Diploma Thesis

Minimum Stretch Spanning Trees

Philipp Boksberger

supervised by

Prof. Dr. Roger Wattenhofer
Fabian Kuhn

Institute for Pervasive Computing
Distributed Computing Group
ETH Zürich, Switzerland

July 18, 2003

Abstract

Spanning trees have always been of great interest in various areas of computer science. The same is true for the idea of shortest paths in a graph. *Minimum Stretch Spanning Trees* can be described as a combination of these two concepts. On the one hand they are spanning trees, on the other hand they have a minimum stretch which means that the distances between the nodes in the spanning tree remain as short as possible. We can talk about detours between endpoints of edges, which are introduced by removing these edges from the original graph in order to obtain a *Minimum Stretch Spanning Tree*. A spanning tree with a minimum stretch is one where the largest of these detours is minimal. Computing a *Minimum Stretch Spanning Tree* is known to be NP-hard for general graphs. This leads us to two simplifications of the problem. First, we restrict the input graphs to special graph families such as grids, grid subgraphs and unit disk graphs. Second, our main interest is in an approximation of the problem and not in the optimal solution. We present an algorithm that computes a spanning tree with stretch $O(OPT^4)$ in time $O(n \log n)$. Besides this we show a greedy and an evolutionary algorithm and prove that they do not produce a spanning tree with stretch better than $O(n)$. At last we present two algorithms for which, in the worst example found, the resulting spanning trees have stretch $\Omega(OPT^2)$ but it remains open how good the approximation factors of these algorithms are.

Contents

Chapter 1

Introduction

Finding a subgraph that somehow represents its original graph is a well known problem that arises quite often in the area of distributed computing. Examples are routing algorithms or protocols to share a common variable. The goal is, however, that these subgraphs approximate the original graphs in a good way. As the title *Minimum Stretch Spanning Trees* implies, the subgraphs we will deal with are spanning trees. They are obviously the smallest connected subgraphs possible, containing all of the original graph's vertices. These trees are important for many algorithms that need a graph's spanning tree to work on it. But how can be guaranteed that such a spanning tree is a good approximation of the original graph? We need to introduce a measure that enables us to assign a characteristic value to a spanning tree. This value shall tell us how good the tree approximates a given graph. One possibility for such a measure that characterizes the approximation quality of a subgraph is the stretch of the subgraph or tree. The notion of stretch was established by Peleg and Ullman when they introduced tree spanners in [17]. They defined a tree t-spanner T of a graph G to be a spanning tree in which the distance between every pair of vertices is at most t times their distance in G. The key idea behind the notion of tree spanners is the approximation of pairwise vertex-to-vertex distances in the original graph by spanning trees. The quality of the distance approximation is measured by the parameter t, which is referred to as the stretch factor.

Some applications need a tree spanner whose stretch is as small as possible because the size of the stretch has a direct effect on the time complexity of the algorithm. An example for such an application is the arrow protocol [6, 9]. The arrow protocol is a distributed queuing protocol which is based on path reversal in a network spanning tree. The protocol runs on a fixed, preselected spanning tree T of the network graph G. The small size of this spanning tree's stretch is crucial to the worst case competitive ratio of the arrow protocol which is s in the sequential case[1] and $s \log r$ under concurrent access[2], where s is the stretch of the preselected spanning tree in the network and r is the number of simultaneous requests in [9].

1.1 The Problem

In order to formalize the problem of *Minimum Stretch Spanning Trees*, we need some basic definitions regarding graphs. Since we are concerned mainly with unweighted graphs, we need a measure that enables us to talk about distances. So, given an unweighted graph $G = (V, E)$, and given a path[3] $[v_0, v_1, \ldots, v_k]$ between two vertices $v_0, v_k \in V$, the length of the path is k. This means, the length of a path is equal to the number of edges or hops in the path. The distance $dist_G(u, v)$ between two vertices $u, v \in V$ is the length of the shortest path between u and v.

Provided with this measure for distances in unweighted graphs we are ready to define the stretch of an edge e in the graph as the length of the shortest path, i.e. the distance between the endpoints of e in $G \setminus e$.[4] See Figure 1.1 for an example.

The next step is to define the stretch of a spanning tree $T = (V, \hat{E})$ in an unweighted graph $G = (V, E)$. Here we focus on the edges which are not in the spanning tree. The stretch of a spanning tree of G is defined as the maximum stretch of an edge $e \in E \setminus \hat{E}$ in T. Note that the path between the endpoints must lie completely in T. The stretch of the spanning trees in Figure 1.2 is 7 and 4, respectively.

This leads us to a notion for the stretch of an unweighted graph, which is the stretch of its spanning tree with minimum stretch. The stretch of the graph in Figure 1.1 is 4 because this is the stretch of the spanning tree with

[1]The sequential case is described in the original paper about the Arrow distributed queuing protocol [6].

[2]The competitive analysis of the Arrow distributed queuing protocol under concurrent access can be found in [9].

[3]A path in $G = (V, E)$ is an ordered list of vertices $[v_0, v_1, \ldots, v_k], v_i \in V$ for $0 \leq i \leq k$ with edges $\{\{v_i, v_i + i\} | 0 \leq i \leq k\} \subseteq E$ such that $v_i \neq v_j$ for every $i \neq j$.

[4]$G \setminus e$ means G without the edge e. More formally: Given a graph $G = (V, E)$, $G \setminus e$ denotes the graph $\hat{G} = (V, \hat{E})$ where $\hat{E} = E \setminus e$.

Figure 1.1: Graph $G = (V, E)$ with $e \in E$. $Stretch_G(e) = 4$.

Figure 1.2: $Stretch_G(T_1) = 7$ and $Stretch_G(T_2) = 4$.

minimum stretch (see Figure 1.2). The following definition summarizes the notion of stretch in the context of unweighted graphs, as suggested in [16].

Definition 1.1.1 (Stretch) *Let $G = (V, E)$ be an unweighted graph, let $\{u, v\} \in E$ be an edge of G and let $T = (V, \hat{E})$ be a spanning tree of G.*

- *The stretch of $\{u, v\}$ is defined as*

$$Stretch_G(\{u, v\}) = dist_{G \setminus \{u,v\}}(u, v).$$

- *The stretch of T in G is defined as*

$$Stretch_G(T) = \max_{e \in E \setminus \hat{E}} \{Stretch_T(e)\}.$$

- *The stretch of G is defined as*

$$Stretch(G) = \min_{T} \{Stretch_G(T)\}$$

According to this notation, the problem of finding a *Minimum Stretch Spanning Tree* can be described as:

Definition 1.1.2 (MSST) *Given an unweighted graph G, the* Minimum Stretch Spanning Tree (MSST) *problem asks for finding a spanning tree T minimizing $Stretch_G(T)$.*

The MSST problem has been introduced by Cai and Corniel in [5]. They showed that determining the existence of a tree t-spanner in unweighted graphs is NP-complete for any fixed $t \geq 4$. Therefore, MSST is known to be NP-hard. This leads us to two simplifications of the problem. First, we restrict the input graphs to special graph families such as grids, grid subgraphs and unit disk graphs. Second, our main interest is in an approximation of the problem and not in the optimal solution.

1.2 Notations and Definitions

We will use the terminology of Bondy and Murty [2]. Graphs in this paper can be either weighted or unweighted and are undirected. They are connected graphs without loops and multi edges. For any graph G, $V(G)$ denotes the vertex set of G, $E(G)$ denotes the edge set of G and if G is planar, $F(G)$ denotes the set of faces of a planar embedding of G.

Since regular grids and grid subgraphs are the fundamental graph families that we will deal with, let us give a definition of them here. We will use the following notation for regular grids and grid subgraphs.

