiam}(U)$$
$$d_e(v,a) \geq (2(c_2-1)/(d+1))\text{rdiam}(U)$$

Thus, we have that every point not in U has distance at least $2(c_2-1)/(d+1)\text{rdiam}(U)$ from U and we can conclude that U is a c_1-cluster for $c_1 = 2(c_2-1)/(d+1)$. □

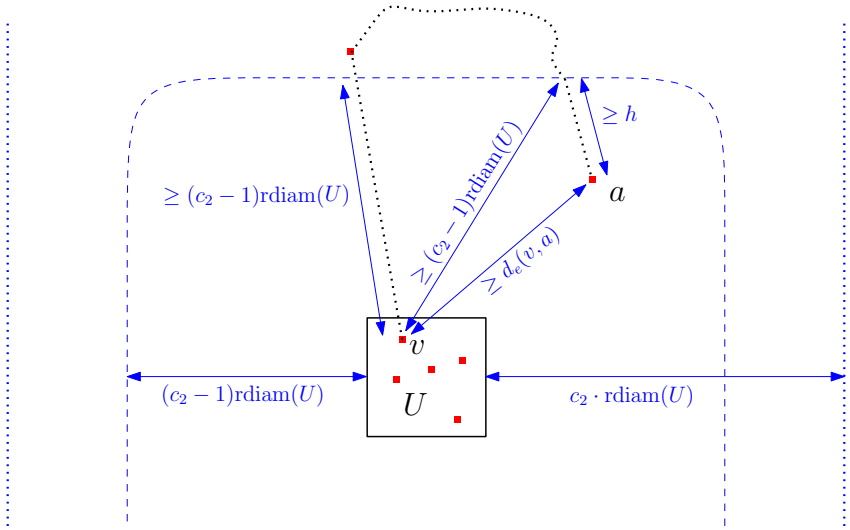

Figure 4.4: The length of the shortest path between any vertex $v \in U$ and a in G (black dotted line) must be at least $c_2 \cdot \text{rdiam}(U) + (c_2 \cdot \text{rdiam}(U) - d_e(v,a))$.

Now we can proof that the algorithm is indeed correct with the following theorem.

Theorem 4.4. Given a point set S and a triangulation on S with dilation d Algorithm 4.2.1 computes correctly a (c_1, c_2)-cluster tree for S with $c_1 = \frac{2c_2}{(c_2+1)}$ and some

constant $c_2 > d$.

Proof. Clearly Algorithm 4.2.1 produces some cluster tree T. We denote by C_v the cluster that is assigned to node v of T. By Lemma 4.3 we know that for all nodes v of T the cluster C_v is at least a c_1-cluster. The only point where the algorithm create some node of T and thus a cluster is at line 26 but at this point all conditions to apply Lemma 4.3 are fulfilled. It remains to show that every c_2-cluster is assigned to a node in T.

Assume some c_2-clusters are missing in T. Let depth(v) be the length of the path from a node v to the root of T. We choose the node v in T such that depth(v) is maximal and v has the following properties:

1. There is a c_2-cluster D with $D \subseteq C_v$.

2. There is no node u of T such that $C_u = D$.

In other words D is a c_2-cluster but not found by the algorithm and D should be assigned to some child of v. Now let $u_1, \ldots, u_m$ be the m children of v in T labeled in the way that $D = \bigcup_{i=1}^{l} C_{u_i}$ for some $2 \leq l < m$. Consider the point where Algorithm 4.2.1 computes C_v. Since we organized the computation of T such that all child clusters of C_v will be computed before C_v, we can assume that the clusters $C_{u_1}, \ldots, C_{u_m}$ are single vertices (i.e. they are already contracted by the algorithm to a shrunken vertex) of the minimum spanning tree G. Now we distinguish between two cases.

First assume that the computation of C_v starts from some u_i with $1 \leq i \leq l$, i.e. we start within D. The algorithm will maintain a subset $U \subseteq V(G)$ and augment U by vertices from G. Since D is a c_2-cluster it holds that $d_e(U, u_j) \leq \text{rdiam}(D)$ for $1 \leq j \leq l$ and $d_e(U, u_j) \geq c_2 \cdot \text{rdiam}(D)$ for $l+1 \leq j \leq m$. Because rdiam(U) is less than rdiam(D) the subset U will be first augmented by the vertices $u_1, \ldots, u_l$. But then all edges in $I_G(U)$ have length greater than $c_2\dot{\text{rdiam}}(U)$ and the algorithm will shrink U to a single vertex u' and assigning u' the cluster $C_{u'}$ (line 26). Hence we get that $C_{u'} = \bigcup_{i=1}^{l} C_{u_i}$ which is equal to D and thus the algorithm will find D.

