EBERHARD-KARLS-UNIVERSITÄT TÜBINGEN

BACHELOR'S THESIS

Visualizing Transitions in Euclidean Minimum Spanning Trees

Author:
Tobias Konrad ELSSNER
Student Number: 3751602

Supervisor:
Prof. Dr. Michael KAUFMANN

A thesis submitted in partial fulfillment of the requirements for the degree of Bachelor of Science

Mathematisch-Naturwissenschaftliche Fakultät
Fachbereich Informatik

09.03.2022

Hiermit versichere ich, dass ich die Arbeit selbständig verfasst, keine anderen als die angegebenen Hilfsmittel und Quellen benutzt, alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet habe und dass die Arbeit weder vollständig noch in wesentlichen Teilen Gegenstand eines anderen Prüfungsverfahrens gewesen ist und dass die Arbeit weder vollständig noch in wesentlichen Teilen bereits veröffentlicht wurde sowie dass das in Dateiform eingereichte Exemplar mit den eingereichten gebundenen Exemplaren übereinstimmt.

I hereby declare that this paper is the result of my own independent scholarly work. I have acknowledged all the other authors' ideas and referenced direct quotations from their work (in the form of books, articles, essays, dissertations, and on the internet). No material other than that listed has been used.

Tübingen, 09.03.2022

Firstname Surname

EBERHARD-KARLS-UNIVERSITÄT TÜBINGEN

Abstract

Mathematisch-Naturwissenschaftliche Fakultät
Fachbereich Informatik

Bachelor of Science

**Visualizing Transitions in
Euclidean Minimum Spanning Trees**

by Tobias Konrad ELSSNER
Student Number: 3751602

Minimum spanning trees are the building block of many applications, ranging from data visualization, over clustering to route and infrastructure planing. Whereas traditional setups consider a set of static nodes, this thesis examines a kinetized version. The goal is to calculate and visualize the topologies of a given euclidean minimum spanning tree in the plane, when one vertex is moving on a predefined, linear trajectory. To do so, the algorithm by Monma and Suri (1992) is re-implemented and integrated into a graphical user interface. The procedure subdivides the two-dimensional vector space into regions, in which all coordinates are connected to the same subset of stationary nodes.
During testing, it appeared that the approach is partially flawed. While one inaccuracy could be remedied, another error, occasionally causing falsely associated topologies for a small fraction of subregions, remains an open problem. To facilitate future investigations, the erroneous topologies can also be graphically highlighted in the interface.

Acknowledgements

First and foremost, I wish to thank my supervisor, Prof. Dr. Michael Kaufmann, who directed me at the topic and patiently guided me through this thesis.
Also, the completion of my studies would not have been possible without the support of my family and especially my parents, Ute and Günter Elßner.
Moreover, I am indebted to my friends[†] Armin Schuster, Björn Rudzewitz, Felix Wambsganz, Jens Gottfried and Marko Knab, to whom I owe the many helpful, motivational, and welcomingly diverting chats and meet-ups.
And, last but not least, I would like to appreciate my mate Ilex Paraguariensis and the instant ramen manufacturers for their stimulating inputs and seasoned advices over a cup or two.

Thank you all.

[†]In alphabetical order.

Contents

List of Figures

List of Abbreviations

MST	**M**inimum **S**panning **T**ree
EMST	**E**uclidean **M**inimum **S**panning **T**ree
GNN	**G**eographic **N**earest **N**eighbor
OVD	**O**riented **V**oronoi **D**iagram
GUI	**G**raphical **U**ser **I**nterface

Chapter 1

Introduction to Minimum Spanning Trees

This chapter is an introduction to the topic of minimum spanning trees, or MSTs for short. First, the general notation used throughout the thesis is laid out. Then, algorithms to construct MSTs are discussed, followed by a brief review of two closely related concepts, Delaunay triangulation and Voronoi diagrams. The third section gives a selection of applications for MSTs, and lastly, the problem to be tackled is presented.

1.1 Notational Preliminaries

For a better readability of subsequent sections and chapters, the notation of concepts used later on is presented first. If not stated otherwise, this chapter is based on the publication by Tarjan (Tarjan, 1983a).
An undirected graph $\mathcal{G}$ is formally defined by

$$\mathcal{G} = (V, E), \tag{1.1}$$

comprising of a set of vertices V and a set of edges $E \subseteq V \times V$. For the sake of brevity, n will henceforth denote the number of vertices, $|V|$, and m the number edges, $|E|$. $\mathcal{G}$ is called *undirected*, if $(u, v) \in E$, it also holds that $(v, u) \in E$. $\mathcal{G}$ is said to be connected, if (and only if) there exists at least one path between all pairs of nodes, and fully connected, if $E = \{(u, v) \mid (u, v) \in V \times V\}$. A *cut* divides V into two complementary sets, V' and V''. An edge (u, v) is said to cross the cut, if $u \in V'$ and $v \in V''$, or vice versa. The degree of a vertex v,

$$deg(v) = |\{(v, u) \mid (v, u) \in E\}| \tag{1.2}$$

returns the number of directly adjacent edges of v in $\mathcal{G}$.
For a given subset $V' \subset V$, the set of *incident* edges $i(V')$ denotes connections between vertices which are in V', and those which are not:

$$i(V') = \{(u, v) \in E | (u \notin V' \wedge v \in V') \vee (v \notin V' \wedge u \in V')\} \tag{1.3}$$

Any vertex $v \notin V'$ is said to *border* V', if

$$(v, u) \in i(V') \tag{1.4}$$

given some $u \in V'$.
If the edges are associated with weights, a weighting function is added to the notion above:

$$\mathcal{G} = (V, E, w), \tag{1.5}$$

where $w : E \mapsto \mathcal{A}$ maps to an arbitrary algebraic structure $\mathcal{A}$. Throughout the paper, and in many applications (see 1.4), $\mathcal{A}$ defines the set of real numbers, $\mathbb{R}$. Furthermore, the edge weights ought to satisfy the euclidean metric, situating the vertices in a d-dimensional vector space separated by euclidean distance. This imposes three constraints on the weighting function (O'Searcoid, 2006): Given some vectors $u, v, x \in V$ of size d, where u_i denotes the ith entry, first

$$\begin{aligned} w(u,v) &\geq 0 \\ w(u,v) &= 0 \leftrightarrow u = v \\ w(u,v) &= \sqrt{\sum_{i=1}^{d} (u_i - v_i)^2}, \end{aligned}$$

that is, self-loops are neglected, and the graph is otherwise fully connected with positive edges. Second, weights need to be symmetrical,

$$w(u,v) = w(v,u), \tag{1.6}$$

meaning the graph is undirected. And third, the triangle inequality has to be fulfilled:

$$w(u,v) + w(v,x) \geq w(u,x) \tag{1.7}$$

Trees, in general, are connected graphs (comprising of one or more nodes), where two nodes are connected by a unique path. Spanning trees, in turn, are trees covering all vertices of a graph. The number of different spanning trees on a graph with n vertices, T_n, is calculated by Cayley's formula (Cayley (1889), Aigner and Ziegler (2001)):

$$T_n = n^{n-2} \tag{1.8}$$

Worth noting, equation 1.8 does not count the number of abstract tree structures, in which vertices are interchangeable, but the number of all possible trees built on a set of distinct nodes. Straightforwardly, a *minimum* spanning tree is therefore a spanning tree on a weighted graph, with the total weight of all edges being minimal. Analogously, a maximum spanning tree maximizes the sum of edge weights. For a given graph, multiple MSTs can exist with the same overall weight; but, if the weights of the edges are dissimilar, it is unique. The MST for a graph $\mathcal{G}$ will be

denoted by $MST(V)$ and contains the tree edges $E_T \subset E$.
MSTs built on euclidean metric, or EMSTs for short, have some useful properties. Being a subgraph of the so-called Delaunay triangulation (more on this in 1.3.1), an EMST is always planar (i.e., no edges are intersecting) (Shamos and Hoey, 1975) . Also, the maximum degree for each EMST node is five, which can be verified by placing six vertices regularly on a unit circle and connecting them to a node in the center. The result is a valid, but non-unique EMST. If now another vertex would be put on the perimeter, it could not be connected with the center, because the distance to one of the remaining vertices on the circle's circumference has to be smaller. Thus, the center node has a maximum degree of six. However, as each vertex on the outer rim has distance 1 to its left and right neighbor, without loss of generality, at least one can be linked to a neighbor, reducing the maximum degree of the center node to five.

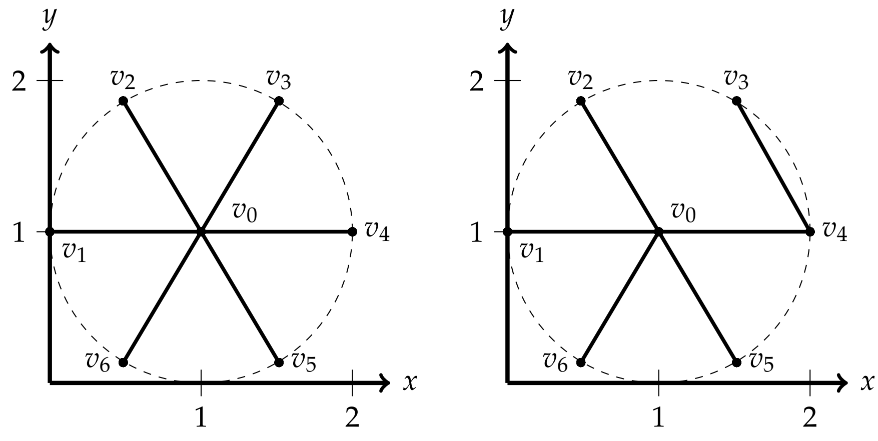

FIGURE 1.1: Left: Non-unique EMST with maximum Degree six.
Right: EMST with maximum Degree five for the same Graph.

By the same reasoning, the enclosed angle between two edges of an EMST can never be smaller than 60°.

1.2 MST Construction

Equipped with these definitions, one can now formulate procedures to build MSTs. On the basis of a greedy approach, three classic algorithms are presented in detail. All four algorithms are variations of the same subject, that is, to color all edges in the graph either blue or red, depending whether they are present or absent in the final MST. This section too is mainly adapted from Tarjan (1983a).

1.2.1 Greedy Approach

The coloring scheme follows directly from the definition of MSTs given above:

Blue Rule

In a cut not being crossed by a blue-colored edge, color the edge associated with the smallest weight blue.

Red Rule

In a cycle with no red edges, color the uncolored edge associated with the largest weight red.

The blue rule states that for any given cut, the MST contains the lightest edge crossing, the red rule eliminates the heaviest edge from any cycle.
Both rules are applied until no edge remains uncolored. The order of appliance does not matter; at every stage, at least one rule is applicable. Furthermore, the blue edges are guaranteed to form an (not necessarily 'the') MST on the graph (Tarjan, 1983a). With this coloring scheme being sufficient to construct MSTs, the subsequent sections explore how the two rules can be implemented efficiently.

1.2.2 Boruvka's Algorithm

Named after its inventor, Czech mathematician Otakar Borůvka (Nešetřil et al., 2001), Borůvka's algorithm involves only a single coloring step:

Coloring Rule

For every blue subtree, select an incident edge of minimal cost, and color it blue.

For each (initially n single-vertex) subtrees, its incident edges need to be checked. At worst, newly colored edges connect subtrees of the same size, thus reducing the number of blue subtrees by half in each iteration. That is why, the coloring step is invoked maximally $log(n)$ times, until one blue tree remains. To find the minimum incident edges, at most m connections have to be visited, leading to a runtime of $\mathcal{O}(mlog(n))$. For applying the coloring rule in an ordered way, Borůvka's procedure requires all edge costs to be distinct: Exemplarily, it could be achieved by sorting the edges first by weight and then by their index in E.
For sparse graphs, there exist more efficient implementations, for example (Cheriton and Tarjan, 1976).

1.2.3 Kruskal's Algorithm

After sorting the edges, perform the following step on the edges in ascending order (Kruskal, 1956):

Coloring Rule

Color the current edge (u, v) red if u and v are part of the same blue subtree, or (u, v) becomes blue.

The algorithm relies on the verification whether u, v already belong to the same subtree. Using the union of disjoint sets (Tarjan, 1983b), these verifications for m edges are completed in overall $\mathcal{O}(m\alpha(m,n))$ steps, where $\alpha(\cdot,\cdot)$ denotes the super-linear, though very slowly growing inverse of an Ackermann-type function. Section 3.2.1 takes a more detailed look on both disjoint sets and and the inverse Ackermann function. Nonetheless, since the number of edges is bounded by $\mathcal{O}(n^2)$, sorting them takes $\mathcal{O}(n^2 log(n))$[1] and dominates the runtime.
In case the edge weights are small integer values, the runtime can be decreased to $\mathcal{O}(m\alpha(m,n))$ by applying radix sort.

1.2.4 Prim's Algorithm

The third algorithm is known by Prim (Prim, 1957), although it has been discovered earlier by Jarnik (see Jarník (1930) for the original publication in Czech and German, and Korte and Nešetřil (2001) for an English translation plus a historical oversight of Jarnik's work) and later republished by Dijkstra (Dijkstra et al., 1959). Jarnik, in turn, developed his algorithm in response to Borůvka's.
Beginning with a random vertex, repeat $n - 1$ times:

Blue Rule
The lightest edge to an incident vertex is colored blue.

Ultraviolet Rule
All incident edges to unvisited vertices are ultraviolet.

Red Rule
Edges to already visited vertices are marked red.

Whereas Borůvka possibly builds multiple subtrees simultaneously, Prim's construction expands a single subtree, until all vertices are covered. The procedure omits sorting all edges, but memorizes for each unvisited vertex the lightest edge leading to it. Thereby, the coloring step can be carried out in $\mathcal{O}(n)$, similarly to depth-first search:
First, find the vertex v with the lightest connection to the subtree in $\mathcal{O}(n)$, color this connection blue, and mark v as visited. Second, the $\mathcal{O}(n)$ outgoing edges from v to the remaining unvisited vertices w are colored ultraviolet. If one of those edges (v, w) is lighter than an existing ultraviolet connection, (u, w), (u, v) becomes red, and (v, w) is stored together with its associated weight under w. Repeat these steps until all vertices are visited.
Doing so results in an overall runtime of $\mathcal{O}(n^2)$, which is why Prim's algorithm is the method of choice for the implementation later on.
When graphs are not fully connected, execution time can be decreased by employing a d-heap data structures, for fast updates and retrieval of minimum elements. The

[1]This can be achieved by any sorting algorithm with a worst-case of $\mathcal{O}(klog(k))$ for k elements, for instance MergeSort (Sedgewick and Wayne, 2011a) or HeapSort (Sedgewick and Wayne, 2011b)

overall number of steps then becomes $\mathcal{O}(mlog_{2+\frac{m}{n}}(n))$, particularly, if the number of edges is in the vicinity of $\Omega(n^{1+\epsilon})$ for $\epsilon > 0$, $\mathcal{O}(\frac{m}{\epsilon})$.