Definition 1.2.1 (Regular Grid) *A regular* $(n \times m)$-grid *is a graph $G = (V, E)$, where every vertex has coordinates from $\{1, \dots n\} \times \{1, \dots m\}$. The set of edges of G is defined as*

$$E = \{\{v_{i,j}, v_{i',j'}\} : v_{i,j}, v_{i',j'} \in V, |i - i'| + |j - j'| = 1\}$$

Given a grid $G = (V, E)$, a grid subgraph $\hat{G} = (\hat{V}, \hat{E})$ has a vertex set $\hat{V} \subseteq V$ and an edge set $\hat{E} \subseteq E$, i.e. it is a grid graph where some vertices and/or edges are missing.

Figure 1.3: A regular (4×6)-grid and a grid subgraph.

1.3 Related Work

As mentioned above, the first time spanners were mentioned was in the context of practical motivations from communication networks. See Peleg and Ullman [17], who introduced spanners to synchronize asynchronous networks. Various kinds of spanners have been analyzed in order of choosing subnetworks with good properties. See [12, 17] for some of the variants. A survey of some results about spanners is presented in [18]. In most applications, the sparseness of a spanner is crucial. The sparsest spanners possible are spanning trees. Therefore the problem of finding t-spanners with a minimum number of edges leads naturally to the notion of tree t-spanners.

Cai [3, 4] and Cai and Corneil [5] showed that the problem of deciding the existence of a tree t-spanner in an unweighted graph G can be solved in polynomial time for $t = 2$, but that the problem is NP-complete for any $t \geq 4$. The case $t = 3$ is still open, but it was conjectured in [5] to be NP-complete. Peleg an Reshef [16] expanded this result. They showed, that the problem cannot be approximated by a factor better than $(1 + \sqrt{5})/2$ unless $P = NP$. In [20], Venkatesan et al. have proven the NP-hardness of the problem even for restricted, unweighted graph classes such as chordal, split, bipartite, or degree-constraint graphs.

Fekete and Kremer [8] focused on the case of planar graphs. They showed that determining the optimal stretch is also NP-hard for unweighted planar graphs. On the other hand they gave a polynomial time algorithm for any fixed t that for any given planar unweighted graph G with bounded face-length, decides whether G has a spanning tree with stretch t.

1.4 Overview and Results

The main result, which is presented in Chapter 4, is an algorithm that works on grid subgraphs or unit disk graphs and approximates a *Minimum Stretch Spanning Tree* with stretch in $O(OPT^4)$.[5] This is a theoretic bound that is only reached by very specific graphs. Chapter 2 is devoted to several lower bounds in connection with the stretch of graphs. Some of the algorithms presented later build up on these lower bounds, especially the OPT^4-algorithm in Chapter 4. In Chapter 3 we discuss a polynomial time algorithm that computes *Minimum Stretch Spanning Trees* for regular $(n \times m)$-grids. It is also discussed there, how this algorithm applies to grid subgraphs. In Chapter 5 we present an abortive evolutionary approach to solve the problem of finding a *Minimum Stretch Spanning Tree.* Abortive, because it did not work as we expected in the beginning. An uncomplex and interesting greedy algorithm is presented in Chapter 6. Unfortunately we cannot give an upper bound for the stretch this algorithm produces. The only thing we can show is that there are examples, where the stretch produced by the algorithm is in $\Omega(OPT^2)$. No worse examples have been found.

[5]While the optimum *Minimum Stretch Spanning Tree* has stretch OPT.

Chapter 2

Considerations of Lower Bounds

Dealing with approximation algorithms, we are interested in having good lower bounds for the *Minimum Stretch Spanning Tree* problem. It is often the case that lower bounds lead directly to good approximation algorithms or that they help analyzing them. With this hope in mind we are going to look at some lower bounds for the stretch of graphs. Some of them will be used later in order to prove results on approximation algorithms, others just help us to get a clearer understanding of *Minimum Stretch Spanning Trees*. A lower bound for the stretch of planar graphs is presented in [15]. We will not discuss this lower bound here.

2.1 A Lower Bound for General Graphs

We first discuss a lower bound that holds for all graphs, weighted or unweighted. In every graph, the length of the longest[1] non-abbreviatable cycle[2] (minus 1, for there are no cycles in the spanning tree) is a lower bound for the graph's stretch. By non-abbreviatable we mean that there exists no abbreviation for any set of edges in the cycle. The longest non-abbreviatable cycle is exactly the same as the longest shortest path between the two endpoints

[1]We defined a measure for distances in unweighted graphs in the introduction. For weighted graphs, the measure is obvious.

[2]A cycle is a path $[v_0, v_1, \ldots, v_n]$, where $v_0 = v_n$, i.e. the path is starting and ending at the same vertex.

of an edge. Therefore it has the same length as the maximum stretch of any edge in the graph[3].

Lemma 2.1.1 *Given a graph $G = (V, E)$, the following holds.*

$$Stretch(G) \geq \max_{e \in E}\{Stretch_G(e)\}$$

Proof: Let e_{max} be the edge with maximum stretch in G. If $Stretch_G(e_{max}) = 0$, G is a tree and contains no cycles. Therefore, $Stretch(G) = 0$ too. If $Stretch(e_{max}) > 0$, e_{max} is part of at least one cycle in G. According to our definition of an edge's stretch, the shortest cycle on which e_{max} lies has length $Stretch_G(e_{max}) + 1$. In the minimum stretch spanning tree of G, this cycle doesn't exist anymore, i.e., at least one edge is missing. Therefore there must be an edge that was on this cycle before and produces a stretch which is at least $Stretch_G(e_{max})$. $\square$

2.2 Lower Bounds for Grid subgraphs

Our next graph family for which we look at lower bounds is the family of grid subgraphs. Both lower bounds mentioned in this section need an important property of grid subgraphs stated in the following lemma.

Lemma 2.2.1 *Inside a cycle of k edges, there can be no more than $\frac{k}{2}(\frac{k}{4}+1) = O(k^2)$ edges. And if there is a set of h edges, the shortest cycle that encloses those h edges cannot be shorter than $-2 + 2\sqrt{1 + 2h} = \Omega(\sqrt{h})$. This means that the number of edges and vertices[4] in an area A is at most proportional to the size of the area.*

$$|V(A)| \in O(A)$$

Proof: The cycle which contains the maximum number of vertices and edges has a quadratic shape. The lemma follows directly. Another way to show the proportionality is the following. For every face which is added to a grid subgraph, one can add at most 4 edges and 4 vertices to the graph. And, vice versa, one cannot add more than 4 vertices or edges without adding a new face too. The proportionality follows. $\square$

[3]See the definition of an edge's stretch in definition 1.1.1.

[4]Note that $|E(A)| \in O(A)$ follows from planarity because $|E| \leq 3n - 6$.

Note that this lemma does not hold for planar graphs where one can add arbitrary many vertices and edges for every face and vice versa. Provided with this property, we can now discuss a non-trivial lower bound for grid subgraphs. The idea behind this lower bound is that of a path of faces that connects a face f_i with the ∞-face. The action of every algorithm that computes a *Minimum Stretch Spanning Tree* can be interpreted as finding a way to merge the faces of the graph until there is only one face, the ∞-face left. A *Minimum Stretch Spanning Tree* can be understood as a graph for which all of its faces have been opened to the ∞-face. We now want to formalize this idea and use it to prove a lower bound.

Let $G = (V, E)$ be a grid-subgraph and let f_i denote the faces in a planar embedding of G. The set of edges which are adjacent to the face f_i is called $E(f_i)$.