Therefore, assume the second case where we start computing C_v from some u_i with $l + 1 \leq i \leq m$. Then U is augmented with vertices from $\{u_1, \ldots, u_m\}$. At some point we must add for the first time some u_j to U with $1 \leq j \leq l$. Since u_j is part of D and D is a c_2-cluster, the edge that is responsible that u_j will be added to U must be longer than $c_2 \cdot \text{rdiam}(D)$. But D contains at least two elements from $\{u_1, \ldots, u_l\}$ and we know by Lemma 4.1 that the by D induced subgraph in G is connected. Therefore, u_j must have an incident edge with length less then rdiam(D). Thus, the computation of C_v

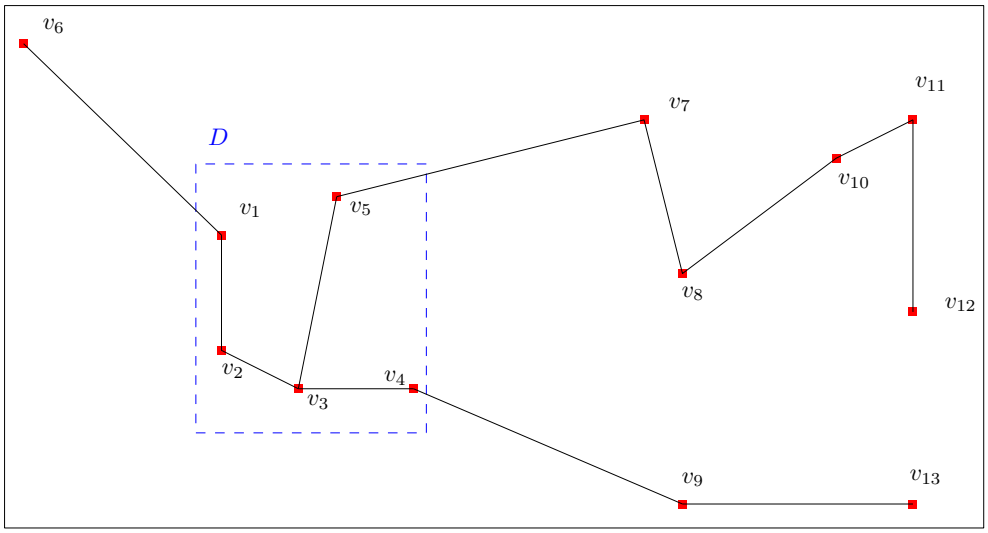

Figure 4.5: The vertices $v_1, \ldots, v_5$ belong to the c_2-cluster D that was not found by the algorithm.

will be suspended and we start computing child cluster of C_v from u_j (lines 15-18 of Algorithm 4.2.1). However, since u_j is contained in D we are now in the same situation as in the first case and thus D will be computed by the algorithm and assigned to a node in T as well.

Therefore, in every case D will be computed correctly and assigned to a child node of v. □

4.4 Running Time

We want to show that the running time of the algorithm is linear in the number of vertices of the triangulation. In order to do this we can show a similar property as stated in Lemma 3.3 in Section 3.1.3 and used by Krznaric and Levcopoulos to prove that their algorithm only needs linear time. We need to show that a spanning tree G of a triangulation with constant dilation has the properties that each vertex of G has constant degree and that for every subset $U \subseteq V(G)$ the number of edges in $I_G(U)$ is bounded by some constant if the subgraph induced by U is connected. Unfortunately these properties are not true.

Consider the following counterexample of a triangulation with constant dilation which results in a spanning tree G with a vertex that has got an arbitrary number of incident edges. Let w be a vertex and α some small angle, say $\alpha = \pi/6$. Furthermore, let

$v_1, \ldots, v_m$ be vertices that are adjacent to w in T and will be adjacent to w in G as well. We place the vertices $v_1, \ldots, v_m$ such that the angle $\angle v_i w v_{i+1}$ is $\frac{\alpha}{m}$ and the distance $d_e(w, v_i)$ is exactly 3^i (see Figure 4.6). We also need some edge that intersects the line

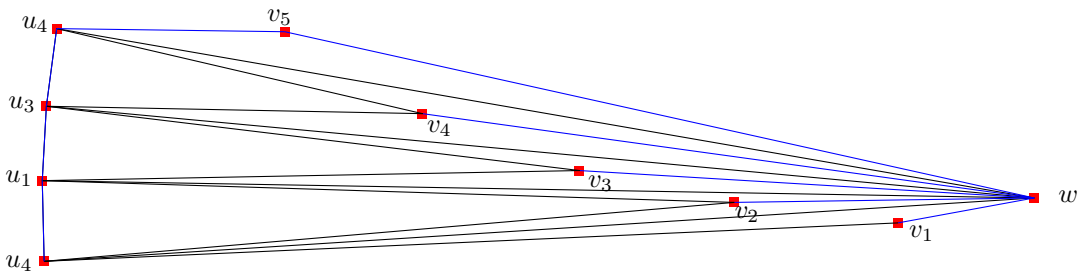

Figure 4.6: This example can be extended to result in a vertex w has arbitrary high degree in G.

segment $v_i v_{i+1}$, or otherwise this line would be an edge of the triangulation and in the minimum spanning tree as well. Therefore, the vertices $u_1, \ldots, u_{m-1}$ are placed on the circle with radius $r = 3^m$ and center w such that the line wu_i bisects the angle $\angle v_i w v_{i+1}$. We add the following edges to obtain a triangulation:

- wu_i for $1 \leq i \leq m-1$
- $v_i u_i$ and $v_{i+1} u_i$ for $1 \leq i \leq m-1$
- $u_i u_{i+1}$ for $1 \leq i \leq m-1$

By the triangle inequality we know that the edges $v_i u_i$ and $v_i u_{i+1}$ are longer than the edge wv_i. It holds that $d_e(w, u_i) \leq d_e(w, v_i) + d_e(v_i, u_i)$ or equivalent $d_e(v_i, u_i) \geq d_e(w, u_i) - d_e(w, v_i)$. Since $d_e(w, v_i)$ is at most 3^{m-1} and $d(w, u_i)$ is the radius r with $r = 3^m$ for all $1 \leq i \leq m-1$, we have that $d(v_i, u_i)$ is greater than the length of the edge wv_i. Thus, all edges wv_i will be contained in the minimum spanning tree of the constructed triangulation.