1.2.5 Other Approaches

Beside these three classical algorithms, two more recent approaches noteworthy. Karger et al. (1995) present a randomized algorithm of expectedly $\mathcal{O}(m)$ steps. It furthermore applies to disconnected graphs, resulting in a forest of MSTs. Combining the coloring rule from Borůvka with a randomized selection of edges, its worst-case runtime is the same compared to Borůvka's algorithm.
Chazelle (2000a) discovered a deterministic approach with a runtime of $\mathcal{O}(m\alpha(m,n))$, with $\alpha(\cdot,\cdot)$ being again an inverse Ackermann function. Discussing the algorithm in detail would exceed the scope of this thesis, but the basic idea is to employ a divide-and-conquer strategy: The initial graph is recursively partitioned into contractible subgraphs, on which the MST is subsequently built. In order to select relevant edges, a probabilistic heap structure, called soft heap (Chazelle, 2000b), is employed, guaranteeing a constant amortized cost for deletion, melding of two heaps, and retrieval of the minimum; only the insertion of new elements takes $\mathcal{O}(log(\frac{1}{p}))$, where $0 < p \leq \frac{1}{2}$ is a predefined error rate.

1.3 Related Concepts

EMSTs are closely related to two concepts in computer science, namely Delaunay triangulation and Voronoi diagrams. In the following two subsections, based on De Berg et al. (1999a) and De Berg et al. (1999b) unless stated otherwise, both are briefly revised.

1.3.1 Delaunay Triangulation

As aforementioned, EMSTs for two-dimensional data points are planar, due to the fact that they form a subgraph of the Delaunay triangulation. Delaunay triangulation subdivides a plane containing data points (separated by euclidean distance) into triangles, with the focus on maximizing the inner angles. Such a triangulation is *maximal*: any additional edge would inevitably intersect with another. The Delaunay triangulation is unique, unless no more than three points lie arbitrarily on a circle. If this is the case, Delaunay triangulation constitutes the dual graph to a Voronoi diagram (cf. subsection 1.3.2). Thus, the edges in the Delaunay triangulation always connect nearest neighbors, and an EMST is the minimal set of connections between nearest neighbors. Figure 1.2 shows a Voronoi diagram and its unique Delaunay triangulation, together with the EMST.
In the plane, Delaunay triangulation can be computed in $\mathcal{O}(nlog(n))$. Delaunay triangulation is a useful tool, for instance when it comes to modeling terrains, or for numerical analysis (De Berg et al., 1999a).

1.3.2 Voronoi Diagrams

Voronoi diagrams are the dual graph to Delaunay triangulation. Given a set V of n two-dimensional coordinates, or *sites* (again, in the euclidean metric space), a Voronoi diagram assigns to each data point $v \in V$ a convex polygon, the Voronoi *region*, in which all coordinates are closer to p than to any other point $u \in V \setminus v$. For this reason, the challenge of computing Voronoi diagrams is sometimes called *Post-Office problem*, because it determines the optimal boundaries of catchment areas. The boundaries of the Voronoi region $Vor(v)$ for v are the perpendicular bisectors h between v and the other data points $u \in V$:

$$Vor(v; V) = \bigcap_{u \in V} h(v, u) \tag{1.9}$$

All coordinates on a boundary are equidistant from both sites they are separating. The point, where two boundaries meet, is the center of a circle, on whose circumference at least three sites are located. That is why, Voronoi diagrams are helpful for determining the smallest or largest empty circle in a set of data points.
With regards to complexity, the number of boundaries and vertices is in the vicinity of $\mathcal{O}(n)$ for Voronoi diagrams in the plane.
Chew and Dyrsdale III (1985) offer a more illustrative explanation of the boundaries, describing them as "the places where wave fronts meet" when "n pebbles are dropped simultaneously" at their respective coordinates into a still pond.
Similar to Delaunay triangulation, euclidean Voronoi diagrams can be efficiently computed in $\mathcal{O}(nlog(n))$, for example by using Fortune's algorithm. Besides the euclidean distance, Voronoi diagrams can be constructed for any distance measure, changing the regions' appearance and possibly affecting the runtime. Such a specific diagram will be presented in section 3.1.2.
Having obtained a Voronoi diagram, the Delaunay triangulation can be easily constructed by connecting all sites with adjacent regions. Figure 1.2 visualizes a Voronoi diagram, Delaunay triangulation, and EMST for eight random coordinates.

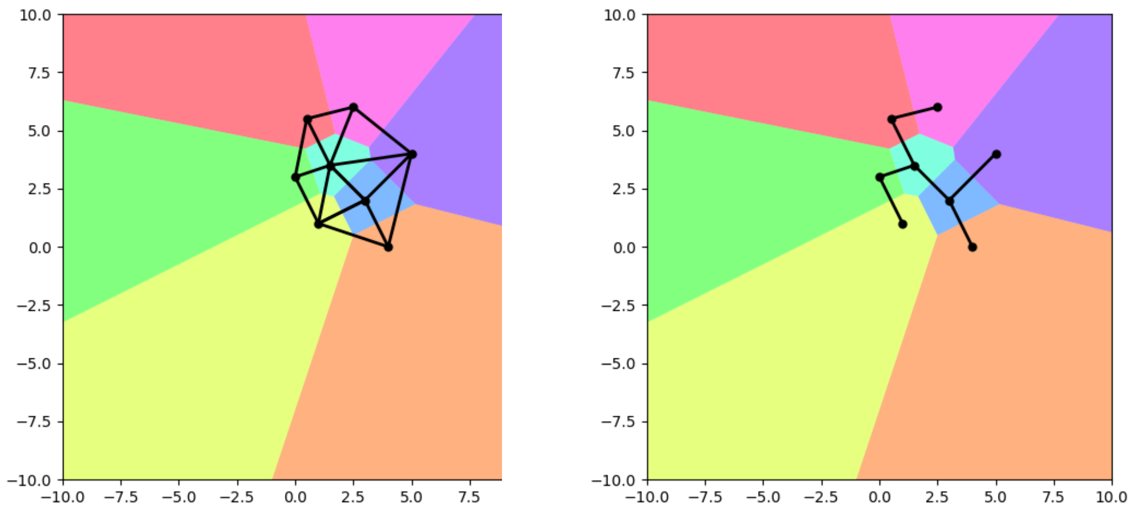

FIGURE 1.2: Left: A Voronoi diagram with its dual Delaunay Triangulation Graph. Right: The same Voronoi diagram with the corresponding EMST.

1.4 Applications of MSTs

The applications of MSTs are manifold. In order to highlight their broad use, four areas are selected.

Traveling Salesman Problem

Already Kruskal (Kruskal, 1956) imagines MSTs to be helpful for solving the *traveling salesman problem* (TSP, for short), where a set of vertices, representing cities, shall be visited once by a single connecting route. In his argument, the TSP constitutes a special case of a non-branching MST, in which the closing edge between the first an last vertex is missing. The general TSP is NP-hard (Korte and Vygen, 2008), as well as the metric TSP (where the edge weights resemble metrical distances). However, for the latter, it can be shown that the optimal solution can be approximated. Christofides (Christofides, 1976) gives an $\mathcal{O}(n^3)$ algorithm based on MSTs to approximate the metrical TSP by a worst-case factor of $\frac{3}{2}$.

Network Design

Apart from route planning, the first published applications of MSTs concern the reduction of wire length in electric circuits. Borůvka uses an MST to plan the electrification of his home region, Moravia (Nešetřil et al., 2001); on a smaller scale, Loberman and Weinberger (1957) efficiently connect terminals in digital circuits. In his extensive survey, Eppstein (Eppstein, 1996) notes how, besides the aforementioned ones, planar, low-degree, and minimum diameter MSTs are useful when optimizing networks.

Data Clustering

Moreover, MSTs can be a starting point for cluster analysis. Gower and Ross

(1969) were among the first to use MSTs to analyze taxonomic relationships. Hierarchical clusters are obtained by grouping the MST edge weights under predefined thresholds.
More recently, Yu et al. (2015) demonstrate how repetitive motifs in MSTs can be identified and what they reveal about the original data.

Data Visualization
Closely related to clustering is the visualization of data. Hurtado et al. (2013) and Hurtado et al. (2018) employ MSTs to illustrate graphs, whose vertices are distributed over two sets. Depending on their set memberships, the vertices are either colored red, blue, or purple, if a node is present in both sets. The goal is form a minimum red-blue-purple spanning graph by connecting blue/ purple, and red/ purple vertices using a minimal sum of edge lengths. This can be approximated with the help of MSTs.
Meulemans et al. (2013) plot MSTs for the most basic visualization of sets of two-dimensional coordinates, successively coloring areas with higher density of points.

1.5 Problem Statement

Given an arbitrary EMST in the plane, this thesis aims to calculate and visualize all EMSTs, while one vertex is moving on a straight trajectory. That kind of problem, not necessarily situated in a euclidean space, is also known by 'moving points' or 'kinetic' MST in the literature (see for instance Fu and Lee (1991) and Agarwal et al. (1998)). Translated to the real world, the task could be to place a harbor alongside a straight canal, and find for each position its optimal connection to an existing set of railroads (idealized by an EMST). To solve it, an algorithm proposed by Monma and Suri (1992) is implemented.
The subsequent paper is organized as follows: In chapter 2, related work is presented. Chapter 3 describes in detail the approach by Monma and Suri, and the last chapter 4 discusses the results and gives ideas for future work.
Pseudo-Code snippets being too elaborate for the main chapters can be found in the appendices A, B and C.

Chapter 2

Related Work

As shown in the previous chapter, MSTs are used in a wide variety of areas, ranging from theoretical concepts to practical applications. Therefore, due to the great number of in-depth publications related to MSTs, this chapter is only meant to be a superficial guide. For more information, the reader is directed to the original publications.

First of all, the informally stated task in 1.5 is formalized: Given a set of two-dimensional coordinates V and an additional point x, compute all emerging EMSTs while x is moving on a linear trajectory through the plane. A linear motion can be modeled by making the dimensional entries v_x, v_y of a vertex v linearly dependent on a continuous time variable $t \in [-\infty, +\infty]$, with arbitrary but fixed $a_{x,y}, b_{x,y} \in \mathbb{R}$:

$$\begin{aligned} v_x(t) &= a_x t + b_x \\ v_x(t) &= a_y t + b_y \\ v(t) &= \begin{pmatrix} v_x(t) \\ v_y(t) \end{pmatrix} \end{aligned} \tag{2.1}$$

For $t = 0$, the result is the EMST on $V \cup x$. The so-called 1-Steiner problem has a similar objective: Additionally to an existing $MST(V)$, a single node x is positioned, such that the new MST on $V \cup x$ is minimal over all choices of x (Georgakopoulos and Papadimitriou (1987), Bose et al. (2020)). In fact, the solution by Monma and Suri (1992), which is implemented here, builds up on the work by Georgakopoulos and Papadimitriou (1987).

To tackle the problem systematically, it shall be dissected into three questions: First, how can changes in trees - edge weight updates, node insertions, or deletions - be efficiently handled? Second, when do changes in the topology occur? And third, what is the lower and upper number of transitions?

One of the first approaches to insert and delete vertices or update edges efficiently in dynamic trees is Sleator and Tarjan (1983), who achieve for each manipulation a runtime of $\mathcal{O}(log(n))$. Frederickson (1985) specifically designs an on-line data structure for MSTs, charging $\mathcal{O}(\sqrt{m})$ per update.

Eppstein (1992) demonstrates how arbitrary insertions and deletions of nodes in two-dimensional EMSTs can be handled in $\mathcal{O}(n^{\frac{5}{6}} log^{\frac{1}{2}}(n))$.

In case the series of k node insertions/ deletions is known beforehand, Eppstein (1994) describes how to handle these in an off-line manner in $\mathcal{O}(log^2(n))$ through repeated reduction and contraction on the graph. During reduction, edges not being involved in any of the k MSTs are discarded; contraction merges vertices that are connected by MST edges. Together, the two steps reduce the number of edges, as well as the number of nodes, leading to the gain in performance. In an on-line set-up, Eppstein (1995) maintains an EMST with an amortized cost of $\mathcal{O}(\sqrt{n}log^2n)$ per update, by reducing the problem to resolving bi-chromatic closest pairs. Henzinger and King (1997) cut the run-time further down to $\mathcal{O}(n^{\frac{1}{3}}log(n))$.
The next question is, when changes in an MST occur. Fu and Lee (1991) make use of Tarjan's method (Sleator and Tarjan, 1983) to calculate all topologies of an EMST in arbitrary dimensionality with the vertices' movement being modeled by a k-th polynomial with respect to t. The idea behind their algorithm is to sort the edges, initially (at $t = 0$) apply Kruskals implementation, and then detect changes in the order of the edges while the points are in motion via a plane-sweep technique. Whenever the position of two edges are altered in the sorted list, the dynamic data structure verifies whether the EMST has changed, too. Doing so results in an overall runtime of $\mathcal{O}(kn^4log(n))$. Since the number of edges m is in the vicinity of n^2 in a fully connected graph, the maximally possible number of intersections is $k \cdot (n^2)^2 = kn^4$, as every edge can intersect any other edge k times. This consideration dates back to Atallah (1985), who proves furthermore $\mathcal{O}(nlog(n))$ for determining the final MST at $t = \infty$. Operating with the same upper bound, Rahmati and Zarei (2012) suggest a similar approach for two-dimensional EMSTs, using a kinetized data structure described by Guibas et al. (1991) for dynamic Delauney triangulation, which takes $\mathcal{O}(log^2(n))$ per update, after $\mathcal{O}(nlog(n))$ preprocessing steps . Such kinetic data structures are also employed in other publications on moving points MSTs, for instance Basch et al. (1997), but their upper bound is below the critical $\mathcal{O}(n^4)$ only under certain conditions. If edge weights in a graph change linearly (not being the case for EMSTs), Agarwal et al. (1998) show how generic MSTs can be maintained through a divide-and-conquer sparsification strategy on the edges in $\mathcal{O}(n^{\frac{2}{3}}log^{\frac{4}{3}}(n))$ steps or, if done in randomized fashion, $\mathcal{O}(n^{\frac{2}{3}}log(n))$. Apart from changes in the order of edges, MSTs are modified if, for the moving nodes, the geographic nearest neighbors alter. That approach is also viable for calculating stricter bounds, and is implemented in the upcoming chapter 3.
Regarding upper bounds, studies on moving points MSTs in the Manhattan (L_1) and Chebyshev (L_∞) metric are most productive. Being piece-wise linear functions, meaning, the edge weights of the underlying graph change linearly, the challenge of determining MSTs can be reduced to other known problems, for instance k-set or geographic nearest neighbors. While the former is concerned about the number of combinatorially distinct ways ways n points in the plane can be divided into subsets of size k via half-spaces (Eppstein, 1998), the latter defines for each given point its closest neighbor in a predefined neighborhood. For nodes actually being