Definition 2.2.1 *Path(f_a,f_b) is an ordered list of faces [f_0,f_1,f_2, $\dots$,f_n] which have the following properties:*

- *$f_0 = f_a$ and $f_n = f_b$.*
- *all the f_i are different (for $0 \leq i \leq n$).*
- *$E(f_{i-1}) \cap E(f_i) \neq \{\}$ for $1 \leq i \leq n$.*

For the computation of the lower bound, we are interested in the perimeter of such a path of faces which can be computed as:

$$p([f_0, f_1, ..., f_n]) = \sum_{i=0}^{n} |E(f_i)| - 2\sum_{i=1}^{n} |E(f_{i-1}) \cap E(f_i)|$$

For each Face f_i we compute the path of faces which has the smallest perimeter and connects it with the ∞-face (f_∞). For setting up the lower bound we are interested in the path having the longest, shortest perimeter. This longest, shortest perimeter is called k.

$$k = \max\left\{\min\left\{p\left(Path\left(f_i, f_\infty\right)\right)\right\}\right\}$$

Lemma 2.2.2 *Given a grid subgraph G, $Stretch(G) \in \Omega(\sqrt{k})$.*

Proof: Given an algorithm Γ that computes a minimum stretch spanning tree[5] T, we assume that $Stretch(G) \notin \Omega(\sqrt{k})$. In G, there exists a face $f_{\max}$

[5]i.e. $Stretch(G) = Stretch_G(T)$.

having a maximum perimeter with length k as defined above. This face $f_{\max}$ is connected to f_∞ through a set M of faces. We now add all the edges to T which were removed by Γ and which are adjacent to f_∞. The result is called $T^\star$.

$$E(T^\star) = E(T) \cup \{e_i : e_i \in E(f_\infty)\}$$

If our assumption that $\mathit{Stretch}(G) \notin \Omega(\sqrt{k})$ holds, then there exists no cycle in $T^\star$ with $\mathit{length} \in \Omega(\sqrt{k})$, because this would produce a $\mathit{Stretch}(G) \in \Omega(\sqrt{k})$. Let $E(M)$ denote the set of edges which are adjacent to at least one face of M. M must contain a path of faces connecting $f_{\max}$ with f_∞. The perimeter of this path cannot be longer then $\min\{p(\mathit{Path}(f_{\max}, f_\infty))\} = k$. Therefore, $|E(M)| \geq k \in \Omega(k)$. All edges of M are enclosed by a cycle in $T^\star$. Because there can only be $\Omega(\sqrt{k})$ edges on a cycle around k edges[6], it follows, that there must exist a cycle with $\mathit{length} \in \Omega(\sqrt{k})$ which is contrary to our assumption. $\square$

Provided with this lower bound for grid subgraphs, it is easy to see that the square root of the longest face length[7] in a grid subgraph is a lower bound for the stretch. This is true because if f_l denotes the face with maximum face length, the face length is equal to $p(f_l)$, which is of course part of $\mathit{Path}(f_l, f_\infty)$.

2.3 A Lower Bound for Regular Grids

Let us again use a stronger restriction of the problem's input graphs, which leads us to complete grids $(n \times m)$-grids. These graphs have a very regular structure. All faces have the same size and shape. This regular structure relieves the task of proving a lower bound. The following lemma states a lower bound[8] for regular grids.

Lemma 2.3.1 *Given a $(n\times m)$-grid G, the following holds for $n \geq 3$, $m \geq 3$.*

$$\mathit{Stretch}(G) \geq 2\left\lceil \frac{1}{2}\min\{n, m\}\right\rceil + 1$$

[6]See Lemma 2.2.1.

[7]See Section 2.4 for an explanation, why it isn't the face length but it's square root which is the lower bound.

[8]Which is the optimal solution for regular grids at the same time. See Chapter 3.

Proof: Due to the regular structure of a grid graph G the following holds.

$$Stretch(G) = \max_{face f_i} \{p(Path(f_i, f_\infty))\}$$

Let us call the faces, which produce this maximum stretch, max faces. There

Figure 2.1: A (6×4)-grid with marked max faces.

is no set of faces that connects such a max face with the ∞-face and has a shorter perimeter than the direct path. The perimeter of such a path can be computed as the lemma states. □

2.4 Alleged Lower Bounds

At the end of this section we want to address some alleged lower bounds. These seem to be lower bounds at first glance, but do not withstand deeper investigations. The first of these alleged lower bounds is

$$Stretch(G) \ngeq \max_{f \in F(G)} \{p(f)\} - 1,$$

where $p(.)$ is the perimeter of a face as defined above. As the counterexample in Figure 2.2 shows, this is not a correct lower bound. The perimeter of the "E"-shaped face is 22, therefore the stretch would have to be 21. But the minimum stretch spanning tree which is drawn in Figure 2.2 has stretch 15, which is the stretch of this graph.

Another alleged lower bound is

$$Stretch(G) \ngeq k = \max \{\min \{p\,(Path\,(f_i, f_\infty))\}\}$$

See Figure 2.3, where $p(Path(f_{\max}, f_\infty)) = 35$, but the minimum stretch spanning tree, drawn in Figure 2.3, has stretch 27. Obviously, there is a set of faces, containing $f_{\max}$ and f_∞, which has a smaller perimeter than $Path(f_{\max}, f_\infty)$.

Figure 2.2: A counter example for the max-facelength lower bound.

Figure 2.3: A counter example for the max-min-path lower bound.

Chapter 3

MSST on Regular Vertex Grids

Regular vertex grids are an interesting form of planar graphs. A number of graph problems can be simplified by restricting the input graphs to vertex grids. This is also true for the problem of finding a *Minimum Stretch Spanning Tree*. We have already defined the regular $(n \times m)$-vertex grid in definition 1.2.1 and in Section 2.3 we proved a lower bound for MSST on regular grids. In this section we are going to show that this lower bound is equal to the minimum stretch for every regular grid and that there is a polynomial time algorithm that computes an optimal stretch spanning tree. But, first of all, we need to prove a general property of regular grids which we will use later.

Lemma 3.0.1 *Given a planar embedding of a regular $(n \times m)$-grid $G = (E, V)$, where F is the set of faces and $N = |V|$. Then,*

$$|E| \leq 2N - 2\sqrt{N} \in O(N) \quad \textit{and} \quad |F| \leq 2 + N \in O(N).$$

Proof: The regular $(n \times m)$-grid with the maximum number of edges for a given number of vertices is a square. And a square of N vertices cannot contain more than $2N - 2\sqrt{N}$ edges. Note that this bound is tight. The upper bound for the number of faces follows from Euler's Formula which says: $|V| - |E| + |F| = 2$. The number of vertices is N and therefore $|E| \leq 2N - 2\sqrt{N}$. If we insert this in Euler's Formula we get $|F| \leq 2 + N - \sqrt{N}$ and it follows that $|F| = O(N)$. $\square$

3.1 A Polynomial Time Algorithm

Let us now present a polynomial time algorithm which reaches the lower bound for every regular grid and therefore computes the optimal MSST of any regular grid. The algorithm is called machete algorithm, because it goes from the grid's border step by step to the inner faces of the grid, just like chopping down a piece of rain forest.

Algorithm 1: Machete algorithm

```
input   : A regular grid $G = (V, E)$
output  : A minimum stretch spanning tree of $G$
repeat
    Mark all edges in $G$ that are adjacent to the $\infty$-face;
    foreach face $f_i$ adjacent to the $\infty$-face do
        delete one marked edge of $f_i$ from $G$;
    end
until There are no faces other than the $\infty$-face;
return $G$;
```

Theorem 3.1.1 *Given a $(n \times m)$-grid G, where $n \geq 3$, $m \geq 3$, the above algorithm computes a minimum stretch spanning tree T of G with*

$$Stretch_G(T) = Stretch(G) = 2\left\lceil \frac{1}{2}\min\{n, m\} \right\rceil + 1.$$

Proof: The algorithm works from the border to the inner part of the graph. Let us introduce the idea of face layers in a grid graph. A layer consists of all faces that have the same face distance to the ∞-face, see figure 3.1. In every execution of the repeat-until loop, one of these face layers is opened, i.e. merged with the ∞-face. Therefore the algorithm needs $\frac{1}{2}\min\{n, m\}$ outer loops. It follows, that the stretch is $2\lceil\frac{1}{2}\min\{n, m\}\rceil + 1$ and because this is equal to the lower bound of Lemma 2.3, we know that this is the optimal solution. $\square$

Figure 3.1: The face layers of a regular $(n \times m)$-grid.