It remains to show that this triangulation has constant dilation. Indeed, this is the case since every vertex is adjacent to w and thus there is path with three vertices that connects every two distinct vertices using w as an intermediate vertex. We will first show that the dilation condition will hold for every two vertices v_j and v_i. Note that it is sufficient to show that the dilation condition will hold for every pair v_i and v_{i+1} because if we choose some v_j with $j < i$ the distance $d_e(v_j, v_{i+1})$ increase but the shortest path distance $d_T(v_j, v_{i+1})$ only decreases compared to $d_e(v_i, v_{i+1})$ and $d_T(v_i, v_{i+1})$ respectively. Since we have chosen that $d_e(w, v_i)$ is exactly $\frac{1}{3} d_e(w, v_{i+1})$ we have that by the triangle

inequality the distance $d_e(v_i, v_{i+1})$ between v_i and v_{i+1} is at least $\frac{2}{3}d_e(w, v_{i+1})$. The shortest path distance in the triangulation T between v_i and v_{i+1} is

$$d_T(v_i, v_{i+1}) = d_e(w, v_i) + d_e(w, v_{i+1}),$$

i.e. the distance from v_i to w together with the distance from w to v_{i+1}. Since the former is at most $\frac{1}{3}d_e(w, v_{i+1})$ we have

$$d_T(v_i, v_{i+1}) \leq \frac{4}{3}d_e(v_i, v_{i+1}).$$

Because of the fact that the distance between v_i and v_{i+1} is at least $\frac{2}{3}d_e(w, v_{i+1})$ the ratio between shortest path distance in the triangulation $d_T(v_i, v_{i+1})$ and the euclidean distance is between v_i and v_{i+1} is

$$\frac{d_T(v_i, v_{i+1})}{d_e(v_i, v_{i+1})} \leq \frac{\frac{4}{3}d_e(v_i, v_{i+1})}{\frac{2}{3}d_e(v_i, v_{i+1})} = 2.$$

For two vertices v_i and u_j we can use the same argument as above since they are connected by a path using w and for $i = m$ and $j = m-1$ the euclidean distance $d_e(v_m, u_{m-1})$ is minimal. But since $d_e(w, v_m) = 3^{m-1}$ and $d_e(w, u_{m-1}) = 3^m$ we have the same situation as above for arbitrary v_i and v_{i+1}. If we choose v_i and u_j with $i \neq m$ and $j \neq m-1$ the euclidean distance $d_e(v_i, u_j)$ can only increase and the length of the path (v_i, w, u_j) only decreases and therefore the detour between these two vertices will decrease.

All vertices u_i lie on a circle with radius $r = 3^m$. Let u_j and u_l be two vertices with $1 \leq j < k \leq m-1$. Then the angle β between wu_i and wu_j is $(k-j)\frac{\alpha}{m}$. But also due to the construction β must be between 0 and $\pi/6$. Now consider the isosceles triangle $u_j w u_k$. Using the definition of the sine function in a right-angled triangle we get for the distance between u_j and u_k

$$d_e(u_j, u_k) = 2r\sin(\beta/2).$$

One path from u_j to u_k in the triangulation is by using the edges $u_i u_{i+1}$ for $j \leq i < k$. We can bound the length of this path from above by using the length of the arc between u_j and u_k which is βr. Thus the ratio between the shortest path distance and the

euclidean distance for u_j and u_k is

$$\frac{d_T(u_j, u_k)}{d_e(u_j, u_k)} \leq \frac{\beta r}{2r\sin(\beta/2)} = \frac{\beta}{2\sin(\beta/2)}.$$

Now we can use the series expansion for the sine function to obtain

$$\frac{d_T(u_j, u_k)}{d_e(u_j, u_k)} \leq \frac{\beta}{2\sum_{n=0}^{\infty}(-1)^n(\beta/2)^{2n+1}/(2n+1)!}$$

and since the series is alternating and the angle β is less than 1 we can bound it from below by $(\beta/2) - (\beta/2)^3/3!$ to get

$$\begin{aligned}
\frac{d_T(u_j, u_k)}{d_e(u_j, u_k)} &\leq \frac{\beta}{2((\beta/2) - (\beta/2)^3/3!)} \\
&\leq \frac{\beta}{2((\beta/2) - (\beta/2)/6)} \\
&= \frac{\beta}{(5/6)\beta} \\
&\leq 2.
\end{aligned}$$

We can conclude that the detour between any two vertices of the constructed triangulation is at most 2 and therefore the triangulation has dilation at most 2.