separated by L_1 and L_∞ distance, Katoh et al. (1995) improve the straight-forward upper bound of $\mathcal{O}(n^4)$, when *all* nodes are moving linearly, under the assumption of most one exchange of edges within their order at any given time stamp. Applying a solution for moving nearest neighbors, they determine $\mathcal{O}(n^{\frac{5}{2}}\alpha(n))$ as general upper bound . In case the n nodes move into $c < n$ different directions, this bound becomes $\mathcal{O}(min(c, \alpha(n))c^2n^2)$; if just k vertices move, the maximum number of transitions is in the vicinity of $\mathcal{O}(k^3n)$. By means of a k-set solution, Dey (1998) further refines the unconstrained $\mathcal{O}(n^{\frac{5}{2}}\alpha(n))$ bound to $\mathcal{O}(mn^{\frac{1}{3}})$.
Unfortunately, it is non-trivial to generalize these results to the euclidean distance. Monma and Suri (1992) prove $\mathcal{O}(n^{2d})$ to be the maximum number of transitions for one moving point in a d-dimensional (sub)space, if $d > 2$, and a tight bound of $\mathcal{O}(n^d)$, when $d \leq 2$. The latter is being discussed in the upcoming chapter. Aronov et al. (1994) elaborated these findings to $\mathcal{O}(n^d log^{2d^2-d} n)$ through a more fine-grained subdivision. Allowing all points to move, Katoh et al. (1995) derive an $\mathcal{O}(n^3 2^{\alpha(n)})$ upper bound for transitions in arbitrary dimensions by counting the maximum number of nearest neighbors for each vertex. Being comparable to the previously mentioned $\alpha(\cdot,\cdot)$ in its behavior, $\alpha(\cdot)$ is an alternative representation of an inverse Ackermann function (Cormen et al. (2009a), p. 581). To the knowledge of the author, these are the sole upper bounds determined for the euclidean distance. Thus, attempts are made to either approximate the euclidean metric with piece-wise linear functions (Basch et al., 1997) or the EMST itself (Meulemans et al., 2018).
Concerning lower bounds, publications are sparse. In the euclidean case, Monma and Suri (1992), calculate a lower bound of $\Omega(n^d)$, when a single vertex moves freely in a d-dimensional (sub)space, by proficiently subdividing the vector space into coarse-as-possible areas in each of which $EMST(V \cup x)$ has a distinct topology. Particularly, when a linear, one-dimensional trajectory is chosen for x, a minimum of $\Omega(n)$ areas (i.e., topologies) are passed. For details on this subdivision, the reader is referred to section two of Monma and Suri (1992).
For L_1 and L_∞ metrics, Katoh et al. (1995) note ,"it is easy to construct an example that requires $\Omega(n^2)$ transitions". Without restrictions on the dimensionality, Eppstein (1998) proves the number of transitions to be not growing slower than $\Omega(m\alpha(n))$. Most recently, Eppstein updated his result to $\Omega(mlogn)$ (Eppstein, 2021).

Out of these publications, only Monma and Suri (1992) are explicitly subdividing the space into regions of singular topologies for a given $EMST(V \cup x)$. The next chapter investigates how they achieve this.

Chapter 3

Computing Regional Topologies

In this chapter, the procedure by Monma and Suri (1992) and its implementation for a two-dimensional EMST is laid out. The algorithm consists of two subsequent subdivisions: The first one identifies regions, whose coordinates share the same powerset of non-moving vertices as candidate EMST neighbors. In the second step, these regions are categorized into subregions, each associated with its specific immobile EMST neighbors.Despite its complex appearance, the algorithm can be executed in $\mathcal{O}(n^2)$ for EMSTs of conventional size. The notation is adapted by Monma and Suri (1992) and adopted to the two-dimensional case.

3.1 Initial Subdivision

Similar to the nearest neighbor, generally the adjacent point with the smallest shared distance (cf. 1.3.2), the *geographic* nearest neighbor denotes the closest neighboring point in a specific predefined area (Yao, 1982). Usually, these regions form wedges arranged around a vertex:

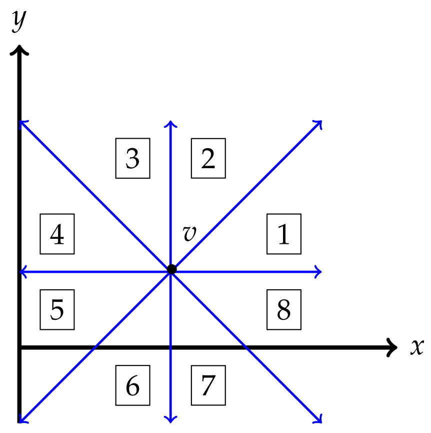

FIGURE 3.1: Yao Graph for a single Vertex v in the Plane

Each of the eight wedges with a corresponding inner angle of 45° around the center vertex denotes a region, for which the nearest neighbors to v are separately computed; the so-called *eight-neighbor-problem* (Yao, 1982). In the next subsection, it will be shown that EMST edges are contained in these regions, if they are sufficiently narrow.

3.1.1 Geographic Nearest Neighbors

Let the plane be characterized by a basis B comprising of the two-dimensional linearly independent basis vectors b_1 and b_2. The convex cone induced by B is given by

$$cone(B) = \{\lambda_1 b_1 + \lambda_2 b_2 \mid \lambda_1, \lambda_2 \geq 0\} \tag{3.1}$$

i.e. all points *within* the cone.
Given an arbitrary coordinate $x \in \mathbb{R}^2$ represented in B,

$$G(x; B) = \{v \in V \mid v - x \in cone(B)\} \tag{3.2}$$

denotes its *geographic set* - in other words, all vertices, which are inside the cone formed by B with x being its origin. Straightforwardly, the *geographic nearest neighbor* in this set, $GN(x; B)$, is the vertex v with the shortest distance to x.
So far, $cone(B)$ covers only a segment of the coordinate system relative to v. In order to extend the notion, a family of bases, $\mathcal{B}$, is introduced, to capture all coordinates of the vector space in a cone:

$$\bigcup_{B \in \mathcal{B}} cone(B) = \mathbb{R}^2 \tag{3.3}$$

The *set of geographic nearest neighbors* (henceforth, GNNs) for x in frame $\mathcal{B}$ is denoted by

$$GN(x; \mathcal{B}) = \{GN(x; B) \mid B \in \mathcal{B}\} \tag{3.4}$$

To get a better picture, see the visualization in 3.2 based on Monma and Suri (1992):

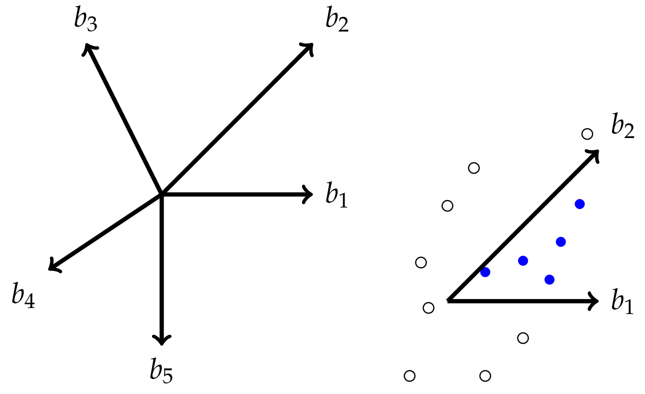

FIGURE 3.2: Left: Possible Bases $B_1 = \{b_1, b_2\}, B_2 = \{b_2, b_4\}, B_3 = \{b_4, b_5\}$, and $B_4 = \{b_4, b_5\}$ and a Frame consisting of $\mathcal{B} = \{B_1, B_2, B_4\}$.
Right: $G(x; B1)$ are marked by blue Dots, other Points by empty Circles.

Edges connecting the vertices of V with their GNNs in $\mathcal{B}$ are

$$E(V;\mathcal{B}) = \bigcup_{v \in V} \{(v,u) \mid u \in GN(v;\mathcal{B})\} \tag{3.5}$$

Let $\theta(B)$ measure the angle between the basis vectors of $b_1, b_2 \in B$:

$$\theta(B) = \arccos\left(\frac{\langle b_1, b_2\rangle}{\|b_1\|\,\|b_2\|}\right) \tag{3.6}$$

Then, the angular diameter of $\mathcal{B}$,

$$Ang(\mathcal{B}) = \max_{B \in \mathcal{B}} \theta(B) \tag{3.7}$$

is the maximum angle between all bases B.
Yao (1982) proves $E(V;\mathcal{B})$ contains the EMST edges of V for a sufficiently small angular diameter. Particularly, if

$$Ang(\mathcal{B}) \leq \sin^{-1}\left(\frac{1}{4}\right) \approx 14.4775^\circ \tag{3.8}$$

then

$$w(x,y) \leq max\,(w(v,x), w(v,y)) \tag{3.9}$$

holds for any two points $x, y \in G(v;B)$ over all $B \in \mathcal{B}$. In other words, the distance between x and y within a cone of v is smaller than their maximum distance *to* v. This condition is sufficient for $EMST(V) \in E(V;\mathcal{B})$:
Suppose there were an edge $(v,y) \in EMST(V) \setminus E(V;\mathcal{B})$, with $y \in G(v;B)$, there would also exist some $x \neq y \in GN(v;B)$, with $(v,x) \in E(V;\mathcal{B})$ and $w(v,x) \leq w(v,y)$. If the edge (v,y) gets deleted, the EMST is split up into two subtrees, where one contains v, and the other x and y. By equation 3.9, x and y *have* to be in the same component, because otherwise, (x,y) would be a shorter edge joining both subtrees than (v,y), violating the assumption of (v,y) being an EMST edge. So, the edge (v,y) in the original EMST can be substituted by (v,x), yielding a spanning tree whose overall weight is not larger than before. Following the procedure until all edges of $EMST(V)$ are from $E(V;\mathcal{B})$ results in a valid EMST, hence proving the proposition (Yao, 1982). For an additional node x, it means that in $EMST(V \cup x)$ it has to be connected to a subset of its GNNs in V.
Furthermore, Yao shows a frame $\mathcal{B}$ satisfying 3.8 can be computed in $\mathcal{O}(1)$. In the implementation later on, the angle is set to 10°, first to cover the total of 360° without remainder, and second to be able to easily verify the results and debug the code.
The task of computing the GNN-graph can be formulated as *oriented Voronoi diagram* (OVD), a special instance of the general Voronoi diagram. Instead of equally expanding on all sides, the regions of an OVD are oriented into a single direction. Particularly, the regions are shaped by the aforementioned cones emerging from each vertex. Within one region, all points are closer to the corresponding vertex than to

any other - i.e., they are geographic neighbors in one base. To cover the plane completely, for each basis $B \in \mathcal{B}$ in counter-clockwise order, an OVD is constructed. Subsequently overlaying the OVDs gives an initial subdivision into polygons of geographic neighbors. A similar approach is presented by Georgakopoulos and Papadimitriou (1987) to solve the 1-Steiner tree problem.
The upcoming two subsections discuss in detail how to compute and overlay the OVDs.

3.1.2 Oriented Voronoi Diagrams

In order to construct OVDs, the neighborhood cone between two bases is redefined in terms of a distance function. σ. Given some vertex v, it should be separated by euclidean distance from any point u within the cone; however, when u lies outside, the distance ought to be infinite. Equation 3.10 captures this idea:

$$\sigma(v,u,B) = \begin{cases} w(u,v) & \text{if } u - v \in cone(B) \\ +\infty & \text{else} \end{cases} \tag{3.10}$$

σ is by definition convex, fulfills the triangle inequality, but is not symmetric.
Monma and Suri (1992) use the algorithm by Chew and Dyrsdale III (1985), which employs a divide-and-conquer strategy to compute the regions. Moreover, they show that every Voronoi diagram based on a convex distance function is planar with star-shaped Voronoi regions (see 3.7), and has $\mathcal{O}(n)$ complexity with regards to edges and vertices. This becomes important later on, when the diagrams are overlaid. While their algorithm runs optimal in $\mathcal{O}(nlog(n))$ (cf. Chew and Dyrsdale III (1985) and Chang et al. (1990) for its optimality), the publication is sparse on details concerning its actual implementation. Therefore, here the technique by Chang et al. (1990) is used, who adapted Fortune's sweepline algorithm designed for euclidean Voronoi diagrams. The pseudo-code is not particularly difficult, but quite lengthy, and thus listed in Appendix A).
Following Chang et al. (1990), two data structures are needed: A queue Q containing the vertices in V (and upcoming intersections of boundaries) sorted in sweepline direction, and a list L storing the active Voronoi regions R_i and their borders B_{ij} for all sites i, j in left-to-right manner. To ensure an overall runtime of $\mathcal{O}(nlog(n))$, L should allow insertions and deletions at any given index in $\mathcal{O}(log(n))$ (see subsections 3.3.2 and 4.2.2 for implementational details). The sweeping direction is determined by the angle bisector of both basis vectors, b_1 and b_2, assuming b_1 being the right- and b_2 being the left-hand side boundary of the cone.
In the main method, Q is instantiated with the data points, its elements are retrieved in the order of their occurrence in sweeping direction.
If the current event is a vertex with coordinates p, the region R_q, in which it is situated, is identified. Any future intersection between the region's left and right boundary has to be deleted from Q due to the expansion of p's region. Next, R_q duplicated,

and the new region R_p and its boundaries B_{qp} and B_{pq} are inserted into L, such that the list entries change from $[R_q]$ to $[R_q, B_{qp}, R_p, B_{pq}, R_q]$. Lastly, possible intersections of B_{qp} and B_{pq} with their left and right neighboring boundaries are added to Q.
If the current event is an intersection of two boundaries, say B_{qr} and B_{rs}, the middle region R_r is closed, and it has to be determined how the border between R_q and R_s succeeds. There are three main cases, depending on the positioning of r and s with respect to the sweeping direction, and two analogous instances, where r and s switch positions. Figures 3.3, 3.4, and 3.5 sketch the three main ones. For a better visability, the inner angle is chosen to be larger than $sin^{-1}\left(\frac{1}{4}\right)$.
When q is at the same height as s, their common bisector becomes the new boundary starting at the intersection of their boundaries B_{qr} and B_{rs}.