3.2 Time and Space Complexity of the Algorithm

Both, the time and the space complexity of the machete algorithm depend on the data structure we use to store the input graph. A good data structure to deal with planar graphs is the doubly connected edge list[1].

Theorem 3.2.1 *Given a planar embedding of a regular $(n \times m)$-grid $G = (V, E)$ and F, its set of faces, the time complexity of the machete algorithm is in*

$$O((|E| + |F|)\min\{n, m\}) = O(|V| \min\{n, m\})$$

and its space complexity is in

$$O(|V| + |E| + |F|) = O(|V|)$$

Proof: Marking the edges which are adjacent to the ∞-face takes $O(|E|)$ time. One execution of the for-each loop takes $O(|F|)$ time, because one has to go through all the faces. The number of layers in a regular grid is $\frac{1}{2}\min\{n, m\}$, which is equal to the number of executions of the repeat-until loop. From Lemma 3.0.1 we know that $O(|E| + |F|) = O(|V|)$. The time complexity follows.

The space complexity is determined by the doubly connected edge list, where a constant amount of data is used for every vertex and edge. For general planar graphs, the amount of storage for a face is linear in the complexity of the subdivisions of this face. Because there are no subdivisions in

[1]See [13] for explanations about the doubly connected edge list.

Figure 3.2: A grid subgraph where the machete algorithm is not optimal.

the faces of regular grids, the amount of storage for faces is constant too. $O(|V| + |E| + |F|) = O(|V|)$ follows from Lemma 3.0.1. □

3.3 This Algorithm on Grid Subgraphs

Let us take a look on how the machete algorithm works on grid subgraphs. Unfortunately it is possible to construct examples, where it is far from being optimal. See figure 3.2 for a such a graph. What is the problem with this graph? The optimal stretch would be $k + 4 = O(k)$, but the machete algorithm returns a spanning tree with stretch $\frac{1}{3}(k-6)(k-2)+7 = O(k^2)$. The reason for this can be explained using again the concept of layers. During the first main loop of the machete algorithm, all faces of the first layer are merged with the ∞-face. These are all the small faces on the border of the graph and the first face with perimeter k. In the second execution of the main loop all the other faces with perimeter k are merged with the ∞-face, because these are layer 2 faces. In short: the problem is, that all the large faces lie in layer 2 and therefore produce the large stretch.

Chapter 4

OPT^4 Approximation for Grid Subgraphs

In this chapter we will present an algorithm based on the lower bound in Section 2.2. From this lower bound we know, that the optimal solution is in $\Omega(\sqrt{k})$, where k was defined as

$$k = \max\left\{\min\left\{p\left(Path\left(f_i, f_\infty\right)\right)\right\}\right\}. \tag{4.1}$$

To make a long story short, our algorithm will compute the minimal path for each face which leads to a tree on the faces, rooted in the ∞-face. Then the algorithm will merge all the faces according to this tree which leads to a stretch of at most k^2. Therefore, if the optimal solution has stretch OPT, the stretch tree computed by our algorithm has at most stretch OPT^4.

4.1 Weighted Dual Graphs

This was the overall idea. Let us now go into the details of the algorithm. We will first introduce the notion of a weighted dual graph of an unweighted grid subgraph. See Figure 4.1 for an example.

Definition 4.1.1 (Dual graph) *A weighted dual graph of an unweighted grid subgraph $G = (V, E)$ is a graph $G^\star$, that contains a vertex for every face of G:*

$$V(G^\star) = F(G)$$

Figure 4.1: An unweighted grid subgraph and the corresponding weighted dual graph.

and $G^\star$ contains an edge between two vertices if and only if the two vertices are adjacent faces in G:

$$E(G^\star) = \{\{f_i, f_j\} : f_i, f_j \in V(G^\star) \ \textit{ and } \ E(f_i) \cap E(f_j) \neq \{\}\}.$$

The weight of an edge in $G^\star$ is the number of common edges of the corresponding faces in G which is defined by the function $w : E(G^\star) \mapsto \mathbb{N}$:

$$w(\{f_i, f_j\}) = |E(f_i) \cap E(f_j)|$$

Note that the ∞-face is also represented by a vertex in $G^\star$. In the algorithm we will refer to the face length of a vertex $v \in V(G^\star)$ by *v.facelength*, which can be computed by adding the weights of the vertex' adjacent edges. In the dual graph, we can compute the perimeter $p(S)$, where S is a set of faces in G, without difficulty according to the following formula, i.e., we have to sum up all the face lengths of faces in S and then subtract two times the weights of all their common edges.

$$p(S) = \left(\sum_{v \in V(G^\star)} v.\mathit{facelength} \right) - 2 \left(\sum_{e \in E(S)} w(e) \right) \tag{4.2}$$

Figure 4.2: A grid subgraph and the spanning tree which the k^2 algorithm returns. The paths of faces are marked in the result.

4.2 The Algorithm

The algorithm is based on Dijkstra's method[1] of finding shortest paths in a graph. In order to use Dijkstra's algorithm for our problem, we just have to adapt the function that computes the distances. For our purpose we need a distance function which keeps track of the chosen faces' perimeter. We start at the ∞-face and compute the paths with minimum perimeter to the inner faces. This can be achieved by using the slightly changed algorithm of Dijkstra on the weighted dual graph. Finding the minimum perimeter $Path(f_i, f_\infty)$ is the same as computing a shortest path tree, rooted in the ∞-face, on the dual graph of our input graph, where it is not the distance but the perimeter of the path that has to be minimized. Given a path of faces from f_∞ to f and the perimeter of this path. And given a face f_{new} that shall be added to the path. In the dual graph, the faces are vertices and f_{new} is connected to a face of the path by an edge e_{new}. We can compute the perimeter of the path we had so far plus the new face as follows.

$$p(Path(f_\infty, f) + f_{new}) = p(Path(f_\infty, f)) + 2w(e_{new}) + f_{new}.facelength$$

Thus we know how to keep track of the perimeter. We will save the so far computed perimeter of $Path(f, f_\infty)$ in $f.perimeter$. In addition we note the predecessor of every face f in the paths of faces in $f.predecessor$ and $f.chosen$ is used to decide if a face has already been chosen by the algorithm.

[1]See [7] or [14] for explanations about Dijkstra's algorithm.

Every face, i.e. vertex of the dual graph is a member of one of three classes. It is either *chosen*, *reached* or *unreached*. For every face f that is chosen, there is already known a path of faces from f to f_∞ with minimal perimeter. Further, there is known a path to every chosen face and there is no such known path for every unreached face. All the reached faces are stored in the border B.

See Figure 4.2 for an illustration of the algorithm. An input grid subgraph is drawn on the left side. The resulting approximated spanning tree is drawn on the right side. There are also drawn the paths of faces with minimum perimeter, which result from the adapted Dijkstra algorithm.

4.3 Proof of Correctness and the OPT^4 Bound

Let us now proof that the spanning tree computed by the above algorithm in deed has a stretch of at most OPT^4. In a first step we will show that the algorithm computes the paths of faces with minimum perimeter correctly. Then we will show that the resulting stretch cannot be greater than k^2. This leads us to the following theorem.