Due to this counterexample we cannot prove the necessary lemma to show a linear running time. The problem is that for a spanning tree with vertices of arbitrary large degree the set P can have size $O(n)$ at each iteration with n being the size of the underlying point set. Since we need to find all edges in P with length less than $c_2 \cdot \text{rdiam}(U)$, every time a point is added to some cluster (see line 23 of Algorithm 4.2.1) the total running time will increase up to $O(n^2)$. Such a running time is inacceptable because we can use for example the Fortune Sweep algorithm to compute the Voronoi Diagram of the underlying point set S in time $O(n \log n)$ and then transform it to the Delaunay triangulation of S in time $O(n)$. Afterwards, the algorithm from Krznaric and Levcopoulos can be used to obtain a hierarchical clustering for S in a faster way. However, it is not known if the computation of a (c_1, c_2)-cluster tree from a triangulation with constant dilation can be speed up by either using a better analysis technique for Algorithm 4.2.1 or a different and faster algorithm to solve the problem.

4.5 Additional Properties of Triangulations with Constant Dilation

In this section we will state some properties of triangulations with constant dilation and their minimum spanning trees that may be useful for proving more complex results or designing algorithms for various problems related to this kind of triangulations.

The first property concerns the closest pair problem. Given a finite planar point set S with $|S| = n$ we want to find the two distinct points $p, q \in S$ that minimize the Euclidean distance $d_e(p, q)$. In the RAM model this problem can be solved efficiently in expected time $O(n)$ [GRSS95]. However, this algorithm relies on the floor function (and random access) and therefore it cannot be ported to the pointer machine model. In fact, it is known that the lower time bound for computing the closest pair of points of a finite point set is $\Omega(n \log n)$ in the algebraic decision tree model [BO83]. Therefore, it is useful to try to speed up the computation if a triangulation with constant dilation on the point set S is given as additional input. We show that if the triangulation has dilation less than 2, the closest pair $\{p, q\}$ must be connected by an edge of T.

Property 4.5. Let T be a triangulation on a planar point set S with dilation $\delta(T) < 2$. Let $\{p, q\}$ be the closest pair of points of S, then there is an edge pq in T.

Proof. Let pq be the shortest edge of T. Assume p and q are not the closest pair of S and let x and y be the closest pair. Clearly x and y cannot be connected by an edge in T or otherwise the distance between p and q would be less then the distance between x and y. But there is a shortest path P in T that connects x and y with length $d_T(x, y)$. Since T has dilation less than 2 it holds that

$$\frac{d_T(x,y)}{d_e(x,y)} < 2 \text{ or equivalently } d_e(x,y) > \frac{d_T(x,y)}{2}.$$

Because the shortest path between x and y must contain at least two edges and every edge has length at least $d_e(p, q)$ we get that

$$d_e(x,y) > \frac{d_T(x,y)}{2} \geq \frac{2d_e(p,q)}{2} = d_e(p,q),$$

which is a contradiction to the assumption that $\{x, y\}$ is the closest pair of S. □

It remains the question whether we can find the closest pair if the dilation of the triangulation is some constant greater than 2. The intuition is that since the dilation is bounded by some constant, we can find for every vertex $w \in S$ all vertices $v \in S \setminus \{w\}$

such that the pair $\{w, v\}$ is a candidate for the closest pair. To do this we use Dijkstra's algorithm starting from w. The following algorithm will find the closest pair for a triangulation T with dilation d.

Algorithm 4.5.1 Computing the closest pair from a triangulation T with dilation d.

```
1: Closest-pair(T,d)
2: Find the shortest edge xy in E(T) and let d_e(x, y) its length.
3: Delete all edges in E(T) with length greater than d · d_e(x, y).
4: Closest pair {p, q} ← {x, y}
5: for all w ∈ V(T) do
6:     Use Dijkstra's algorithm to find the set of vertices C that are connected with w
   through a path of length at most d · d_e(x, y).
7:     for all v ∈ C do
8:         if d_e(w, v) < d_e(p, q) then
9:             {p, q} ← {w, v}
10:        end if
11:    end for
12: end for
13: return {p, q}
```

The following theorem states that Algorithm 4.5.1 computes correctly the closest pair of a point set in linear time, if given a triangulation on that point set with bounded dilation.

Theorem 4.6. Let T be a triangulation with dilation d on a planar point set S with $|S| = n$. Algorithm 4.5.1 computes the closest pair of S in time $O(n)$.

Proof. Let $xy \in E(T)$ be the shortest edge of the triangulation T. If the pair $\{x, y\}$ is not the shortest pair of S there must be a pair $\{p, q\}$ with $d_e(p, q) < d_e(x, y)$. Since the triangulation has dilation d, there must be a path of length at most $d \cdot d_e(p, q)$ that connects p and q and we get

$$d_T(p, q) \leq d \cdot d_e(p, q) \leq d \cdot d_e(x, y).$$

Therefore, we can remove all edges with length greater than $d \cdot d_e(x, y)$ from T because they cannot be an edge of the path between p and q. Since we check in the for-loop in lines 5-12 all pairs of vertices $\{v, w\}$ that are connected by a path of length at most $d \cdot d_e(x, y)$, the algorithm finds correctly the closest pair.

It remains to show that the running time is linear. Since we remove all edges with length greater than $d \cdot d_e(x, y)$, there are only edges left with length between $d_e(x, y)$ and $d \cdot d_e(x, y)$. We claim that every vertex has only a constant number of incident edges with such length. Since T has dilation d and xy is the shortest edge, we know that for the shortest pair $\{p, q\}$ it must hold that $d_e(p, q)$ is greater than $d_e(x, y)/d$. Thus, there is an exclusion disc with radius $d_e(x, y)/d$ around each vertex $v \in S$ such that no other point of S can lie within this disc.