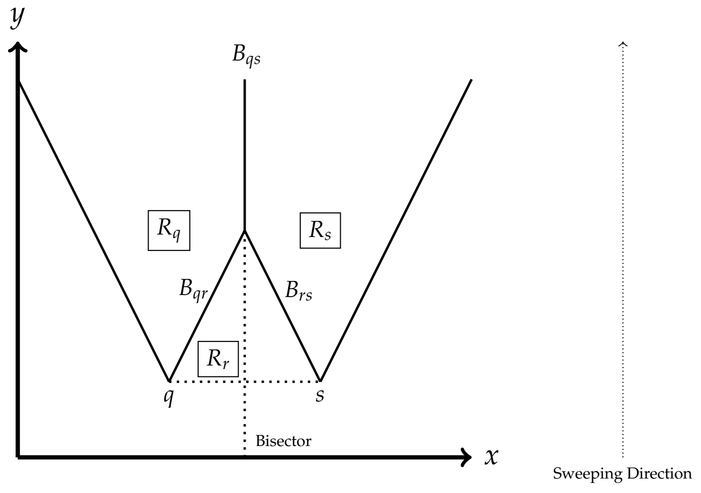

FIGURE 3.3: q and s are at same Height: Bisector is the Boundary

Otherwise, with s coming after q, R_s's left-hand side boundary B_{rs} is continued, unless their common bisector does not intersect; vice versa, when q occurs after s, R_q's left-hand side border B_{qr} continues.

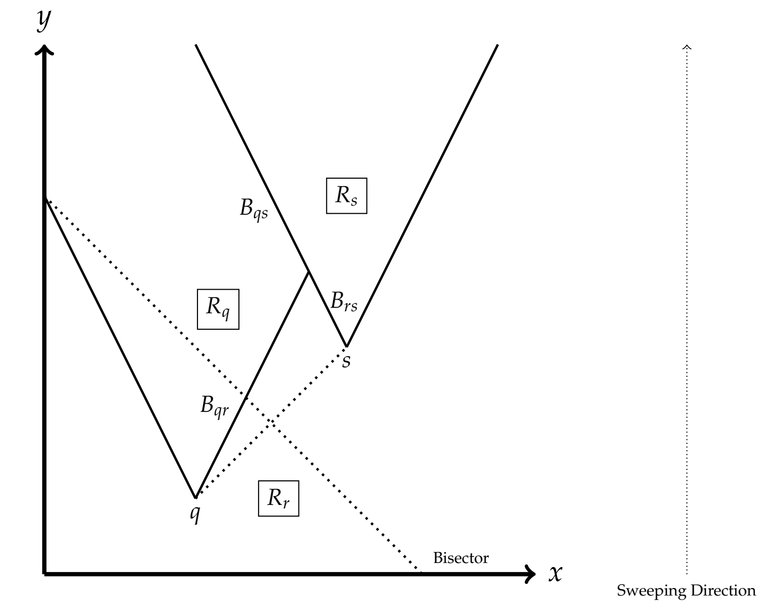

FIGURE 3.4: s comes after q and their Bisector does not intersect

If the bisector of q and s intersects their continued boundary, it becomes the third part of their common border. 3.5 depicts the scenario, again for s being handled after q. Analogously, in case q follows s, first q's left boundary B_{qr} succeeds, until their common bisector takes over. This is in line with the observation by Georgakopoulos and Papadimitriou (1987), who summarize that the border between two points consists at most of three segments.

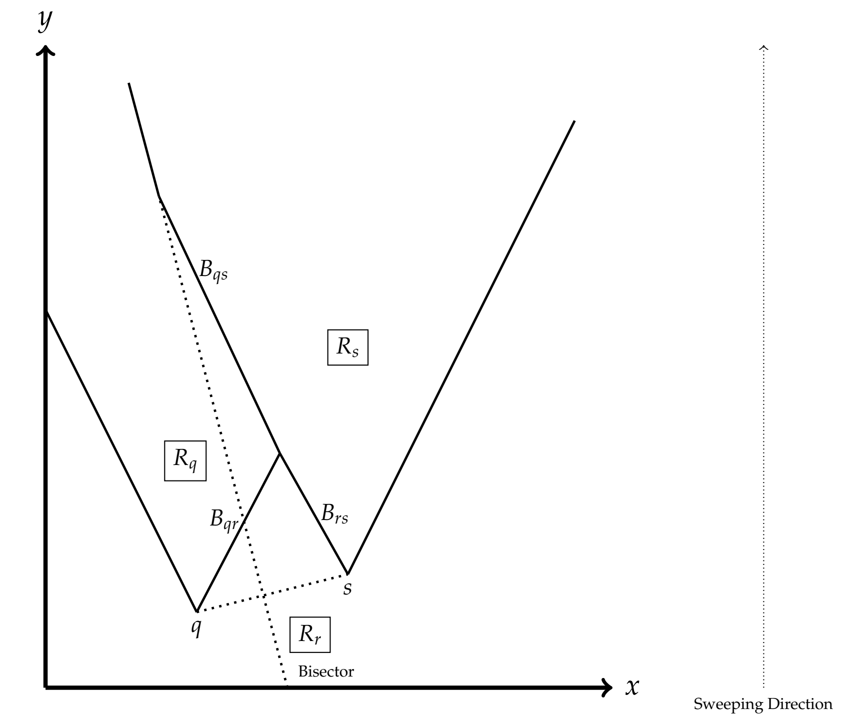

FIGURE 3.5: s comes after q and their Bisector does intersect

To give a complete picture, Fig 3.6 shows an exemplary OVD for the eight points from the beginning. Again, every color stands for another region. After the OVDs are computed, the open regions are clipped along a bounding box constructed from the outmost x- and y-values of boundary intersections. This makes the box significantly larger than the EMST within, because the bases in the OVDs have an inner angle of 10°, so boundaries meet at much further distance.

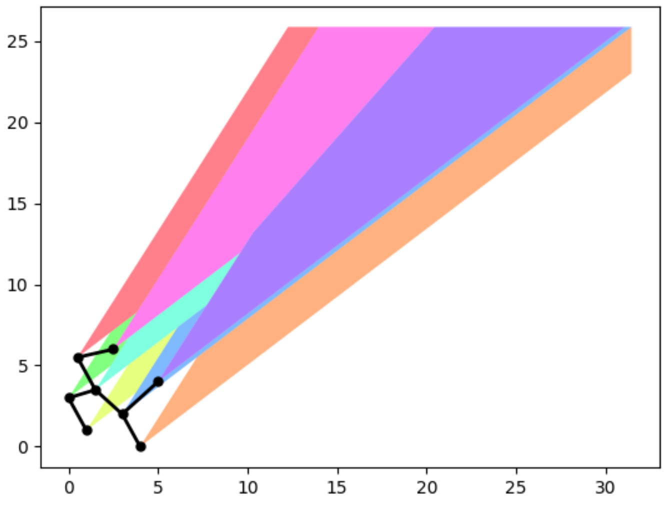

FIGURE 3.6: Exemplary OVD for the eight Vertices

All united OVDs for one vertex are depicted in 3.7. Its star-shaped form is now well visible.

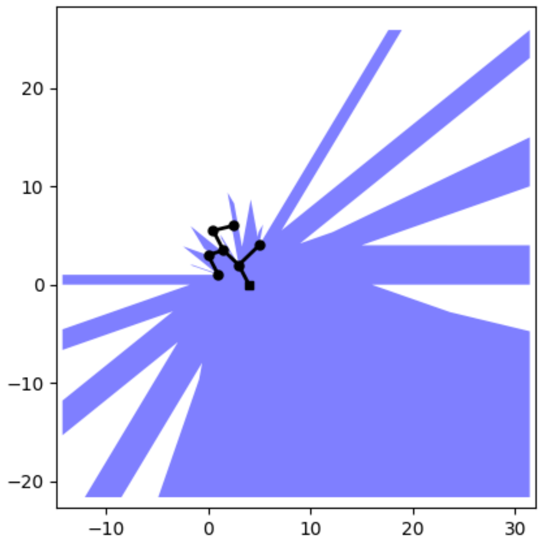

FIGURE 3.7: United OVD for the bottom Vertex (marked by a Square)

3.1.3 Map Overlay

After an OVD has been calculated for each basis in frame $\mathcal{B}$, they are overlaid. Every cell C in the resulting subdivision resembles the set of coordinates $x \in C$, being the GNNs to at most $|\mathcal{B}| = \mathcal{O}(1)$ vertices, whose regions are overlapping. Vice versa, the set of GNNs

$$G(C) = G(x; \mathcal{B}) \subseteq V \tag{3.11}$$

for all $x \in C$ is invariant across C, although the distance measure σ (see 3.10) is not symmetric: If the set of geographic neighbors would not be the same for two coordinates $x, y \in C$, there had to be a vertex $v \in V$ lying in the cone of x and not in the cone of y for some basis. This would imply the existence a basis vector originating in v separating x and y, which is, however, impossible, as x and y are in the same cell. Also, since the boundaries of all OVDs are considered, the ordering of the geographic neighbors remains the same for all coordinates in a cell; thus, the geographic *nearest* neighbors are invariant over one cell. Recalling the findings by Yao (1982), these $G(C)$ are now the candidate vertices possibly linked to an $x \in C$ in $EMST(V \cup x)$. Since the number of OVDs, $|\mathcal{B}|$, is of constant size, and the edges in an OVD are bounded by $\mathcal{O}(n)$ (Chew and Dyrsdale III, 1985), the number of overlaps amounts to $\mathcal{O}(n^2)$.
Envisioning now an additional vertex on a linear trajectory through the cells, at most $\mathcal{O}(n)$ lines (and regions) can be intersected, with each of the $\mathcal{O}(1)$ OVDs containing $\mathcal{O}(n)$ boundaries. With the number of GNNs for every cell being constant, the tight upper bound of EMST topologies, when a single vertex moves along a straight line, is $\mathcal{O}(n)$. Combined with their result on lower bounds of EMST topologies, the number of transitions ranges in the setup between $\Omega(n)$ and $\mathcal{O}(n)$ (Monma and Suri, 1992).
Monma and Suri (1992) do not specify how the overlay can be calculated. They cite Georgakopoulos and Papadimitriou (1987), who prove by triangulating all polygonal Voronoi regions of the $k = |\mathcal{B}|$ OVDs, and overlapping one triangle at a time, a runtime of $\mathcal{O}(k^2 n^2)$ can be achieved. Being a perfectly viable method, its implementation would exceed the scope of the thesis. That is why, the polygonal regions are overlaid with the help of a programming library (cf. 3.3.5) specialized on geometric manipulations. Regarding runtime, the library supposedly takes $\mathcal{O}((n + k) log(n))$ steps, where k denotes the complexity in terms of intersections of the overlay.
Figure 3.8 shows the overlaid OVDs of the eight points from the beginning (see 1.2). Every subset of overlapping sites receives a distinct color, with the original star-shaped polygons still being visible. The plot is truncated to fit the EMST for relation.

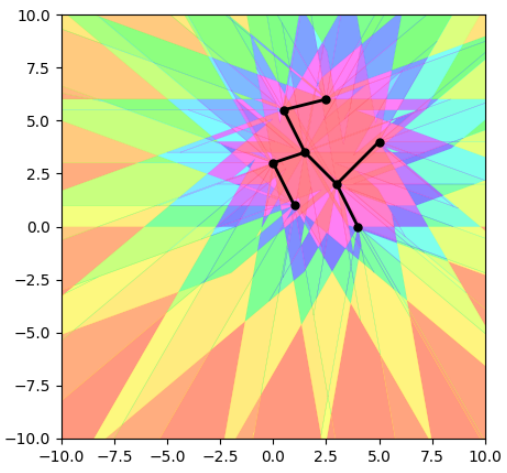

FIGURE 3.8: Overlay of all OVDs for the eight Vertices

3.2 Refined Subdivision

The subdivision so far consists of cells C and provide possible EMST connections to $G(C)$. That means, every C potentially corresponds to

$$2^{G(C)} \leq 2^{|\mathcal{B}|} \leq \sum_{i=1}^{5} \binom{|\mathcal{B}|}{i} \tag{3.12}$$

EMST topologies[1]. The goal of this section is to further subdivide all cells, such that all points x in a refined cell belong to a single configuration of $EMST(V \cup x)$.
To do so, each cell is partitioned into subregions $C(S)$ for subsets $S \subseteq G(C)$ of size five, where coordinates $x \in C(S)$ are solely connected with vertices S in $EMST(V \cup x)$. Monma and Suri (1992) list three criteria for x to be attached to some $s_i \in S$:

Connectivity
$\forall s_i \in S : (x, s_i) \in EMST(V \cup x)$

Non-Connectivity
$\forall s_i \in V \setminus S : (x, s_i) \notin EMST(V \cup x)$

Proximity
$\forall s_i, s_j \in S : w(x, s_i) \leq w(x, s_j)$

Coordinates meeting these conditions shall be situated in $C(s_i, S)$. Then, it holds

$$C(S) = \bigcup_{s_i \in S} C(s_i, S) \tag{3.13}$$

Equipped with these necessities, the procedure to compute the $C(s_i, S)$ looks as follows. The first part repeats the aforementioned initial subdivision of overlaid OVDs.