Theorem 4.3.1 *Given a grid subgraph $G = (V, E)$ with $Stretch(G) = OPT$. For the spanning tree T_{alg}, returned by the k^2 algorithm, the following holds. Then follows the initialization of the vertices.*

$$Stretch_G(T_{alg}) \in O(OPT^4)$$

Proof: We will prove this theorem by proving the following two lemmas. Given a grid subgraph $G = (V, E)$ as input,

Lemma 4.3.1 *The steps 1 to 5 of the k^2 algorithm compute for each face $f_i \in F(G)$ a path of faces $Path(f_i, f_\infty)$ with minimum perimeter, and*

Lemma 4.3.2 *Step 6 of the algorithm, where edges are removed, leads to a stretch of at most k^2, where k is defined as in equation 4.1.*

From these lemmas follows the theorem, because according to Lemma 2.2 the optimal solution $OPT \in \Omega(\sqrt{k})$, and therefore $O(k^2) = O(OPT^4)$. $\square$

Proof (Lemma 4.3.1): The first step of the algorithm just computes G's dual graph $G^\star$. To use Dijkstra's principle for finding a shortest path, two conditions must be satisfied, as explained in [14]. Our notion of paths of faces do satisfy these conditions, because

Algorithm 2: k^2 algorithm

```
input  : A grid subgraph $G = (V, E)$
output : A spanning tree $T$ of $G$ with $Stretch_G(T) \leq k^2$
1 Compute $G$'s the dual graph $G^\star$;
2 /* Initialization */
  foreach $f \in V(G^\star)$ do
      $f.perimeter := \infty$;
      $f.predecessor := undefined$;
      $f.chosen := false$;
  end
3 /* $f_\infty$ is the start vertex */
  $f_\infty.perimeter = 0$;
  $f_\infty.predecessor := f_\infty$;
  $f_\infty.chosen := true$;
4 ExtendBorder($f_\infty$);
5 /* Compute shortest paths starting from $f_\infty$ */
  while $B \neq \{\}$ do
      choose $v \in B$, where $v.perimeter$ is minimal and remove $v$ from $B$;
      $v.chosen := true$;
      ExtendBorder($v$);
  end
6 /* Delete the edges induced by the shortest path tree */
  foreach $f \in V(G^\star)$ do
      Delete an edge $e \in \{E(f) \cap E(f.predecessor)\}$ in $G$;
  end
  return $G$;
```

Procedure `ExtendBorder(`v`)`

```
foreach $\{v, v'\} \in E(G^\star)$ do
    if $v.perimeter - 2w(\{v, v'\}) + v'.facelength < v'.perimeter$ then
        $v'.predecessor := v$;
        $v'.perimeter := v.perimeter - 2w(\{v, v'\}) + v'.facelength$;
        add $v'$ to $R$
    end
end
```

1. for every path of faces with minimum perimeter $P_{\min}(f_a, f_b)$ and every edge $\{f_a, f_c\}$ it is true that

$$p\left(P_{\min}(f_a, f_b)\right) + p\left(f_c\right) - 2\left|E\left(P_{\min}(f_a, f_b)\right) \cap E\left(f_c\right)\right| \geq p\left(P_{\min}(f_a, f_c)\right)$$

2. for at least one path of faces with minimum perimeter $P_{\min}(f_a, f_b)$ and every edge $\{f_a, f_c\}$ it is true that

$$p\left(P_{\min}(f_a, f_b)\right) + p\left(f_c\right) - 2\left|E\left(P_{\min}(f_a, f_b)\right) \cap E\left(f_c\right)\right| = p\left(P_{\min}(f_a, f_c)\right).$$

Basically this means that the perimeter of a path of faces can not become shorter when adding a face to the path. This would not be true in the case of arbitrary sets of faces, but for our case where we only use paths of faces as defined in definition 2.2.1 the above conditions hold. Therefore the algorithm computes the path with minimum perimeter for every face of the graph. □

Proof (Lemma 4.3.2): The problem that might arise while deleting edges in step 6 of the algorithm is, that several paths of faces could have some faces in common and that their perimeter gets enlarged therefore. We have to show now, that the perimeter will not become larger than k^2, where k is defined as in equation 4.1.

The perimeter of the longest shortest path of faces has length k, by definition. Let us call the face that implies this path f_{max}. This means that there are k edges on that perimeter, where another path could be appended, to use then the same path of faces as f_{max}. The additional parts of these appended paths cannot be longer than k, for $Path(f_{max}, f_\infty)$ with perimeter k is the path with the maximum perimeter. This means that all this appended paths cannot be farther away than $\frac{k}{2}$ vertices, so they all lie in a circle with

Figure 4.3: The longest shortest path of faces, together with appended paths.

radius $\frac{k}{2}$. See Figure 4.4. This circle has a perimeter of πk. It follows from Lemma 2.2.1 that there can only be $O(k^2)$ edges inside this circle. Therefore the perimeter of all the paths is in $O(k^2)$. $\square$

4.4 Time and Space Complexity of the Algorithm

For graphs with $|E| \leq |V|^2$ and with a heap for managing the vertices in the border, Dijkstra's algorithm needs time $O(n \log n)$, where $n = |V|$.[2] Using a doubly connected edge list[3], the dual graph of a grid subgraph can be computed in linear time $O(n)$. Removing the edges in step 7 of the algorithm needs linear time again. The space needed for the computation is determined by the way we save the graph. As we have already seen in Section 3.2, the doubly connected face list needs space in $O(n)$. This leads us to the following theorem.

[2]See [14].

[3]See [13].

Figure 4.4: An example where the k^2 upper bound of the algorithm is tight.

Theorem 4.4.1 *Given a planar embedding grid subgraph $G = (V, E)$, where $n = |V|$, the time complexity of the k^2 algorithm is in $O(n \log n)$ and its space complexity is in $O(n)$.*

4.5 Some Remarks about the OPT^4 Bound

How good is this OPT^4 bound? For the k^2 algorithm, the bound is tight, i.e., there are examples of grid subgraphs where the spanning tree returned by the algorithm has a stretch in $\Theta(OPT^4)$. Such an example is shown in the Figures 4.4 and 4.5. Let us first look at Figure 4.4. This is an example, where the paths of faces have many faces in common. The arrows show the paths with minimum perimeter. Removing the edges to construct the spanning tree leads to a stretch in $\Theta(k^2)$ in this example. Therefore Figure 4.4 shows that the upper bound of the algorithm is tight, whereas Figure 4.5 presents a graph where the optimal stretch's lower bound, presented in Section 2.2, is tight. This is true because the minimum stretch would be reached removing the edges inside the grid subgraph, while the k^2 algorithm will delete edges on the border. k is the stretch that results from computing the paths of faces with minimum perimeter. The minimum stretch would have been in $\Theta(\sqrt{k})$.

We can now combine these two examples by filling all faces of Figure 4.4 as demonstrated in Figure 4.5. What we get is an example where the minimum stretch is in $\Theta(\sqrt{k})$ and the stretch computed by the algorithm is

Figure 4.5: Here, the $\sqrt{k}$ lower bound of the minimum stretch is tight.

in $\Theta(k^2)$. This shows that the OPT^4 bound is tight.

The OPT^4 bound can also be expressed in terms of n, the number of vertices. We can show, that the spanning tree resulting from the k^4 algorithm is an $n^{\frac{3}{4}}$-approximation of the *Minimum Stretch Spanning Tree.* To see this we have to investigate two cases. First, if $Stretch_G(T_{OPT}) \leq \sqrt[4]{n}$ it follows that the approximation factor

$$\frac{Stretch_G(T_{Alg})}{Stretch_G(T_{OPT})} \leq Stretch_G(T_{OPT})^3 \leq n^{\frac{3}{4}}.$$

Since the stretch of spanning tree computed by the algorithm cannot be larger than n, we know that for the other case where $Stretch_G(T_{OPT}) > \sqrt[4]{n}$ follows that

$$\frac{Stretch_G(T_{Alg})}{Stretch_G(T_{OPT})} \leq \frac{n}{n^{\frac{1}{4}}} = n^{\frac{3}{4}}.$$

Chapter 5

An Abortive Evolutionary Algorithm

Beside the OPT^4 algorithm we want to discuss some other possible approaches to compute a *Minimum Stretch Spanning Tree.* The first one is an evolutionary algorithm which starts with a random solution and tries to improve it. We will see that our problem cannot be solved with this kind of algorithm. Another idea are two very simple greedy algorithms, which will be discussed in the next chapter.