Now let $w \in S$ be some vertex. Consider the annulus A with center w, inner radius $d_e(x, y)$ and outer radius $d \cdot d_e(x, y)$ and let $S_w \subset S$ the set of all vertices that lie within A. We show that since each vertex in S_w has an exclusion disc with radius $d_e(x, y)/d$, we can place only a constant number of vertices in A. Consider for every vertex in $v \in S_w$ the disc with D_v with center v and radius $d_e(x, y)/(2d)$, i.e. the radius of D_v is exactly half the radius of v's exclusion disc. Note that for every two distinct vertices v, v' of S_w the discs D_v and $D_{v'}$ will not intersect because v and v' have at least distance $d_e(x, y)/d$. Furthermore, consider a slightly larger annulus A' with inner radius $d_e(x, y) - d_e(x, y)/(2d)$ and outer radius $d \cdot d_e(x, v) + d_e(x, y)/(2d)$ and note that for each vertex v the disc D_v lies completely within A'. The area of A' is

$$\begin{aligned}
\text{area}(A') &= \pi((d \cdot d_e(x, v) + \frac{d_e(x, y)}{2d})^2 - (d_e(x, y) - \frac{d_e(x, y)}{2d})^2) \\
&= \pi d_e(x, v)^2((d + \frac{1}{2d})^2 - (1 - \frac{1}{2d})^2) \\
&= \pi d_e(x, v)^2(d^2 + \frac{1}{d})
\end{aligned}$$

and the area that a disc D_v for some vertex $v \in S_w$ covers is

$$\text{area}(D_v) = \pi d_e(x, v)^2/d^2.$$

Thus, we can place at most a constant number, namely

$$\frac{\text{area}(A')}{\text{area}(D_v)} = d^4 + d$$

many vertices v with disc D_v within A' and, since A is smaller than A', within A, too.

Finally, note that every path of length $d \cdot d_e(x, y)$ has at most $\lfloor d \rfloor$ edges since the shortest

edge of T has length $d_e(x, y)$. Thus, we can stop Dijkstra's algorithm in line 6 after all vertices that are connected to w by a path with at most d edges are found. The tree computed by Dijkstra's algorithm has depth at most d and because all vertices of the tree have constant degree, it has constant many nodes. The critical data-structure used by Dijkstra's algorithm is a priority queue. However, since the tree has a constant number of leaf vertices and each leaf vertex has constant degree, there can be at most a constant number of vertices in the priority queue at any time. Thus, all priority queue operations take constant time.

Since Dijkstra's algorithm will find a constant number of vertices, lines 7-12 will take constant time as well and the total running time will be $O(n)$. □

The second property is related to the minimum spanning tree G of a triangulation with constant dilation. We show that for any vertex w of G the vertices that are adjacent to w in G have a circular exclusion region that depends on the length of the edge that connects w with its neighbor such that within the exclusion region of some neighbor v of w there can be no other neighbor v' of w.

Property 4.7. Let $\mathcal{T}$ be a family of triangulations with constant dilation $d = \delta(\mathcal{T})$ and $T \in \mathcal{T}$ a triangulation on a planar point set S with minimum spanning tree G. Let w be an arbitrary vertex of G and v a vertex adjacent to w in G. Then there is a disc D with radius $d_e(w, v)/d$ and v as center such that no other neighbor v' of w in G can lie within D.

Proof. Let v be some vertex adjacent to w in G. We define the disc D as the disc with center v and radius $d_e(w, v)/d$. Assume there is a vertex v' that is adjacent to w and lies within D. Then the distance $d_e(v, v')$ between v and v' is less than $d_e(w, v)/d$. But since the triangulation has dilation d, there is a path $P = (v = p_1, \ldots, p_l = v')$ from v to v' of length at most $d \cdot d_e(v, v')$. Using that $d_e(v, v')$ is less than $d_e(w, v)/d$ we get that the length of P is less than $d \cdot d_e(w, v)/d = d_e(w, v)$. Therefore all edges in P have length less than $d_e(w, v)$. Removing the edge wv from G splits G into two parts. Now we can find an edge $p_i p_{i+1}$ in P such that adding $p_i p_{i+1}$ reconnects these two parts to a new tree, say G'. But since the length of $p_i p_{i+1}$ is less than the length of wv the weight of G' is less than the weight of G which contradicts the minimality of G. □

Property 4.7 can be used to prove that for any vertex v within an annulus with constant size that has w as center there can be only a constant number of vertices that are adjacent to w in G. The proof is similar to the proof of Theorem 4.6.

Property 4.8. Let G be the minimum spanning tree of a triangulation with dilation d and w some vertex of G. Furthermore, let r_1 and r_2 be two constants independent of the number of vertices of G with $0 < r_1 < r_2$. Then there are only a constant number of vertices v of G with $r_1 \leq d_e(w, v) \leq r_2$ that are adjacent to w in G.