[1]Under the premise that an EMST node has maximally degree five.

```
def subdivision():
    Compute Overlay of OVDs

    for cell $C$ in Overlay:
        for $S \subseteq G(C)$:
            for $s_i \in S$:
                compute_region($C$, $s_i$, $S$)
```

In the second part, this partition gets refined. Given the amount of cells in the overlay is in the vicinity of $\mathcal{O}(n^2)$, and $G(C)$ is of constant size, the subroutine `compute_region(`C`, `s_i`, `S`)` is called $\mathcal{O}(n^2)$ times:

```
def compute_region($C$, $s_i$, $S$):
    # Overlap cell C with euclidean Voronoi region of $s_i$ w.r.t. S
    $C' = C \cap Vor(s_i; S)$

    # Draw edge from x to $s_i$
    $\mathcal{G}' = EMST(V) \cup (x, s_i)$

    # $\mathcal{G}'$ contains cycles by joining $(x,s_j)$, $s_j \in G(C) \setminus s_i$
    Find longest edges in the resulting cycles with length $r_j$

    # $U(s_j,r_j)$ denotes a circle centered at $s_j$ with radius $r_j$
    $C(s_i,S) = C' \bigcap_{s_j \in S \setminus s_i} U(s_j,r_j) \bigcap_{s_j \in G(C) \setminus S} \overline{U}(s_j,r_j)$
```

Following Monma and Suri (1992), the intersection of C with the Voronoi region of s_i with respect to the other vertices in S ensures the proximity criterion, while overlays of circles/ circle complements guarantee the connectivity requirements. Exploiting the cycle-property, by which any edge additionally drawn in the EMST has to be the heaviest connection in the resulting cycle, $x \in C(s_i, S)$ lies within the circle of radius r_j around s_j, if $s_j \in S$, and outside the circle of radius r_j around s_j, when $s_j \notin S$. In other words, if x is connected to some $s_j \in S$ via (x, s_j), x has to be closer to s_j than the length r_j of the longest edge in a resulting cycle, and therefore in the circle $U(s_j, r_j)$. Vice versa, if x should *not* be attached to some $s_j \in G(C) \setminus S$, it has to be farther away than the length of the longest edge in a cycle, thus being situated *outside* $U(s_j, r_j)$, and within its complement $\overline{U}(s_j, r_j)$. Monma and Suri (1992) claim that by exploring all possible combinations of subsets, together with overlapping the circles and their complements, a complete subdivision is achieved.

Both the computation of the Voronoi region, as well as the overlay of circles can be carried out in constant time, because $G(C) = \mathcal{O}(1)$. Practically, `compute_region` is again realized with the help of the same libraryalready used for the overlay.

However, there are problems with the approach (see 3.2.2). For a better understanding, the remaining algorithm is laid out, before errors are addressed.

3.2.1 Least common Ancestors

To detect the longest edges in a cycle, Monma and Suri (1992) first restructure the EMST into a binary *edge tree*, whose root and intermediate nodes resemble EMST edges, and whose leaves nodes are the vertices of the EMST. The edge at root position is the longest edge in the EMST, and its children are the longest edges in the two subtrees obtained by deleting the originally longest edge stored in root. Clearly, the edge tree consists of $n + (n-1) = 2n - 1 = \mathcal{O}(n)$ nodes. Finding the longest edge between two EMST nodes then becomes the task of finding the least common ancestor in the edge tree. Monma and Suri (1992) use the least common ancestor search by Harel and Tarjan (1984) with $\mathcal{O}(n)$ preprocessing and $\mathcal{O}(1)$ query time to find the longest common edge. Since the algorithm by Harel and Tarjan (1984) would exceed the scope of this thesis, a less efficient, but much simpler off-line approach by Tarjan (Tarjan (1979), Tarjan (1983b), Cormen et al. (2009a)) is implemented.
In this project, the construction of the edge tree takes in the worst case $\mathcal{O}(n^2)$ steps: After the deletion of the longest edge from a subtree, both roots of the two resulting subtrees need to be found, which takes at most $\mathcal{O}(n)$ for each of the $n-1$ edges. There sure are more efficient ways to accomplish this task, but the described method fits elegantly into the implemented tree data structure, and does not impair the theoretical runtime of overall $\mathcal{O}(n^2)$.
On the edge-based tree, the least common ancestor (Tarjan (1979), Cormen et al. (2009a) p.584) is calculated via union of disjoint sets represented by a tree/forest structure (Tarjan, 1983b). The goal is to efficiently test whether two elements belong to the same set by examining if two elements have the same root in the tree: At the beginning, each node in the edge tree is instantiated as root of its own set, and its own ancestor. Whenever two vertices are found to be descendants of the same (intermediate, or root) edge node, their disjoint sets are united under their common ancestor. The unification follows two heuristics: *Union by rank* and *path compression*. While the former ensures that subsets with lower tree structures are linked under subset of higher depth, the latter guarantees vertices to be directly grouped under the root. Both methods aim to reduce the number of steps to determine the common parent set. Figures 3.10 and 3.9 describe the mechanisms.

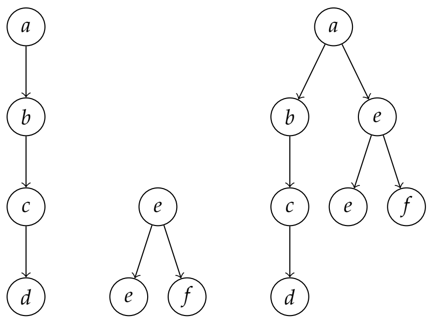

FIGURE 3.9: Sketch of Union-by-Rank

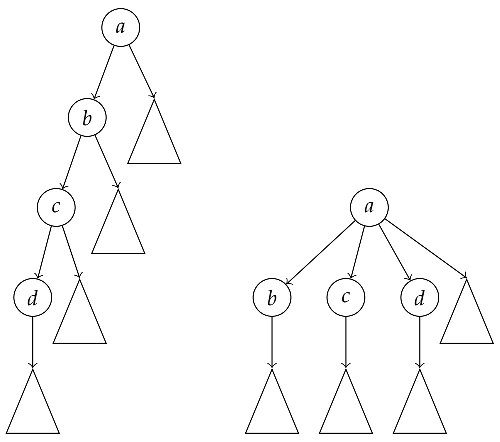

FIGURE 3.10: Sketch of a Set-Structure before and after Path Compression

In pseudo-code, the heuristics look the following (Cormen et al. (2009a), p. 571 and p.584): For every vertex, `make_set` initializes a *set node*, comprising of a parental set node, its rank, its ancestor in the edge tree, and an indicator if it has already been visited. The distinction between `parent` and `ancestor` is important: `parent` refers to the parent in the set tree structure, whereas `ancestor` denotes the ancestor in the edge tree.

```
def make_set(v):

    v.parent = v
    v.rank = v

    v.ancestor = None
    v.visited = False
```

Method `find_set` formalizes the path compression: While searching for the root of the set structure v is part of, it links all intermediate set nodes on its way to the root, too.

```
def find_set(v):
    if v ≠ v.parent:
        v.parent = find_set(v.parent)

    return v.parent
```

`union` and `link` are the straightforward implementation of *union by rank*.

```
def union(v, u):
    link(find_set(v), find_set(u))

def link(v, u):
    if v.rank > u.rank:
        u.parent = v
    else:
        v.parent = u

        if v.rank == u.rank:
            u.rank = u.rank + 1
```

Finally, the least common ancestors can be computed. The method `lca(u)` is called once with the root of the edge tree, and then walks top-down depth-first through the tree. Along its way, the longest edges between EMST vertices u, v are stored in the two-dimensional array `ancestors`.

```
def lca(u):
    make_set(u)
    find_set(u).ancestor = u

    for each child v of u in the edge tree:
        lca(v)
        union(u, v)
        find_set(u).ancestor = u

    u.visited = True

    for each v ∈ V:
        if v.visited:
            ancestors[u, v] = ancestors[v, u] = find_set(v).ancestor
```

For each parental node, a set is initialized and recursively united with its children's sets. As the rank of v could be larger than u's, u is possibly linked under v, and the ancestor of u's root needs to be set to u - otherwise, the structure of the edge tree would be mismatched. Afterwards, u is marked being visited, and, for every other visited vertex $v \in V$ in the EMST, the common ancestor is determined by the ancestor of v's root. That is, because v has already been visited either at the same level or further up in the edge tree, and so its root's ancestor returns the correct node of which u and v are descendants. Overall, the ancestor for every pair of vertices is computed once, since `lca()` visits every u only one time, and marks u accordingly. Tarjan (Tarjan (1979), Tarjan (1983b), Cormen et al. (2009a)) proves that k invocations of `make_set`, `union`, and `find_set` in arbitrary order, with l `make_set` calls, can be computed at worst in $\mathcal{O}(k\alpha(l))$, $\alpha(\cdot)$ being the inverse Ackermann function. This implies a runtime of $\mathcal{O}(n^2\alpha(n))$ for `lca` on the edge tree, comprising of $2n - 1 = \mathcal{O}(n)$ `make_set` and `union` operations, and $n^2 + n = \mathcal{O}(n^2)$ invocations of `find_set`. Although being slower than the $\mathcal{O}(n^2)$ implementation of Monma and Suri, for a reasonable number of vertices the quadratic runtime will not be exceeded:

$$\alpha(n) = \begin{cases} 0 & \text{if } 0 \leq n \leq 2 \\ 1 & \text{if } n = 3 \\ 2 & \text{if } 4 \leq n \leq 7 \\ 3 & \text{if } 8 \leq n \leq 2047 \\ 4 & \text{if}^2\ 2047 \leq n \leq 10^{80} \end{cases} \tag{3.14}$$

For an extensive proof, the interested reader is directed to Cormen et al. (2009a).

3.2.2 Errors in the Algorithm

Unfortunately, the rationale behind `compute_region(`C`, `s_i`, `S`)` is flawed. First, the intersection with the Voronoi region of s_i given the nodes in S does not result in a proper subdivision of cell C. In case $|G(C)| > 1$, there are multiple subsets $S \subset$

[2]Actually, n is equal or less than a number *much* larger than 10^{80}.

$G(C)$ containing single elements. However, the Voronoi region $Vor(s_i; S)$, where $S = \{s_i\}$, is a single region indefinitely spanning the plane. Thus, when there are multiple singleton sets $S \subset G(C)$, say $S' = \{s_i\}$ and $S'' = \{s_j\}$ with $s_i \neq s_j$, $C \cap Vor(s_i; S') = C \cap Vor(s_j; S'')$, and the regions of topologies $C(s_i, S')$ and $C(s_j, S'')$ can possibly overlap. A fix to this inaccuracy is to change the line

$$C' = C \cap Vor(s_i; S) \tag{3.15}$$

to

$$C' = C \cap Vor(s_i; G(C)) \tag{3.16}$$

which gives a consistent subdivision of each cell into non-overlapping sub-cells C', and satisfies the proximity criterion. The adjustment does not affect the proclaimed runtime, because $VOR(s_i; G(C))$ also has a constant number of borders. So, the pseudo-code becomes

```
def compute_region($C$, $s_i$, $S$):
    # Overlap cell C with euclidean Voronoi region of $s_i$ w.r.t. V
    $C' = C \cap Vor(s_i; G(C))$

    # Draw edge from x to $s_i$
    $\mathcal{G}' = EMST(V) \cup (x, s_i)$

    # $\mathcal{G}'$ contains cycles by joining $(x, s_j)$, $s_j \in G(C) \setminus s_i$
    Find longest edges in the resulting cycles with length $r_j$

    # $U(s_j, r_j)$ denotes a circle centered at $s_j$ with radius $r_j$
    $C(s_i, S) = C' \bigcap_{s_j \in S \setminus s_i} U(s_j, r_j) \bigcap_{s_j \in G(C) \setminus S} \overline{U}(s_j, r_j)$
```

Now, each cell is fully subdivided by overlaying circles and their complements, such that every coordinate is attributed to a single topology. Assume $G(C)$ to comprise of arbitrary vertices $\{u, v, w\}$. Without loss of generality, let the focus be on u's part of C, $C' = C \cap Vor(u; G(C))$. The set of subsets containing u is

$$\mathcal{S}_u = \{\{u\}, \{u, v\}, \{u, w\}, \{u, v, w\}\} \tag{3.17}$$

and for each subset $S \in \mathcal{S}_u$, there exists a unique division of C':

$$\begin{aligned} \{u\} &:= C' \cap \overline{U}(v, r_v) \cap \overline{U}(w, r_w) \\ \{u, v\} &:= C' \cap U(v, r_v) \cap \overline{U}(w, r_w) \\ \{u, w\} &:= C' \cap U(w, r_w) \cap \overline{U}(v, r_v) \\ \{u, v, w\} &:= C' \cap U(v, r_v) \cap U(w, r_w) \end{aligned} \tag{3.18}$$

As every logical combination of circles and their complements is present, C' is now completely partitioned.

The second issue arises during the subsequent intersection of circles and circle complements. While for most regions the correct topology is calculated, occasionally,

a small fraction is false. The false topologies are always the result of a erroneously refined subdivision, because they never contain vertices not being part of the GNNs. Both the weight differences of the produced and the optimal trees, as well as the areas of the false sub-cells are large enough, for why precision errors can be ruled out. Alas, the problem could not be resolved by the author and is left for discussion in 4.1.1. In order to retain a usable algorithm, the topology of every region is checked for correctness, and changed if necessary. More precisely, the centroid of each cell is representatively selected, and then subsequently connected to all subsets of the cell's GNNs. By deleting the longest common edges between the vertices in the subsets, a tree structure is enforced. Afterwards, the subset associated with lowest tree weight is chosen to be the topology for the whole cell.
This step is solely of cosmetic nature; it does *not* recalculate the region itself, only its attributed topology. It is meant to retain the overall usability of the program, by accepting limitations in certain regions. To be fully transparent, the errors are part of the visualization (cf. 3.3.6). Both the number of subsets, as well as their size, is constant, so the correction does not add any costs to the overall computation.
On the upside, though not being minimal, the produced trees are always planar, which might be favorable in some applications. Since every cell is intersected with the Voronoi regions of its GNNs, edges to nodes being too far away and connected to other parts of the tree are therefore prohibited.

3.3 Implementational Details

In this section, details of the program that have so far been omitted are presented. All functions are implemented in `python3.8`[3]. For basic operations, `numpy`[4] (Harris et al., 2020), version `1.17.4`, and `scipy`[5] (Virtanen et al., 2020), release `1.7.1`, are imported.
Six aspects shall be highlighted: Sorting, list operations, orientation of points with respect to lines, finding intersections, geometric manipulations, and, lastly, how the results are visualized.