5.1 The Algorithm

In this evolutionary approach, the algorithm is given an arbitrary spanning tree together with the input graph. Then the algorithm keeps changing edges which are in the tree with edges that are not in the tree, as long as these changes lead to an improvement of the tree's stretch. More formally: We define a function that assigns a characteristic value to every possible spanning tree. Then we change an edge of the given spanning tree with an edge of the input graph if thereby the tree's resulting value improves[1]. We will first present the skeleton of the algorithm and then discuss several functions to evaluate a spanning tree.

Both, the evaluation function f and the initial spanning tree are inputs of the evolutionary algorithm. Of course, the algorithm could compute a random spanning tree by itself and we could also describe the algorithm

[1]Note that in our case an improvement is a decrease of the spanning tree's stretch.

Algorithm 4: Evolutionary Algorithm

```
input  : A grid subgraph $G = (V, E)$
         $T = (V, \hat{E})$, an arbitrary (e.g. random) spanning tree of $G$
         A function $compare : (G, T_1, T_2) \to \{-1, 0, 1\}$
output : A spanning tree of $G$

$changedEdge := true$;
while $changedEdge$ do
    foreach $Edge\ \{u, v\}\ in\ E \setminus \hat{E}$ do
        $T_{before} := T$;
        add $\{u, v\}$ to $T$;
        $changedEdge := false$;
        $pathLength :=$ length of the path from $u$ to $v$ in $T$;
        $i := 1$;
        while not $changedEdge$ and $i < pathLength$ do
            $e_i =$ edge $i$ on the path from $u$ to $v$ in $T$;
            delete $e_i$ from $T$;
            if $compare(G, T, T_{before}) = -1$ then
                $changedEdge := true$;
            end
            else
                add $e_i$ to $T$;
            end
            $i := i + 1$;
        end
        if $changedEdge = false$ then
            delete $\{u, v\}$ from T;
        end
    end
end
return $T$;
```

that way. The outer while loop ensures that the algorithm proceeds until there is no edge change that improves the value of the spanning tree. The foreach loop handles all the edges not part of the actual spanning tree T. For each of these edges, it is tested, if a change with another edge leads to an improvement of the tree's value. Note that we only need to look at edges which lie on the path in T that leads from one of the handled edge's endpoits to the other. This is the case because two edges can only be changed if they are on a cycle. Otherwise the change leads to an unconnected spanning tree. Because there is only one path between any two vertices in a tree, there is also only one cycle containing the actual handled edge, in T. And this cycle is the path between the endpoints of the edge. The first thing that happens in the foreach loop is that the current tree is stored in the variable T_{before}. Then the current edge is added to the tree. In the inner while loop, it is tested for all the edges on the path of the edge if a change leads to an improvement of the tree, and if so, the edges are changed and the inner loop is quitted[2]. If no edge was changed within the inner while loop, the added edge has to be removed.

5.2 Two Evaluation Functions for this Algorithm

Let us now discuss two evaluation functions for this algorithm. We will present the functions and show why they are not appropriate to solve the problem of finding a *Minimum Stretch Spanning Tree* of a grid subgraph. The first evaluation function is quite simple:

$$compare_1(G, T_1, T_2) = \begin{cases} -1, & \text{if } Stretch_G(T_1) < Stretch_G(T_2) \\ 0, & \text{if } Stretch_G(T_1) = Stretch_G(T_2) \\ 1, & \text{if } Stretch_G(T_1) > Stretch_G(T_2) \end{cases}$$

The problem with this evaluation function is that it cannot handle spanning trees in which two edges produce the same maximum stretch. The spanning tree in Figure 5.1 has stretch 13, which is reached by two of the edges. Although a change of edges would be possible, it does not take place because

[2]Another variant of the algorithm would be to look for the change of edges which improves the spanning tree at most. But this variant wouldn't solve the problem that arises in the examples presented later either.

Figure 5.1: The evaluation function *compare*$_1$ leads to a bad stretch here.

it does not improve the stretch of the spanning tree. An improvement would only occur after two changes. Therefore, nothing is changed and the returned spanning tree has a very large stretch. One could easily create an example with an arbitrary bad stretch. A tree with more than two edges reaching the maximum stretch leads to an example where more changes must take place at once. Regarding these problems we introduce another evaluation function that works with lexicographic comparisons on descending lists of stretches:

$$compare_2(G, T_1, T_2) = \begin{cases} -1, & \text{if } StretchList_G(T_1) <_l StretchList_G(T_2) \\ 0, & \text{if } StretchList_G(T_1) =_l StretchList_G(T_2) \\ 1, & \text{if } StretchList_G(T_1) >_l StretchList_G(T_2) \end{cases}$$

where

$$StretchList_G(T) = [Stretch_T(e_0), Stretch_T(e_1), ..., Stretch_T(e_n)]$$

with $e_i \in E \setminus \hat{E}$ and $Stretch_T(e_i) > Stretch_T(e_{i+1})$ for every $i < n$. The operators with the subscript l denote lexicographic comparisons[3]. This second evaluation function will handle the problems discussed so far. But there are other cases where this function fails too. Let us describe such an example where *compare*$_2$ cannot improve a spanning tree although it would have been possible. We can achieve this with a spanning tree where two or more edges have to be changed in order to improve its stretch. Therefore we need a spanning tree where changing only one pair of edges leads to a tree with the same stretch list, but changing two pairs of edges improves the tree. This is the case in the example in Figure 5.2. The stretch of the spanning tree drawn in the figure is k^2 but the minimum stretch would be k. There is no pair of edges changed because every change of only one edge in $E \setminus \hat{E}$ leads to a spanning tree with the same stretch list. Knowing this example, it is

[3]By lexicographic we mean that the stretch of all edges in $E \setminus \hat{E}$ is considered. To make an edge change take place it would be enough to improve the stretch of one of these edges, even if this would not change $Stretch_G(T)$.

Figure 5.2: The evaluation function *compare*$_2$ leads to a bad stretch here.

Figure 5.3: The evaluation function *compare*$_2$ leads to a stretch in $O(n)$ here.

easy to construct examples where three or more pairs of edges have to be changed until the stretch list improves.

After all we will present an example with stretch in $O(n)$, where n is the number of vertices in the grid subgraph. The example is shown in Figure 5.3. A spanning tree as it is drawn in this figure cannot be improved by changing a pair of vertices. The example can easily be enlarged. The stretch of the drawn spanning tree T can be computed as follows. The number of the rectangles consisting of two k-cycles is

$$\frac{n-3}{3\frac{k-2}{2}}.$$

Each of these rectangles adds $\frac{k-2}{2}$ to the stretch. At the beginning of the sequence 1 is added to the stretch and at the end 2 is added to the stretch. It follows that

$$Stretch_G(T) = \frac{n-3}{3\frac{k-2}{2}} \frac{k-2}{2} + 3 = 2\frac{n-3}{3} + 3 \in O(n).$$

Recapitulating what we have seen in this chapter we can conclude that evolutionary algorithms that change pairs of edges are not convenient to solve the problem of finding a *Minimum Stretch Spanning Tree*. This is the case because there are spanning trees where we have to change an arbitrary number of edge pairs in order to improve the tree.

Let us look to the problem from a different perspective to make this clear. We can think of this problem as minimizing a function. The function takes a spanning tree and assigns a stretch to it for a given grid subgraph. To find the minimum stretch spanning tree we have to minimize this function. This is exactly what our algorithm does. It tries to find a local minimum of this function by going always in a direction that leads to a lower stretch. The problem that arises is, that we can construct grid subgraphs where there are spanning trees whose stretch lie in an arbitrary large plateau. This means that all the neighbors in an arbitrary distance of these spanning trees have the same stretch. This makes it impossible for our algorithm to minimize the function.