Proof. Let A be the annulus centered at w with inner radius r_1 and outer radius r_2 and let $S_w \subset S$ be the set of vertices that lie within A and are adjacent to w in G. The area that A covers is $\pi(r_2^2 - r_1^2)$ which is some constant since r_1 and r_2 are constants. By Property 4.7 we know that each vertex that lies within A has a circular exclusion region with radius at least r_1/d. The idea is that we can only place a constant number of vertices within A until their exclusion regions cover the whole area of A.

Consider for each vertex $v \in S_w$ the disc D_v with center v and radius $r_1/2d$, i.e. the radius of D_v is exactly one half the radius of the circular exclusion region of v. Note that for every two vertices v, v' of S_w the discs D_v and $D_{v'}$ do not intersect. Now we define a slightly larger annulus A' with center w, inner radius $r_1' = r_1 - r_1/(2d)$ and outer radius $r_2' = r_2 + r_1/(2d)$. Observe that for each vertex $v \in S_w$ the disc D_v lies completely in A'. Since the area of A' is given by

$$\begin{aligned} \text{area}(A') &= \pi((r_2')^2 - (r_1')^2) \\ &= \pi((r_2 + r_1/(2d))^2 - (r_1 - r_1/(2d))^2) \end{aligned}$$

and since r_1, r_2 and d are all constants, the area will be some constant as well. Furthermore, for each vertex $v \in S_w$ the disc D_v covers at least

$$\text{area}(D_v) = \pi r_1^2/(4d^2)$$

of A which is some constant, too. Thus, we can place only a constant number of vertices in the annulus A'. Since A is smaller than A' this will also hold for A and we can conclude that there can be only a constant number of vertices in S_w and therefore only a constant number of vertices that are adjacent to w in G. □

Property 4.8 enables us to argue that certain operations can be done faster for special point sets S. For instance, if we have given a triangulation T with constant dilation and the spread of the underlying point set S is bounded by, say $O(n)$, where $n = |S|$, it can be shown that every vertex of the minimum spanning tree G of T has degree at most $O(\log n)$. Therefore, let w be a vertex of G and let x the length of the shortest edge incident to w in G. We place $O(\log n)$ annuli around w with annulus A_i having inner radius $2^{i-1}x$ and outer radius $2^i x$. Note that since the spread is bounded by $O(n)$

we will cover all points with these annuli. By Property 4.8 each annulus contains at constant number of vertices adjacent to w and thus there are at most $O(\log n)$ many vertices adjacent to w in G. We can also show that a similar result hold for connected subgraphs of G by extending the proof of Property 4.8 to those graphs instead of single vertices.

To obtain a speedup for Algorithm 4.2.1 we replace the sets P and Q with a priority queue to maintain the edges incident to a partially computed cluster. If the spread is bounded by $O(n)$, the priority queue do not contain more than $O(\log n)$ edges at each step of the algorithm. Thus, each priority queue operation can be done in time $O(\log \log n)$ and since every edge will placed at most 2 times in some priority queue, the running time of the algorithm will be $O(n \log \log n)$. However, since the main goal of cluster trees is to build up a hierarchical clustering such that within each cluster the spread between the child clusters is bounded (cmp. Section 2.4.2), it is not a valid assumption that the spread of the point set S is bounded.

5 Summary

5.1 Conclusion

In Chapter 2 we defined basic graph theoretic notations with special interest to planar graphs and geometric triangulations. Furthermore, we defined the dilation of a planar graph, the property that was mainly used for the proofs in Chapter 4. The main goal of this thesis was to efficiently compute a c-cluster tree from a triangulation with constant dilation. Thus, we also defined c-cluster and c-cluster trees and introduced a relaxed version of these trees, the (c_1, c_2)-cluster trees. (c_1, c_2)-cluster trees have a less stricter definition than the c-cluster trees and therefore allow more flexibilty and expose more potentialities to compute them. It turned out that (c_1, c_2)-cluster trees have some important properties in common with c-cluster trees and that both are similar structures in the sense that we can transform any (c_1, c_2)-cluster tree into a c-cluster tree for $c = c_2$ (see Section 2.4.2). Since the computation of c-cluster trees and (c_1, c_2)-cluster trees from triangulations strongly build on minimum spanning trees of the triangulations, we showed in Section 2.6 how to efficiently compute minimum spanning trees using the method of Borůvka. For the class of planar graphs (where geometric triangulations belong to) this can even be done in time linear in the number of vertices of the graph.

In Chapter 3 we gave a description of an algorithm developed by Krznaric and Levcopoulos that, given the Delaunay triangulation, computes a c-cluster tree for the underlying point set S in linear time. It uses the fact that the Euclidean minimum spanning tree of S is equivalent to the spanning tree of the triangulation. The algorithm starts by extracting the minimum spanning tree from the triangulation and then walks along the the edges of the spanning tree to construct a c-cluster tree. Afterwards, we mentioned some applications of c-cluster trees in algorithmic geometry and cluster analysis and explained at a concrete example how it can be used to efficiently approximate the single-linkage clustering.