3.3.1 Sorting

Be it the descendingly in case of the edges for the edge tree, or ascendingly for the sweepline approach to compute OVDs, sorting is one of the fundamental operations of the project. To account for various objects (edges, coordinates), modi (descending, ascending), compare measures, and the dynamic set-up (online insertion/ deletion of elements), a customized priority queue based on a binary heap is implemented (Sedgewick and Wayne, 2011b).
The idea is to maintain a balanced binary tree, where $(key, \ , object)$ tuples are stored,

[3]`https://www.python.org/downloads/release/python-380/`, lastly visited June 12, 2022
[4]`https://numpy.org/`, lastly visited June 12, 2022
[5]`https://scipy.org/`, lastly visited June 12, 2022

with the highest priority key being in root position. Such a tree can easily be represented in an array `Q`, where the root is at index 1, and the two children of a parent at n are at indices $2n$ and $2n+1$. It holds that in a maximum (minimum) priority queue, the children's keys are always smaller (larger) than the parent. An efficient implementation relies on three methods: `swim` and `sink`, moving nodes up or down depending to their priority, and `exchange`, which swaps the indices of two nodes. Prioritization is done by a compare method `comp`(i, y), stating if $Q[i] \leq Q[j]$ or $Q[i] > Q[j]$ for some indices i and j.

As long as the key at index i is prioritized, it *swims* towards the root by changing positions. Indices can only be integers, so the floor-function is applied.

```
def swim(i):
    while $i > 1$ and comp($\lfloor \frac{i}{2} \rfloor$, i):
        exchange($\lfloor \frac{i}{2} \rfloor$, i)
        $i = \lfloor \frac{i}{2} \rfloor$
```

`sink` performs the opposite motion. While the key at i has lower priority, it moves further down the tree. Additionally, its status needs to be compared with the sibling to preserve the order.

```
def sink(i):
    while $2 \cdot i \leq$ Q.size:
        $j = 2 \cdot i$
        if $j <$ queue.size and comp(j, $j+1$):
            $j = j+1$
        if not comp(i, j):
            break
        exchange(i, j)
        $i = j$
```

The implementation of `exchange` is straightforward:

```
def exchange(i, j):
    tmp = Q[i]
    Q[i] = Q[j]
    Q[j] = tmp
```

After having defined these methods, the functionality of `Q` can be completed. New keys are inserted by increasing the current size, giving the new key the lowest priority, and let it rise up.

```
def insert(key):
    Q.size = Q.size + 1
    Q[Q.size] = key
    swim(Q.size)
```

When the element of highest priority is retrieved, the key with the lowest status is teleported to the root, and then sunken to its new position.

```
def pop():
    root = Q[1]
    exchange(1, Q.size)
    Q[Q.size] = None
    Q.size = Q.size − 1
    sink(1)

    return root
```

deleting the key at index i works analogously:

```
def delete(i):
    key = Q[i]
    exchange(i, Q.size)
    Q[Q.size] = None
    Q.size = Q.size - 1
    sink(i)

    return root
```

Doing so maintains a fairly balanced tree and guarantees a worst-case costs of $\mathcal{O}(log(n))$ for insertion and deletion, and $\mathcal{O}(1)$ for peeking at the current minimum/ maximum element, when no deletion is desired.

3.3.2 List Operations

For the optimal runtime of the OVD algorithm, an $\mathcal{O}(log(n))$ execution of both finding the correct index and inserting/ deleting the region or boundary object in the left-to-right sorted list is essential. To achieve the former, a binary search is implemented, which minimizes the interval where the index is located, depending on whether the region or boundary lies to the left or right of the pivot element (see the upcoming subsection). Regarding the latter, the python module `blist` is employed[6]. Being part of the official python package index, it converts any ordinary python list into a `blist` in constant time, and performs, among other operations, insertion and deletion in $\mathcal{O}(log(n))$ with a "hybrid array/tree structure"[7].
4.2.2 presents an alternative approach to be implemented in future versions.

3.3.3 Orientation of Coordinates

Knowing if a point in the plane is to the left or right of a line segment is crucial for the computation of the OVDs. Let p_1 be the coordinate, whose orientation with respect to segment $\overline{p_0p_2}$ ought to be tested. Two constellations are possible:

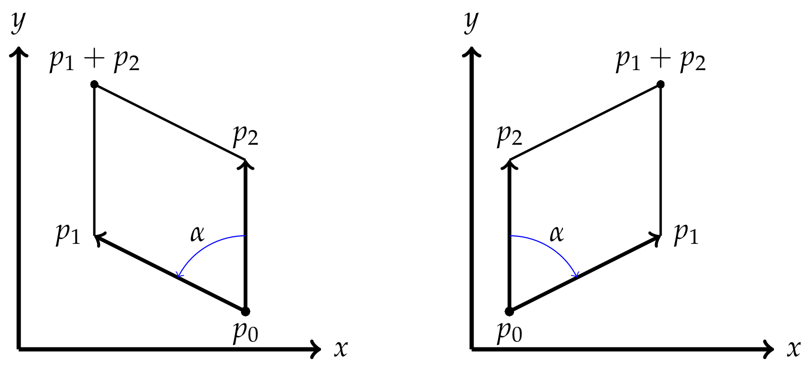

FIGURE 3.11: Left: p_1 being to left/ in counterclockwise Direction to p_2. Right: p_1 being to the right/ clockwise to p_2.

[6] `https://pypi.org/project/blist/`, lastly visited June 12, 2022
[7] `http://stutzbachenterprises.com/blist/blist.html`, lastly visited June 12, 2022

The signed cross product (Cormen et al., 2009b)

$$
\begin{aligned}
(p_1 - p_0) \times (p_2 - p_0) = \\
&= (p_{1_x} - p_{0_x}) \cdot (p_{2_y} - p_{0_y}) \\
&- (p_{1_y} - p_{0_y}) \cdot (p_{1_x} - p_{0_x})
\end{aligned}
\tag{3.19}
$$

calculates the area of the parallelogram $\langle p_0, p_1, p_1 + p_2, p_2 \rangle$. An alternative approach uses the determinant of the matrix

$$
\begin{vmatrix} (p_1 - p_0)_x & (p_2 - p_0)_x \\ (p_1 - p_0)_y & (p_2 - p_0)_y \end{vmatrix}
\tag{3.20}
$$

If the outcome is positive, the segment $\overline{p_0 p_1}$ lies to the right (clockwise, $0 < \alpha < 180°$) of $\overline{p_0 p_2}$; when negative, it is situated to the left (counterclockwise, $0 > \alpha > -180°$). If the cross product/ determinant is zero, the segments $\overline{p_0 p_1}$ and $\overline{p_0 p_2}$ are collinear. Rarely, when coordinates are close to each other, or the enclosed angle α is very small, the results can be error-prone. In these cases, a binary search on the cannot return the index of a region or boundary. The only remedy is to traverse the list of active regions and boundaries completely to find that position for which the cross product between a region and adjacent boundaries is minimal. Although on average $\mathcal{O}(nlog(n))$ can be maintained, it leads to a worst-case of $\mathcal{O}(n^2)$ for the OVD algorithm.

3.3.4 Detection of Intersections

Another important subroutine is the identification of intersections. To solve it efficiently, the approach by Antonio (1992) is implemented.
Given a line segment $\overline{p_1 p_2}$, every point p can be represented by the linear combination

$$
p = p_1 + \alpha(p_2 - p_1) \tag{3.21}
$$

with $\alpha \in [0, 1]$. If the line is an infinite ray, starting at p_1 and crossing p_2, this assumption is dropped. That accounts for most boundaries encountered during the sweepline procedure, when an endpoint is not yet determined.
Thus, an intersection p_i of two line segments $\overline{p_1 p_2}$ and $\overline{p_3 p_4}$ fulfills the two equations

$$
p_i = p_1 + \alpha(p_2 - p_1) \tag{3.22}
$$

$$
p_i = p_3 + \beta(p_4 - p_3) \tag{3.23}
$$

By subtracting 3.23 from 3.22 results in

$$
(p_1 - p_3)_x + \alpha(p_2 - p_1)_x + \beta(p_3 - p_4)_x = 0 \tag{3.24}
$$

$$
(p_1 - p_3)_y + \alpha(p_2 - p_1)_y + \beta(p_3 - p_4)_y = 0 \tag{3.25}
$$

For the sake of readability, the tree individual subtractions are replaced:

$$\begin{aligned} a &= (p_2 - p_1) \\ b &= (p_3 - p_4) \\ c &= (p_1 - p_3) \end{aligned} \tag{3.26}$$

By doing so, equation 3.24 is resolved for α and β:

$$\begin{aligned} \alpha &= \frac{-c_x - \beta b_x}{a_x} \\ \beta &= \frac{-c_x - \alpha a_x}{b_x} \end{aligned} \tag{3.27}$$

These outcomes are then each plugged into 3.25

$$\begin{aligned} c_y + \left(\frac{-c_x - \beta b_x}{a_x} \right) a_y + \beta b_y = 0 \quad &\Leftrightarrow \\ a_x c_y - a_y c_x - \beta a_y b_x + \beta a_x b_y = 0 \quad &\Leftrightarrow \\ \beta (a_x b_y - a_y b_x) = a_x c_y - a_y c_x \quad &\Leftrightarrow \\ \beta = \frac{a_x c_y - a_y c_x}{a_x b_y - a_y b_x} \end{aligned} \tag{3.28}$$

$$\begin{aligned} c_y + \alpha a_y + \left(\frac{-c_x - \alpha a_x}{b_x} \right) b_y = 0 \quad &\Leftrightarrow \\ b_x c_y + \alpha a_y b_x - b_y c_x - \alpha a_x b_y = 0 \quad &\Leftrightarrow \\ \alpha (a_y b_x - a_x b_y) = b_y c_x - b_x c_y \quad &\Leftrightarrow \\ \alpha = \frac{b_y c_x - b_x c_y}{a_y b_x - a_x b_y} \end{aligned} \tag{3.29}$$

Since both denominators are the same, the overall calculation comprises just of nine additions and six multiplications at worst. Also, when dealing with closed segments, i.e. $\alpha, \beta \in [0, 1]$, the following pseudo-code rules out contrary cases:

```
def intersects(denominator, enumerator):

    # Testing for Collinearity
    if denominator == 0:
        return False

    if denominator > 0:
        if enumerator < 0 or enumerator > denominator:
            return False
        elif enumerator > 0 enumerator < denominator:
            return False
    return True
```

To determine the intersection, α and β can now be put back in the original formula. Additionally, a bounding box test is put in front, which checks by means of start and end coordinates of segments `s1` and `s2`, whether an intersection can occur at all. The method inspects if the bounding boxes of both segment overlap; a necessary, but not sufficient condition for an intersection to happen. Figure 3.12 illustrates the idea. Antonio (1992) notes that, though the bounding-box test charges additional operations, experiments show it can save up to 20% time. The code is omitted here for the sake of brevity, but can be found in Appendix B.

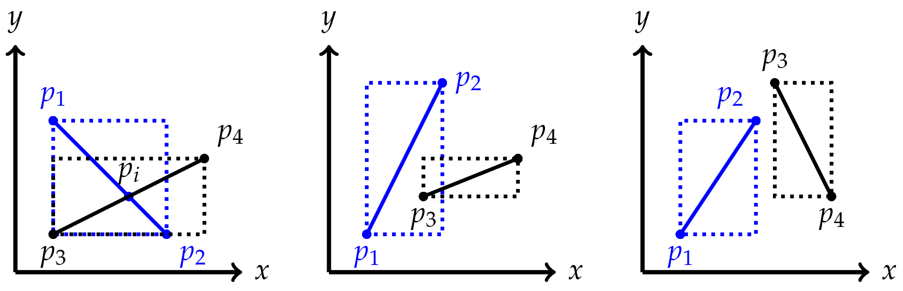

FIGURE 3.12: Left: Bounding Boxes overlap, and the Segments intersect.
Middle: Bounding boxes overlap, but the Segments do not intersect.
Right: Bounding Boxes are disjoint, and the Segments do not intersect.

Intersections are, similarly to the orientation tests, rarely prone to imprecisions. In some scenarios, the calculated intersection for the same two lines can vary by a small margin. This affects especially the deletion of intersections from the the event queue in the sweepline algorithm. Normally, by using hashing, deleting entries would take $\mathcal{O}(1)$. With the caveat of imprecise intersections, every time a to-be-deleted intersection is not found in the queue, all its $\mathcal{O}(n)$ entries need to examined, and, if one is close enough (euclidean distance < 0.01), it is deleted nonetheless. The sweepline execution is degraded to $\mathcal{O}(n^2)$ in worst case. Notwithstanding, the average case remains at $\mathcal{O}(nlog(n))$.

3.3.5 Geometric Manipulations

In order to compute the various geometric tasks - from the transformation of boundaries into OVD polygons, to their overlay, and refined subdivision - the `shapely` (version 1.7.1)[8] (Gillies et al., 07) and `geopandas`[9] (Jordahl et al., 2020) (release 0.10.2+57.gcb0c8b5) libraries are employed. `shapely` bundles functions written specifically for planar geometries in `C++` from `Geos` GEOS contributors (2021), which is a port of the Java Topology Suit. `geopandas`, in turn, combines `shapely` with `pandas` data types. Concrete information on the implemented overlay algorithms is sparse, and the publicly available implementations are unfortunately not well-documented regarding the implemented methods[10], either. The manual of `Java Topology Suit` (Davis and Aquino (2004), p. 25) suggests that the procedure to overlay geometries