Chapter 6

Remarks about two Greedy Algorithms

Another idea we want to discuss is that of greedy algorithms. As it is often the case with algorithms of this nature, the greedy algorithms we are going to describe are very straightforward and uncomplex. The basic idea behind the algorithms is: as long as there are cycles, remove the edge, that enlarges the stretch of the resulting tree as little as possible.

6.1 A Greedy Algorithm

See Algorithm 5 for a description of a first version of a greedy algorithm.

Of course, this algorithm might remove suboptimal edges, but this is due to its greedy nature. The question is, how bad this can go. For the algorithm

Figure 6.1: $O(n)$ stretch example for the first greedy algorithm.

Algorithm 5: Greedy Algorithm

```
input  : A grid subgraph $G = (V, E)$
output : A spanning tree of $G$

Make $T$ a copy of $G$;
while $|E(T)| > |V(T)| - 1$ do
    $Stretch_{min} := \infty$;
    foreach $e \in E(T)$ do
        $E(T) := E(T) \setminus \{e\}$;
        if $Stretch_T(G) < Stretch_{min}$ then
            $e_{min} = e$;
            $Stretch_{min} = Stretch_T(G)$;
        end
        $E(T) := E(T) \cup \{e\}$;
    end
    $E(T) := E(T) \setminus \{e_{min}\}$;
end
return $T$;
```

described above, we can show an example where the minimum stretch is 3 and the stretch obtained by the algorithm is in $O(n)$. This example is presented in Figure 6.1. First the algorithm will remove some edges that produce stretch 3. Assume it removes the edges marked in Figure 6.1 (2). After the algorithm has removed these edges it must remove further edges. Every remaining edge will produce stretch 5. Assume the algorithm removes the edges marked in Figure 6.1 (3), then the edges in Figure 6.1 (4) and so on. It is obvious that we can enlarge this example to make the algorithm produce every stretch we want, where the minimum stretch remains 3.

We could reduce this problem by using some randomness in the algorithm if there are several edges that produce the same stretch. This would prevent us from getting such a regular[1] result.

6.2 Time Complexity of the Greedy Algorithm

Let us prove the following theorem about the time complexity of the greedy algorithm.

Theorem 6.2.1 *Given a grid subgraph $G = (V, E)$. The greedy algorithm presented above has time complexity in $O(n^3 \log n)$.*

Proof: According to Lemma 3.0.1, a grid subgraph with n vertices can have at most $2n - \sqrt{n}$ vertices. Therefore the number of passes of the outer while loop can be $(2n - \sqrt{n}) - n + 1 = n - \sqrt{n} + 1 \in O(n)$. There can be $O(n)$ edges that are processed in the foreach loop, and for every of these edges the stretch has to be computed, i.e. a shortest path has to be found, which can be done in time $O(n \log n)$. The time complexity follows. $\square$

6.3 Improved Version of the Greedy Algorithm

There is another way to improve the algorithm besides the adding of randomness. We get an algorithm which can handle the problem mentioned

[1]Which is bad in this case, because it produces the large stretch.

above with a little change. Let us get a deeper understanding of what the greedy algorithm does, in order to see the little change in it, that leads to the improvement.

A way to analyze the greedy algorithm presented above is to investigate its behavior on the dual graph of its input. We can construct a dual graph as proposed in a previous chapter. Each face of the original graph becomes a vertex in the dual graph and to vertices of the dual graph are connected if and only if the corresponding faces have common edges. Therefore each edge of the dual graph stands for a number of edges in the original graph. We can then weight each of the dual graph's edges with the stretch that the removing of one of the corresponding edges in the original graph would produce. What the greedy algorithm does is nothing other than computing a minimum spanning tree on this weighted dual graph. The only problem is that the weights of the edges change throughout the execution of the algorithm because the more edges are removed by the algorithm, the larger becomes the stretch that is produced from removing further edges. But if we ignore this changing of the weights, the greedy algorithm computes a minimum spanning tree on the dual graph. It does so by always choosing the edge with the smallest weight that produces no cycle, as it is described in [14].

But this algorithm that traces back on [11], is not the only way to obtain a minimum spanning tree that is mentioned in [14]. Another possibility is the algorithm of Dijkstra, Prim and Jarnik [7, 19, 10]. This algorithm starts from one vertex and builds a so called border that contains all the edges that can be reached from this start vertex. Then the border's edge with minimum weight is added to the spanning tree and the border is extended by the edges adjacent to the other vertex of the chosen edge. These steps are repeated until a spanning tree is obtained. See [14] for more detailed explanations.

How can we implement this other way of obtaining a minimum stretch spanning tree? We just have to transform this idea back from the dual graph to the original grid subgraph. There this means that we have to start from one face and remove greedily the edge adjacent to this face, which produces a tree with minimum stretch. When we choose the ∞-face as start face, this leads us to the following algorithm.

Note that in this approach the ∞-face is merged with other faces until there is nothing else than the ∞-face. Therefore it is enough to add all the edges adjacent to the ∞-face, after the edge that produces the minimum stretch has been removed.

Algorithm 6: Improved greedy algorithm

```
input   : A grid subgraph $G = (V, E)$
output  : A spanning tree of $G$

Make $T$ a copy of $G$;
$Border := Border \cup \{e \in E) : e \text{ is adjacent to } f_\infty\}$;
while $|E(T)| > |V(T)| - 1$ do
    $Stretch_{min} := \infty$;
    foreach $e \in Border$ do
        $E(T) := E(T) \setminus \{e\}$;
        if $Stretch_T(G) < Stretch_{min}$ then
            $e_{min} = e$;
            $Stretch_{min} = Stretch_T(G)$;
        end
        $E(T) := E(T) \cup \{e\}$;
    end
    $E(T) := E(T) \setminus \{e_{min}\}$;
    $Border := Border \cup \{e \in T(E) : e \text{ is adjacent to } f_\infty\}$;
end
return $T$;
```

Figure 6.2: Quadratic example for the greedy algorithm.

Unfortunately we cannot present a result about the approximation factor of this algorithm. The only thing we can show is that the resulting tree can have stretch OPT^2. This is true because of the example presented in Figure 6.2.

In this example, the algorithm will remove the edges of the small rectangles first. The problem is, that it removes the edges on the outer border first. This leads to a large stretch. The minimum stretch would have been the perimeter of the large rectangle. But with these holes in it, the perimeter of the large face becomes quadratic in the minimum stretch. This is the case because the stretch consists of all the edges inside the rectangle. According to Lemma 2.2.1, the number of edges inside a cycle with length k is in $O(k^2)$.

Note that this algorithm becomes arbitrary bad on planar graphs where the Lemma 2.2.1 does not hold. On such a graph we can draw as many holes as we want in a cycle and therefore we can construct an example with any stretch we want.

6.4 Time Complexity of the Improved Greedy Algorithm

In this section we will investigate if the change in the greedy algorithm affects its time complexity. We can argue as before that the outer while loop can be processed $O(n)$ times. Because it is possible that there are $O(n)$ edges in the border, the foreach loop has to handle $O(n)$ edges. Computing the stretch needs $O(n \log n)$ as before. Therefore we see that the time complexity remains the same.

Chapter 7

MSST on Unit Disk Graphs

In this last chapter we want to consider which of the described algorithms can be used not only on regular grids and grid subgraphs, but also on unit disk graphs. These graphs consists of vertices in the plane. Two of the vertices are connected by an edge if and only if the distance between them is at most 1. The idea is not to look for new algorithms, but to think of how the algorithms described so far can be applied to this new graph family. Thereby it does not pay to rethink the evolutionary algorithm or the first version of the greedy algorithm, because they won't produce a better *Minimum Stretch Spanning Tree* approximation on unit disk graphs. But let us now investigate, if we can use the OPT^4-algorithm on unit disk graphs.

7.1 The OPT^4 Algorithm on Unit Disk Graphs

The OPT^4 algorithm described in chapter 4 is based on two properties of its input graph.