In the next chapter it was examined if the ideas from Chapter 3 can be adapted and extended to work with triangulations that only have constant dilation instead of requiring the Delaunay triangulation. Therefore, we first showed that there is a connection between

the minimum spanning tree of a triangulation with constant dilation and c-clusters by proving that a c-cluster corresponds to a connected subgraph of the minimum spanning tree. Afterwards, it was shown that some geometric properties of spanning trees of triangulations with constant dilation are similar to properties of the Euclidean minimum spanning tree (see. Section 4.3). However, these properties are not as strict as they are for the Euclidean minimum spanning tree but at this point the flexibility of the (c_1, c_2)-cluster tree, the relaxed version of the c-cluster tree, helped out. We gave an algorithm that computes a (c_1, c_2)-cluster tree from a given triangulation with constant dilation and proved its correctness. While analyzing the running time of the algorithm it turned out that there are properties of the Euclidean minimum spanning tree that are not to be true for minimum spanning trees of triangulations with constant dilation. More precisely, every vertex of the Euclidean minimum spanning tree has a constant degree, but this is not the case for general minimum spanning trees. Therefore, a counterexample for a triangulation with constant dilation was given that will result in a minimum spanning tree with a vertex of arbitrary high degree.

At the end we stated different properties of triangulations with constant dilation that were encountered during the research but could not be used in any particular proof. This includes an algorithm for finding the closest pair of a planar point set S in linear time if a triangulation with constant dilation on S is known.

5.2 Open Problems

The most interesting open question may be if we can compute from a given triangulation with constant dilation a c-cluster tree or a (c_1, c_2)-cluster tree in linear time. It is desired that such an algorithm can be implemented on a pointer machine but an algorithm for the stronger RAM model would also be a good result. The major difficulty is to achieve a linear running time. Since there are not many computational methods and algorithm design techniques known to fit in these tight time constraints, it is unlikely that there is a completely different algorithm that computes an (c_1, c_2)-cluster tree than the one presented in Chapter 4. Therefore, it is probably more promising to further restrict the input in order to get linear running time with the presented algorithm.

This leads to the second open question: Is there a property that we can add as additional requirement for the triangulation except for constant dilation such that the running time of Algorithm 4.2.1 is linear. This additional property must ensure that the degree of every vertex of the minimum spanning tree G of the triangulation is bounded by some constant and that for every connected subgraph H of G the number of edges

incident to exactly one vertex of H, i.e. the size of the set $I_G(V(H))$, is bounded by some constant, too. In other words analogue versions of Property 3.2 and Lemma 3.3 must hold for the minimum spanning tree of the input triangulation. This would imply that the running time of the Algorithm 4.2.1 is linear and the analysis would be analogue to the one for the algorithm from Krznaric and Levcopoulos from Chapter 3. One promising candidate for this additional property is the diamond property. On the one hand the counterexample of Section 4.4 contradicts the diamond property and it cannot be easily modified to fulfill the diamond and be a valid counterexample at the same time. On the other hand the diamond property is closely related to the dilation of a triangulation in the sense that by Theorem 2.6 every triangulation that posses the diamond property has constant dilation.

The object of research of this thesis was to develop fast algorithms by exploiting the structures of triangulations with constant dilation. However, it turned out that all results do not be in need of the fact that the input graph is actually a triangulation. We only need some planar embedding of a graph on a point set S with all edges being line segments, called planar straight line graph, such that the dilation of this graph is some constant. Thus, we can rephrase the above question in a more general way and ask if there is a class of planar straight line graphs instead of triangulations such that their spanning trees will fulfill the necessary properties to obtain a linear running time for Algorithm 4.2.1.

Bibliography

[AUH83] Alfred V. Aho, Jeffrey D. Ullman, and John E. Hopcroft. *Data structures and algorithms*. Addison-Wesley, Reading, Mass. :, 1983.

[BDL+10] Prosenjit Bose, Luc Devroye, Maarten Löffler, Jack Snoeyink, and Vishal Verma. The dilation of the delaunay triangulation is greater than $\pi/2$. *CoRR*, abs/1006.0291, 2010.

[BE95] Marshall Wayne Bern and David Eppstein. Mesh generation and optimal triangulation. In Ding-Zhu Du and Frank Kwang-Ming Hwang, editors, *Computing in Euclidean Geometry*, number 4 in Lecture Notes Series on Computing, pages 47–123. World Scientific, second edition, 1995.

[BKRW02] Adam L. Buchsbaum, Haim Kaplan, Anne Rogers, and Jeffery Westbrook. Linear-time pointer-machine algorithms for least common ancestors, mst verification, and dominators. *CoRR*, cs.DS/0207061, 2002.

[BO83] Michael Ben-Or. Lower bounds for algebraic computation trees. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, STOC '83, pages 80–86, New York, NY, USA, 1983. ACM.

[Bor26] Otakar Boruvka. O Jistém Problému Minimálním (About a Certain Minimal Problem) (in Czech, German summary). *Práce Mor. Prírodoved. Spol. v Brne III*, 3, 1926.

[Che86] P Chew. There is a planar graph almost as good as the complete graph. In *Proceedings of the second annual symposium on Computational geometry*, SCG '86, pages 169–177, New York, NY, USA, 1986. ACM.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2 edition, 2001.

[CT76] D. Cheriton and R. Tarjan. Finding minimum spanning trees. *SIAM Journal on Computing*, 5(4):724–742, 1976.

[dBCvKO00] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer, 2nd edition, April 2000.

[DDMW94] Matthew T. Dickerson, Robert L. Scot Drysdale, Scott A. McElfresh, and Emo Welzl. Fast greedy triangulation algorithms. In *Proc. 10th Ann. Symp. Computational Geometry*, pages 211–220, 1994.