[8]`https://shapely.readthedocs.io/en/stable/manual.html`, lastly visited June 12, 2022
[9]`https://geopandas.org/en/stable/`, lastly visited June 12, 2022
[10]`https://github.com/shapely/shapely/blob/main/shapely/set_operations.py`, lastly visited June 12, 2022
`https://github.com/geopandas/geopandas/blob/main/geopandas/tools/overlay.py`, lastly visited June 12, 2022

is similar to the one presented in De Berg et al. (1999c). They achieve for two subdivisions S_1 and S_2 the union $S_1 \cup S_2$, intersection $S_1 \cap S_2$, and geometric difference $S_1 \setminus S_2$ in $\mathcal{O}((n+k)log(n))$, where n is total number of nodes in S_1 and S_1, and k denotes the complexity of the overlay in terms of intersections between S_1 and S_2 .
While intersecting single objects (for example in snippet 3.2) works fine in `shapely`, the overlay algorithm of `geopandas` takes in practice too much time, especially with a larger number of regions [11]. Two reasons could be found: `shapely` distinguishes critically between geometrical objects, such as `LineStrings`, `Polygons`, and `Multipolygons`. Although all oriented Voronoi regions are of type `Polygon`, their intersections and differences may result in other objects, whose types need to be checked individually. Additionally, it is necessary to buffer objects to straighten out small self-intersections or holes in the polygons, caused by internal precision errors. Especially the geometric difference method relies on valid polygons for an accurate performance, whereas the intersection and union operations are rather robust.However, these measures slow the program down, already for a minimum number of two input points.
To improve the runtime practically without exceeding the scope of the thesis, the overlay is computed with a theoretically slower, but demonstrably faster approach, where every initial subregion is calculated individually (see snippet in Appendix C). First, for all input vertices, the OVD regions are united in $\mathcal{O}(n(n+k)log(n))$, resulting in the characteristic star-shaped polygons (cf. figure 3.7). Then, for every subset of V with size $1, 2, \ldots, |\mathcal{B}|$, the the united regions of the nodes in questions are intersected. Afterwards, the union of remaining vertices *not* being in the subset has to be subtracted. Otherwise, the overlay would contain areas with incorrect GNNs.
The procedure avoids the aforementioned intermediate checks of data types and tedious buffering, as the intersection and union function work reliably on various input data - only once per subregion, a buffer becomes necessary, right before the set-theoretic subtraction.
Overall, $\mathcal{O}(n^3)$ intersections and unions ($\mathcal{O}(n^2)$ possible overlaps, multiplied by $\mathcal{O}(n)$ for additionally uniting the $\mathcal{O}(n)$ remaining regions) are computed, yielding a total runtime of $\mathcal{O}(n^3(nlog(n) + klog(n))) = \mathcal{O}(n^4 log(n) + n^3 klog(n))$ by applying the overlay-algorithm to separate objects. But, surprisingly, in practice it is significantly faster than `geopandas`' in-built overlay procedure for whole subdivisions.It is worth mentioning that the overlay is not the most time-consuming part of the program. In fact, the subroutine `compute_region` takes much longer to execute. Although being theoretically called $\mathcal{O}(1)$ times for $\mathcal{O}(n^2)$ subregions, and each invocation, in turn, comprises of $\mathcal{O}(1)$ calculations, the absolute number of calls can be relatively high:

[11]https://github.com/geopandas/geopandas/issues/706, lastly visited June 12, 2022
https://github.com/geopandas/geopandas/issues/404, lastly visited June 12, 2022

Considering an 10° inner angle between the OVD bases yields

$$|\mathcal{B}| = \frac{360^\circ}{10^\circ} = 36 = \max_{C} |G(C)| \tag{3.30}$$

for the maximum number of GNNs. The theoretical number of initial subregions for n vertices thus becomes

$$\sum_{i=1}^{36} \binom{n}{i} \tag{3.31}$$

for $n \geq |\mathcal{B}|$, given at most 36 different regions can overlap, and

$$\sum_{i=1}^{n} \binom{n}{i} = 2^n - 1 \tag{3.32}$$

if $n < |\mathcal{B}|$. These numbers do not necessarily denote the actual amount of subregions, as cells with the same GNNs do not have to be adjacent. In the example 3.8 with $n = 8$, the number of initial regions (distinguished by `shapely`) is 1224; after refinement, there are 1249. Most cells lie in the outer 'spikes' of the star-shaped OVDs, which are often attributed to a single topology, explaining the low increase.
Since called for every member of a subset S with at most five elements, the upper bound of `compute_region` invocations is

$$\begin{aligned}
&\sum_{i=1}^{5} i \cdot \binom{|\mathcal{B}|}{i} = \\
&\binom{36}{1} + 2 \cdot \binom{36}{2} + 3 \cdot \binom{36}{3} + 4 \cdot \binom{36}{4} + 5 \cdot \binom{36}{5} = \\
&36 + 1260 + 21420 + 235.620 + 1.884.960 = \\
&2.143.296
\end{aligned} \tag{3.33}$$

for every subregion.
If the number of input nodes is smaller than 36, say eight, the number of invocations is still bounded by

$$\begin{aligned}
&\sum_{i=1}^{5} \binom{n}{i} = \\
&\binom{8}{1} + 2 \cdot \binom{8}{2} + 3 \cdot \binom{8}{3} + 4 \cdot \binom{8}{4} + 5 \cdot \binom{8}{5} = \\
&8 + 56 + 168 + 280 + 280 = \\
&792
\end{aligned} \tag{3.34}$$

for cells where the OVD regions of all eight vertices overlap. Therefore, the time needed to compute the refinement might take several minutes, even for small n.
An attempt to unite adjacent cells corresponding to the same GNNs or topology is implemented, but currently not used. Close, but non-adjacent regions are occasionally clustered, while adjacent regions sometimes are not. Even though two regions

border each other, there might be a small gap (or sometimes an overlap) between them, due to internal precision errors. Finding an appropriate buffer-size to check if adjacent regions overlap, touch, or intersect is difficult. If chosen too large, close, but non-adjacent regions are combined, whereas if too small, adjacent regions are not clustered at all. Furthermore, the `overlap`-, `touch`-, and `intersect`-function in `shapely` do not always return correct values, especially when dealing with more complex-shaped polygons.

3.3.6 Graphical User Interface

Lastly, the results are visualized via an embedded `matplotlib` (Hunter, 2007) plot in a `tkinter` (Lundh, 1999) window.
The user can place the data points V of the initial EMST in a predefined coordinate system. With every new vertex, the current weight of the EMST is returned by default in centimeters (cm). When the EMST is complete, the topological regions can be computed, with every topology receiving its own color. Next, a line can be drawn, along which the additional vertex x moves. The topologies of $EMST(V \cup x)$ are then plotted on the push of a 'plot next'/'plot prev' button. Errors in the algorithm are made transparent by coloring regions with false topologies red, and correct ones blue. Additionally, erroneously drawn edges are marked red, actual edges blue, and connections being present both in the false and correct tree are black. Differences in the tree weights are also indicated.
To increase the usability, the last action can always be undone. Because the bounding box is much larger than the frame of the plot, two buttons, one to display the whole box and one to focus on the EMST, are implemented. As `matplotlib` offers a device to zoom-in, another button is installed to zoom-out by a factor of 1.15. This enables the user to easily place vertices outside the initial frame. Furthermore, it is possible to pause the responsiveness of the plot, in order to zoom-in or pan axes without inadvertently adding more data points. Finally, there is a clear-all functionality to delete everything and start again, a button to open a help-window, and a button to close the application.
Figure 3.13 displays the interface showing the correct/ cosmetic areas (measured in cm^2).

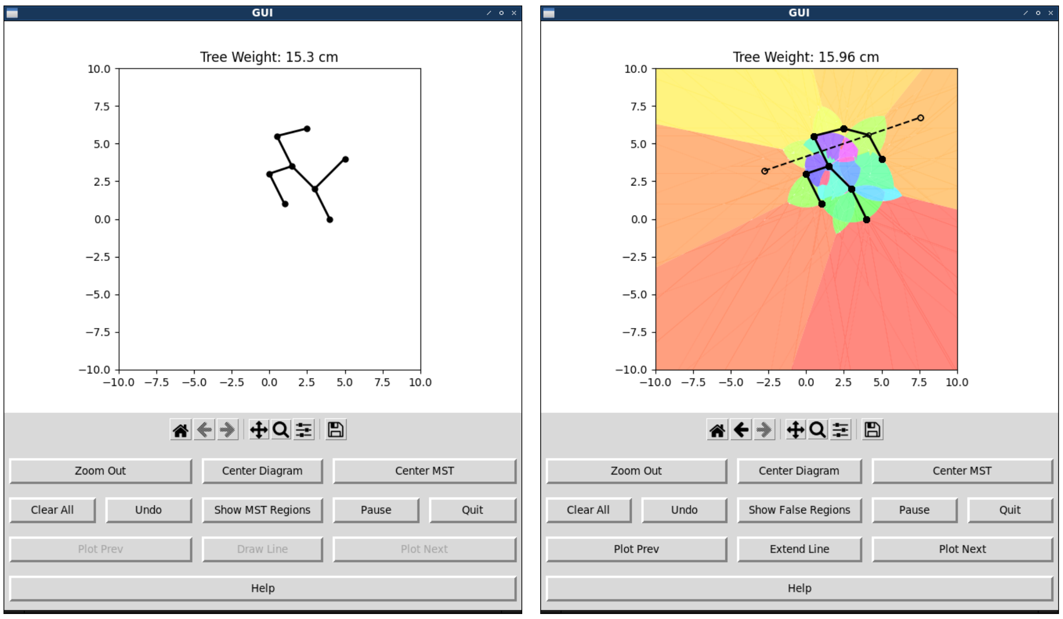

FIGURE 3.13: Left: Exemplary EMST. Right: Topological Regions plus one moving Vertex.

Figure 3.14 illustrates the handling of erroneous regions. Bolder blue lines mark euclidean Voronoi cells around the vertices.

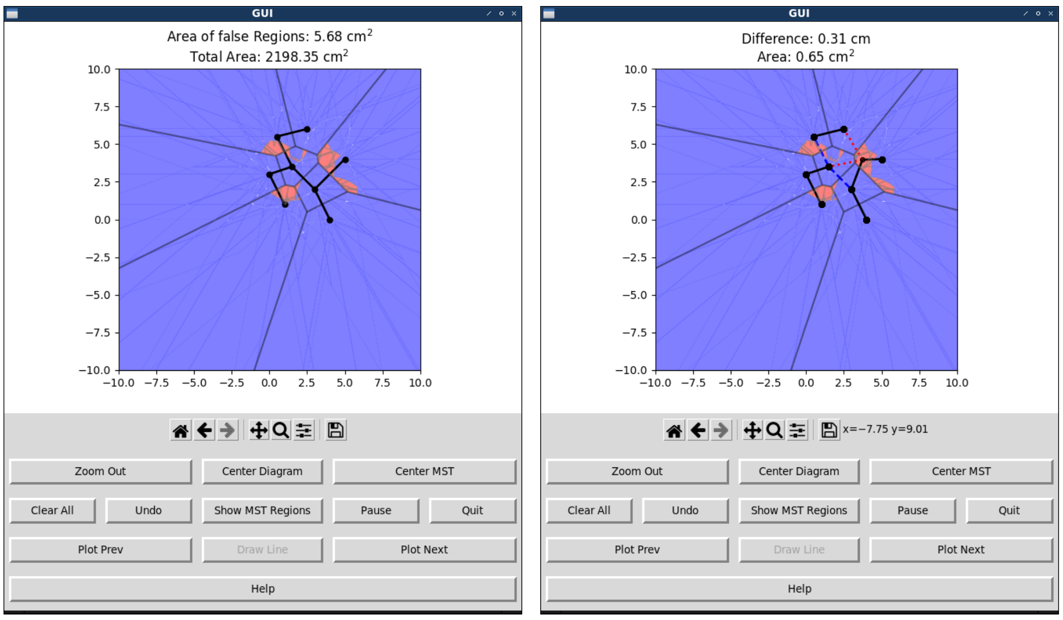

FIGURE 3.14: Left: False Region(s) with Size Indication. Right: Topological Error with Tree Weight Difference.

Chapter 4

Discussion & Future Work

The goal of this project is to describe and re-implement the algorithm by Monma and Suri (1992). In the last chapter, the outcomes are discussed, and ideas for future work are laid out.

4.1 Discussion

Two main results are about to be discussed: First, the errors found during the implementation, and second, the program itself, and how it can be useful despite prevalent defects.

4.1.1 Errors in Topologies

In the `compute_region`, a minor inaccuracy could be resolved (cf. 3.2.2). However, in certain instances, there is still a small fraction of regions (around 0.001 % of total area of the bounding box), for which the topology is falsely calculated. Being an open problem, discussed here.
An analysis of GNNs in false areas reveals that the actual EMST *never* contains an edge drawn to some vertex *not* being part of the GNNs of the area. Hence, the error does not emerge from falsely calculated OVDs, or their overlay, but from the refinement step. Figure 4.1 shows a minimal example of such one false region. Let u, v, and w denote the three vertices in left-to-right order, and R the false region. A closer look on the wrong topology (based on $EMST(V \cup x)$ with the center x of R) shows the following:

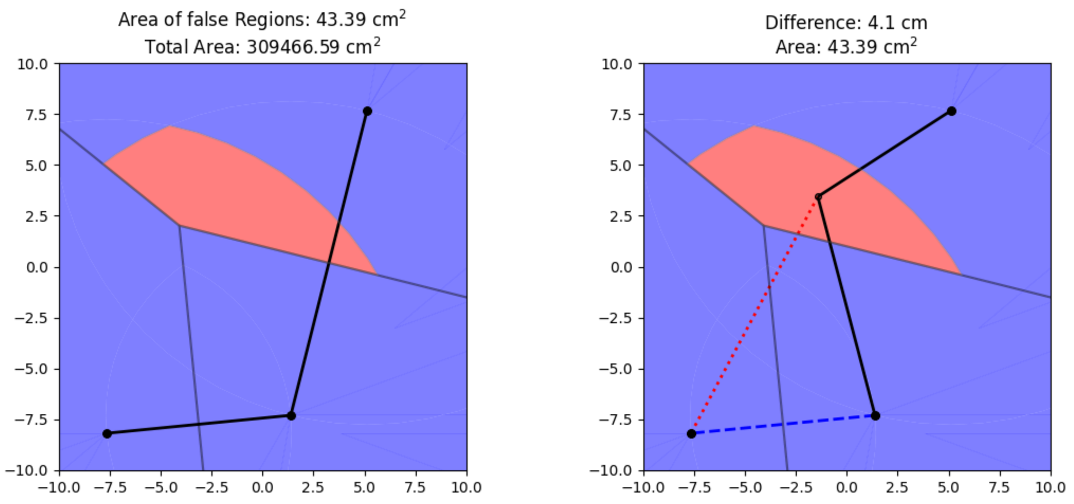

FIGURE 4.1: Minimal Example of three Vertices

In the concrete example, x is connected to all u, v, w, whereas the correct topology only includes edges to v and w. $\{u, v, w\}$ are GNNs to all coordinates in R. So, the question arises, why u is also selected. After recapitulating the (edited) refinement procedure,

```
def compute_region(C, $s_i$, S, $G(C)$):
    # Overlap cell C with euclidean Voronoi region of $s_i$ w.r.t. V
    $C' = C \cap Vor(s_i; G(C))$

    # Draw edge from x to $s_i$
    $\mathcal{G}' = EMST(V) \cup (x, s_i)$

    # $\mathcal{G}'$ contains cycles by joining $(x, s_j)$, $s_j \in G(C) \setminus s_i$
    Find longest edges in the resulting cycles with length $r_j$

    # $U(s_j, r_j)$ denotes a circle centered at $s_j$ with radius $r_j$
    $C(s_i, S) = C' \bigcap_{s_j \in S \setminus s_i} U(s_j, r_j) \bigcap_{s_j \in G(C) \setminus S} \overline{U}(s_j, r_j)$
```

the question can be rephrased in terms of what happens for subset $\{v, w\}$. Apparently, the complementary circle around u has a too wide radius. It thus seems as if solely taking the weight of the longest common edges between the vertices of a subset is not sufficient. One indication for the hypothesis is the irreproducibility of the error for MSTs with less than three nodes, because there exists just one connection. That gives rise to the following idea.
When $x \in C$ is connected to v and w, their longest common edge, (v, w), is bypassed and removed. This means, the longest common edge between v and w would no longer be (v, w). In the EMST though, the single remaining connection would be (u, v), which is shorter than (v, w). The circle complement $\overline{U}(u, w(u, v))$ then included the cell currently colored red, leading to a correctly calculated area for subset $\{v, w\}$.
Unfortunately, by pursuing this strategy, a proper subdivision could no longer be guaranteed. Consider the sub-cell $C' = C \cap Vor(u, G(C))$ of C with $G(C) = \{u, v, w\}$ for subsets $\{u, w\}$ and $\{u, v\}$. In the former, the longest common edge between u

and w would denote a different one than in the latter. Therefore, C' is possibly not fully subdivided by the circles and their complements. Certain areas in C' might be multiply covered, or others not at all. Furthermore, the order, in which the longest common edges are deleted, would matter. In sub-cell $C' = C \cap Vor(w, G(C))$, the area for subset $\{u, v, w\}$ depended on whether $U(u, r_u)$ or $U(v, r_v)$ is calculated first, asfor both circles the radius is originally the length of edge (v, w).