1. The input graph has to be planar, and
2. The number of vertices in an area A of the input graph is in $O(A)$.

Without the first property we could not speak about faces and paths of faces. The second property is the one, stated in Lemma 2.2.1, which is also crucial for the algorithm. Unit disk graphs do not fulfill either of these properties. But it is possible to transform them in a way that they satisfy these two constraints while the stretch is enlarged by a constant factor only. Such a transformation is described in [1]. Let us give a quick overview of how this works. For details, consult [1].

The first step of the transformation is to compute a dominating set of the unit disk graph. This can be achieved by computing a maximal[1] independent set, which is always a dominating set. This is true for all graphs[2]. If we focus only on the computed dominators, we have already met the second constraint mentioned above. This is true because of the inherent structure of unit disk graphs. All vertices that are in a circle with radius 1/2 are connected with each other. And therefore there can only be one dominator in such a circle, i.e. the number of dominators in an area A is in $O(A)$. But these dominators are not very useful yet because they are just an unconnected set of vertices. The next step is therefore to connect these dominators. If the original unit disk graph was connected, it is possible to connect the dominators with 2 or 3 hops each, i.e. each dominator is connected to its neighbour dominators which can be reached from it in two or three hops[3]. On the resulting graph we compute a Delaunay Triangulation[4] or a Gabriel Graph on the resulting graph, in order to meet also the first constraint and make the graph planar. All the vertices that were neither in the dominating set nor part of the connections between the dominators, can be connected to their dominators with one hop.

This construction leads to a spanner of the original unit disk graph. This spanner has constant stretch and satisfies both properties that we need to use the OPT^4-algorithm on it. Therefore we can approximate a *Minimum Stretch Spanning Tree* on unit disk graphs with this algorithm as we can do it on grid subgraphs.

[1]Note, that there is a difference between a "maximal" and a "maximum" independent set. The "maximal" independent set, we use here, is one where no more vertices can be added, but the number of vertices does not have to be a maximum or optimum.

[2]We can add no more vertices to a maximal independent set, i.e. every vertex that is not in it, is dominated by a vertex in the independent set. Therefore, a maximal independent set is a dominating set. See [1] for more details.

[3]This works because if there would be two dominators that could only be connected with 4 hops, i.e. 3 vertices in between them, it follows that there is a vertex that is not dominated and is no dominator at the same time, which is a contradiction to step 1 where we built a dominating set. See [1] for details.

[4]For our purposes the Delaunay Triangulation is prefered to the Gabriel Graph, because its stretch factor is smaller.

Chapter 8

Conclusions

We discussed various algorithms to compute a *Minimum Stretch Spanning Tree* in regular grids, grid subgraphs and unit disk graphs, where the main focus was on grid subgraphs. Although it seems that there exists an approximation algorithm that computes a spanning tree with stretch OPT^2 or $\sqrt{|V|}$ for grid subgraphs, we could not prove this. Intuitively, the improved version of the greedy algorithm and the machete algorithm appear to be good candidates for OPT^2-algorithms. This intuition is fortified by the fact that we have not been able to find a counter example yet. As opposed to these rather vague assumptions, we have also discussed and proved results about other algorithms. The main result of this diploma thesis is discussed in Chapter 4 where we present an OPT^4 algorithm together with an analysis of the upper bound of the stretch it produces. Now, the next step is to simulate this algorithm and to investigate how large the stretch becomes in real world examples. We assume that the stretch will be near to OPT^4 only in special examples, but it might be near OPT^2 in most of the relevant practical cases. The simulation will help to answer this question. After all, we proved some "negative" results and showed that the spanning trees resulting from an edge changing evolutionary algorithm can have a stretch in $O(n)$. It is interesting to see that there are examples where this approach does not work. The same is true for the simple greedy algorithm.

8.1 Open Problems

During the simulation of the evolutionary and the simple greedy algorithm it became clear that randomly chosen spanning trees have a surprising good stretch most of the time. It is worth investigating them and try to find out how probable it is to randomly choose a spanning tree with a large stretch.

Another thing that remains open is an analysis of the improved greedy algorithm and the machete algorithm. Either one can find an example which proves that the stretch of the resulting worst case spanning trees is in $\Omega(n)$, or it is possible to prove that the stretch is in $O(OPT^2)$. A different approach is to try to improve the OPT^4 algorithm so that the upper bound becomes smaller.

Because the problem of finding a *Minimum Stretch Spanning Tree* seems to be hard even in the simplified environment of grid subgraphs, it surely would pay to analyze if the problem hardly ever can be approximated better than a certain factor.

Bibliography

[1] K. Alzoubi, P.-J. Wan and O. Frieder, *Message-Optimal Connected Dominating Sets in Mobile Ad Hoc Networks*, Proc. of the 3^{rd} ACM Int. Symposium on Mobile ad hoc networking & computing (MOBIHOC), 2002.

[2] J. A. Bondy and U. S. R. Murty, *Graph Theory with Applications*, North-Holland, New York, 1976.

[3] L. Cai, *Tree spanners: spanning trees that approximate distances*, Ph.D. thesis, University of Toronto, Toronto, Canada, 1992. Available as *Technical Report 260/92*, Department of Computer Science, University of Toronto.

[4] L. Cai, *NP-completeness of minimum spanner problems*, Discrete Applied Mathematics, 48, pp. 187-194, 1994.

[5] L. Cai and D.G. Corniel, *Tree Spanners*, SIAM Journal on Discrete Mathematics, 8: 359-387, 1995.

[6] M. Demmer and M. Herlihy, *The Arrow Directory Protocol*, Proceedings of 12th International Symposium on Distributed Computing, 1998.

[7] E. W. Dijkstra, *A note on two problems in connection with graphs*, Numerische Mathematik, 1:269-271, 1959.

[8] S. P. Fekete and J. Kremer, *Tree Spanners in Planar Graphs*, 24th International Workshop on Graph-Theoretic Concepts in Computer Science, 1998.

[9] M. Herlihy, S. Tirthapura, and R. Wattenhofer, *Competitive Concurrent Distributed Queuing*, Proceedings of the Twentieth ACM Symposium on

Principles of Distributed Computing (PODC), Newport, Rhode Island, August 2001.

[10] V. Jarnik, *O jistem problemu minimalnim*, Moravske Prirodovedecke Spolecnosti 6, 57-63, 1930 (In Czech.).

[11] J. B. Kruskal, *On the shortest spanning subtree of a graph and the traveling salesman problem*, In Proc. AMS 7, S. 48-50, 1956.

[12] A. L. Liestman and T. Shermer, *Grid Spanners*, Networks, 23, pp. 123 - 133, 1993.

[13] D. E. Muller and F. P. Preparata, *Finding the intersection of two convex polyhedra*, Theoretical Computer Science, 7:217-236, 1978.

[14] T. Ottmann and P. Widmayer, *Algortihmen und Datenstrukturen*, 3rd Edition, Spektrum Akademischer Verlag, 1996.

[15] D. Peleg and D. Tendler, *Low Stretch Spanning Trees for Planar Graphs*, September 2001.

[16] D. Peleg and E. Reshef, *A variant of the arrow distributed directory protocol with low average case complexity*, in Proc. 25th Int. Colloq. on Automata, Language and Programming, LNCS, Vol. 1443, pages 670-681, Springer, Berlin, 1998.

[17] D. Peleg and J. D. Ullman, *An optimal synchronizer for the hypercube*, in Proc. 6th ACM Symposium on Principles of Distributed Computing, Vancouver, 1987, pages 77-85.

[18] D. Peleg and A. A. Schäffer, *Graph spanners*, Journal of Graph Theory, 13:99-116, 1989.

[19] R. C. Prim, *Shortest connection networks and some generalizations*, Bell System Techn. J., 36:1389-1401, 1957.

[20] G. Venkatesan, U. Rotics, M. S. Madanlal, J. A. Makowsky and C. P. Rangan, *Restrictions of minimum spanner problems*, Information and Computation, 136(2):143-164, 1997.