[Def77] D. Defays. An efficient algorithm for a complete link method. *The Computer Journal*, 20(4):364–366, 1977.

[DFS90] David P. Dobkin, Steven J. Friedman, and Kenneth J. Supowit. Delaunay graphs are almost as good as complete graphs. *Discrete Comput. Geom.*, 5(4):399–407, May 1990.

[Die06] Reinhard Diestel. *Graphentheorie*. Springer Verlag, 2006.

[DJ89] Gautam Das and Deborah Joseph. Which triangulations approximate the complete graph? In Hristo Djidjev, editor, *Optimal Algorithms*, volume 401 of *Lecture Notes in Computer Science*, pages 168–192. Springer Berlin / Heidelberg, 1989.

[ELL01] Brian S. Everitt, Sabine Landau, and Morven Leese. *Cluster Analysis*. Wiley, 4th edition, January 2001.

[Fár48] István Fáry. On straight line representation of planar graphs. *Acta Univ. Szeged. Sect. Sci. Math.*, 11:229–233, 1948.

[FB74] R. A. Finkel and J. L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta Informatica*, 4:1–9, 1974. 10.1007/BF00288933.

[GRSS95] Mordecai Golin, Rajeev Raman, Christian Schwarz, and Michiel Smid. Simple randomized algorithms for closest pair problems. *Nordic J. of Computing*, 2(1):3–27, March 1995.

[HKP12] Jiawei Han, Micheline Kamber, and Jian Pei. 10 - cluster analysis: Basic concepts and methods. In *Data Mining (Third Edition)*, pages 443 – 495. Morgan Kaufmann, Boston, third edition edition, 2012.

[JD88] Anil K. Jain and Richard C. Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[JvR71] N. Jardine and Cornelis Joost van Rijsbergen. The use of hierarchic clustering in information retrieval. *Information Storage and Retrieval*, 7(5):217–240, 1971.

[KG92] J. Keil and Carl Gutwin. Classes of graphs which approximate the complete euclidean graph. *Discrete & Computational Geometry*, 7:13–28, 1992. 10.1007/BF02187821.

[KL95a] Drago Krznaric and Christos Levcopoulos. Computing hierarchies of clusters from the euclidean minimum spanning tree in linear time. In P. Thiagarajan, editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 1026. Springer Berlin / Heidelberg, 1995.

[KL95b] Drago Krznaric and Christos Levcopoulos. The first subquadratic algorithm for complete linkage clustering. In John Staples, Peter Eades, Naoki Katoh, and Alistair Moffat, editors, *Algorithms and Computations*, volume 1004 of *Lecture Notes in Computer Science*, pages 392–401. Springer Berlin / Heidelberg, 1995. 10.1007/BFb0015445.

[KL98] Drago Krznaric and Christos Levcopoulos. Computing a threaded quadtree from the delaunay triangulation in linear time. *Nordic J. of Computing*, 5(1):1–18, March 1998.

[Kru56] Jr. Kruskal, Joseph B. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):pp. 48–50, 1956.

[LM12] Maarten Löffler and Wolfgang Mulzer. Triangulating the square and squaring the triangle: Quadtrees and delaunay triangulations are equivalent. *CoRR*, abs/1205.4738, 2012.

[Mar04] Martin Mareš. Two Linear Time Algorithms for MST on Minor Closed Graph Classes. *Archivum mathematicum*, 40(3):315–320, 2004.

[Mat94] Tomomi Matsui. A linear time algorithm for the minimum spanning tree problem on a planar graph, 1994.

[MR08] Wolfgang Mulzer and Günter Rote. Minimum-weight triangulation is np-hard. *J. ACM*, 55(2):11:1–11:29, May 2008.

[PS83] G. Punj and D. W. Stewart. Cluster Analysis in Marketing Research: Review and Suggestions for Applications. *Journal of Marketing Research*, 20(2):134–148, 1983.

[Sch79] Arnold Schönhage. On the power of random access machines. In Hermann Maurer, editor, *Automata, Languages and Programming*, volume 71 of *Lecture Notes in Computer Science*, pages 520–529. Springer Berlin / Heidelberg, 1979.

[Sch80] A. Schönhage. Storage modification machines. *SIAM Journal on Computing*, 9(3):490–508, 1980.

[SD95] Peter Su and Robert L. Scot Drysdale. A comparison of sequential delaunay triangulation algorithms. In *Proceedings of the eleventh annual symposium on Computational geometry*, SCG '95, pages 61–70, New York, NY, USA, 1995. ACM.

[Sib73] R. Sibson. Slink: An optimally efficient algorithm for the single-link cluster method. *The Computer Journal*, 16(1):30–34, 1973.

[SW88] Hanan Samet and Robert E. Webber. Hierarchical data structures and algorithms for computer graphics. part i. *IEEE Comput. Graph. Appl.*, 8(3):48–68, May 1988.

[Tar75] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, April 1975.

[Wag37] K. Wagner. Über eine eigenschaft der ebenen komplexe. *Mathematische Annalen*, 114:570–590, 1937. 10.1007/BF01594196.

[Xia11] Ge Xia. Improved upper bound on the stretch factor of delaunay triangulations. In *Proceedings of the 27th annual ACM symposium on Computational geometry*, SoCG '11, pages 264–273, New York, NY, USA, 2011. ACM.