These considerations imply that, in the current set-up, it is not possible to readily adjust the radii of the circles and their complements without risking an improper subdivision. Despite the apparent erroneous behavior of the algorithm, the author has no solution to the problem at the time writing. How the defect arises from the code, and how it can be fixed, remains open for future investigations.

4.1.2 Summary

Notwithstanding, the project is still to a large extend useful. In case errors occur, incorrect regions cover about $\approx 0.001\%$ of the total area; for the residual cells, the returned topologies are correct. And, even if topologies are false, the resulting trees are at least planar.
These flaws apart, the other main issue is efficiency. In order to keep the project manageable (it currently comprises of over 5,000 lines of commented code), a dependence on libraries is unavoidable, and leads to an overall estimated runtime of $\mathcal{O}(n^4 log(n) + n^3 k log(n))$. The next chapter gives ideas to improve on that.
In conclusion, the project is meant to provide a starting point for future investigations into the approach by Monma and Suri. Given the limitations of a Bachelor's thesis, the focus was both to implement the algorithm suggested by Monma and Suri (1992), and make insufficiencies in the approach accessible with a user interface.

4.2 Future Work

Leaving errors aside, there are four direct improvements for the project.

4.2.1 Map Overlay

Clearly, the overlay of subdivisions is what decelerates the project most. Instead of relying on a library, the overlay of polygons can be independently implemented. One approach is presented in De Berg et al. (1999c). At the moment, the $\mathcal{O}(n)$ regions are intersected one at a time with
$\mathcal{O}(n^4 log(n) + n^3 k log(n))$. However, they could be overlaid all together in
$\mathcal{O}(|\mathcal{B}| \cdot (n + k) log(n))) = \mathcal{O}((n + k) log(n))$, by detecting intersections between two OVDs and simultaneously keeping track of the labels of overlapping subregions.
In case of the intersections in `compute_region` (4.1.1), the use of a library is justified, since the number of operations are constant in relation to the size of the input EMST.

4.2.2 List Operations

Another library the current code depends on is `blist`, on whose basis a binary search is performed. A data structure supporting the same runtime for insertions, identifications, and deletions is the binary self-balancing *red-black-tree* (Cormen et al., 2009c). By sorting given elements in ascending order from left to right, it allows all aforementioned operations in $\mathcal{O}(log(n))$ steps, while balancing can be carried out in constant time. Implementing those treeswould make the project more transparent and easier to use without additional importation.

4.2.3 Precision Model

A further issue concerns precision in the computation of orientation and intersections, sometimes leading to incorrect results. One remedy could be a more elaborate precision model which allows more decimal digits and prohibits internal rounding errors.

4.2.4 Least Commmon Ancestor Search

Lastly, finding least common ancestors in the edge tree can be accelerated. Although the in-theory runtime of $\mathcal{O}(n^2)$ is not affected by customary input sizes, there is room for improvement in the magnitude of $\mathcal{O}(n\alpha(n))$, when the suggested procedure by Harel and Tarjan (1984) were used.

Appendix A

Oriented Voronoi Diagrams

The first appendix contains the pseudo-code for the computation of the OVDs, following (Chang et al., 1990).

```
# Basis vectors for the cones
basis = [$b_1$,$b_2$]

# Sweeping Direction is the Angle Bisector of B
sweep_dir := ($b_1 + b_2$)/2

# Set of OVD Edges
$E = \emptyset$

# Initialize the Queue
$Q = \emptyset$

# Initiate L with Environment Region $R_*$
$L = [R_*]$

def main():
    # Sort Vertices in sweeping Direction
    $Q$.initialize($V$, sweep_dir)

    while $Q \neq \emptyset$ do:
        # Get coordinate of the next event in sweeping direction
        $p$ = $Q$.pop()

        if $p \in V$:
            handle_site($p$)
        else:
            handle_intersection($p$)
```

```
def handle_site($p$):
    Find $R_q \in L$ s.t. $p \in R_q$
    $i = L$.index_of($R_q$)

    # Left Boundary of $R_q$
    $B_{rq}$ = None
    if $i-1 \geq 0$:
        $B_{rq} = L[i-1]$

    # Right Boundary of $R_q$
    $B_{qs}$ = None
    if $i+1 < L$.size
        $B_{qs} = L[i+1]$

    # New Point p lies in $R_q$
    # Thus an Intersection of $B_{rq}$ and $B_{qs}$ needs to be deleted
    if $B_{rq}$ is not None and $B_{qs}$ is not None and $B_{rq}$.intersects($B_{qs}$):
        $Q$.delete($L[i-1]$.intersection($L[i+1]$))

    # Duplicate $R_q$
    $L$.insert($R_q$,$i$)

    # Insert Boundary between p and q in Direction $b_1$ starting at p
    $B_{qp} = p + b_1$
    $L$.insert($B_{qp}$,$i$)

    # Insert new region $R_p$ around p
    $L$.insert($R_p$,$i$)

    # And Boundary between p and q in Direction $b_2$ starting at p
    $B_{pq} = p + b_2$
    $L$.insert($B_{pq}$,$i$)

    # Insert any left Intersection of new Boundary $B_{rq}$ with $B_{qp}$
    if $B_{rq}$.intersects($B_{qp}$):
        $Q$.insert($B_{rq}$.intersection($B_{qp}$))

    # Insert any right Intersection of new Boundary $B_{pq}$ with $B_{qs}$
    if $B_{pq}$.intersects($B_{qs}$):
        $Q$.insert($B_{pq}$.intersection($B_{qs}$))
```

```
def handle_intersection(p):

    Find $B_{qr}, B_{rs}$ s.t. $p = B_{qr}$.intersection($B_{rs}$)

    $i = L$.index_of($B_{qr}$)
    $j = L$.index_of($B_{rs}$)

    Set $p$ as end point of $B_{qr}$ \text{and} $B_{rs}$

    # Add finished Boundaries to the Set of Edges
    $E = E \cup B_{qs}$
    $E = E \cup B_{rs}$

    # New Boundary starting at the Intersection
    Create Boundary $B_{qs}$ between $q$ and $s$

    # Delete any (now impossible) future Intersections
    # Of some $B_{pq}$ and $B_{qr}$ on the left
    $B_{pq}$ = None
    if $i - 2 \geq 0$:
        $B_{pq} = L[i-2]$

    if $B_{pq}$ is not None and $B_{pq}$.intersects($B_{qr}$):
        $Q$.delete($B_{pq}$.intersection($B_{qr}$))

    # And of $B_{rs}$ and some $B_{st}$ on the right
    $B_{st}$ = None
    if $j + 2 < L$.size
        $B_{st} = L[j+2]$

    if $B_{rs}$ is not None and $B_{rs}$.intersects($B_{st}$):
        $Q$.delete($B_{rs}$.intersection($B_{st}$))

    # Delete $B_{qr}, R_r, B_{rs}$ from $L$
    $L$.delete($i$)
    $L$.delete($i$)
    $L$.delete($i$)

    # Replace by new boundary $B_{qs}$
    $L$.insert($B_{qs}, i$)

    # Insert any left Intersection of new Boundary $B_{qs}$ with $B_{pq}$
    if $B_{pq}$.intersects($B_{qs}$):
        $Q$.insert($B_{pq}$.intersection($B_{qs}$))

    # Insert any right Intersection of new Boundary $B_{qs}$ with $B_{pq}$
    if $B_{qs}$.intersects($B_{st}$):
        $Q$.insert($B_{qs}$.intersection($B_{st}$))
```

Appendix B

Bounding-Box Test

This appendix gives the pseudo-code for the bounding-box test (Antonio, 1992) used in the detection of intersections. Returns `True`, if the bounding boxes of the both line segments overlap, otherwise `False` (see 3.12). The function is called separately with the x- and the y-coordinates of the start- and endpoints of two segments s1 and s2.

```
def box_test(s1_start, s1_end, s2_start, s2_end):

    # By default, s1_end < s1_start
    lo = s1_end
    hi = s1_start

    a = s1_end - s1_start
    b = s2_start - s2_end

    # Otherwise, exchange pointers of hi and lo
    if a >= 0:
        lo = s1_start
        hi = s1_end

    # If s2_end < s2_start
    if b > 0:
        # If the largest x/y value of s1 is smaller
        # than the lowest x/y value of s2
        # Or
        # The largest x/y value of s2 is smaller
        # than the lowest x/y value of s1,
        # s1 and s2 cannot intersect
        if hi < s2_end or s2_start < lo:
            return False

    # Else: s2_start < s2_end
    else:
        # Analogous
        if hi < s2_start or s2_end < lo:
            return False

    return True
```

Appendix C

Overlay

The following pseudo-code computes the overlay of the OVDs for frame $\mathcal{B}$.

```
def overlay():

    # Initially empty list of areas
    areas = []

    # Unite all OVD regions of one vertex
    # Let $OVD_B$ be the OVD for basis $B$
    # And $OVD_B(v)$ be the OVD region of $v \in V$ for $B$
    for $v \in V$:
        $OVD(v) = \bigcup_{B \in \mathcal{B}} OVD_B(v)$

    # Get the subsets of size 1, 2, ...$|\mathcal{B}|$ of $V$
    for $S \in \{S \subseteq V \mid |S| \leq |\mathcal{B}|\}$:

        overlap = $\bigcap_{u \in S} OVD(u)$
        subtractor = $\bigcup_{u \in V \setminus S} OVD(u)$
        overlap = overlap\ subtractor

        # Add the overlapping region and its corresponding subset
        areas.append((overlap, $S$))

    return areas
```

Bibliography

Agarwal, P. K., Eppstein, D., Guibas, L. J., and Henzinger, M. R. (1998). Parametric and kinetic minimum spanning trees. In *Proceedings 39th Annual Symposium on Foundations of Computer Science (Cat. No. 98CB36280)*, pages 596–605. IEEE.

Aigner, M. and Ziegler, G. M. (2001). Cayley's formula for the number of trees. In *Proofs from The Book*, pages 155–160. Springer.

Antonio, F. (1992). Faster line segment intersection. In *Graphics Gems III (IBM Version)*, pages 199–202. Elsevier.

Aronov, B., Bern, M., and Eppstein, D. (1994). On the number of minimal 1-steiner trees. *Discrete & Computational Geometry*, 12(1):29–34.

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

Basch, J., Guibas, L. J., and Zhang, L. (1997). Proximity problems on moving points. In *Proceedings of the thirteenth annual symposium on Computational geometry*, pages 344–351.

Bose, P., D'Angelo, A., and Durocher, S. (2020). On the restricted 1-steiner tree problem. In *International Computing and Combinatorics Conference*, pages 448–459. Springer.

Cayley, A. (1889). A theorem on trees. *Quart. J. Math.*, 23:376–378.

Chang, M. S., Huang, N.-F., and Tang, C.-Y. (1990). An optimal algorithm for constructing oriented voronoi diagrams and geographic neighborhood graphs. *Information Processing Letters*, 35(5):255–260.

Chazelle, B. (2000a). A minimum spanning tree algorithm with inverse-ackermann type complexity. *Journal of the ACM (JACM)*, 47(6):1028–1047.

Chazelle, B. (2000b). The soft heap: an approximate priority queue with optimal error rate. *Journal of the ACM (JACM)*, 47(6):1012–1027.

Cheriton, D. and Tarjan, R. E. (1976). Finding minimum spanning trees. *SIAM journal on computing*, 5(4):724–742.

Chew, L. P. and Dyrsdale III, R. L. (1985). Voronoi diagrams based on convex distance functions. In *Proceedings of the first annual symposium on Computational geometry*, pages 235–244.

Christofides, N. (1976). Worst-case analysis of a new heuristic for the travelling salesman problem. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009a). *Introduction to algorithms*, chapter 21 Data Structures for Disjoint Sets, pages 561–585. MIT press.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009b). *Introduction to algorithms*, chapter 33.1 Line-Segment Properties, pages 1015–1021. MIT press.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009c). *Introduction to algorithms*, chapter 13 Red-Black Trees, pages 308–338. MIT press.

Davis, M. and Aquino, J. (2004). Jts technical specifications. Technical report, Vivid Solutions.

De Berg, M., Van Kreveld, M., Overmars, M., and Schwarzkopf, O. (1999a). *Computational geometry*, chapter 9 Delaunay Triangulations, pages 183–210. Springer, second edition edition.

De Berg, M., Van Kreveld, M., Overmars, M., and Schwarzkopf, O. (1999b). *Computational geometry*, chapter 7 Voronoi Diagrams, pages 147–163. Springer, second edition edition.

De Berg, M., Van Kreveld, M., Overmars, M., and Schwarzkopf, O. (1999c). *Computational geometry*, chapter 2 Line Segment Intersection, pages 19–43. Springer, second edition edition.

Dey, T. K. (1998). Improved bounds for planar k-sets and related problems. *Discrete & Computational Geometry*, 19(3):373–382.

Dijkstra, E. W. et al. (1959). A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271.

Eppstein, D. (1992). Fully dynamic maintenance of euclidean minimum spanning trees.

Eppstein, D. (1994). Offline algorithms for dynamic minimum spanning tree problems. *Journal of Algorithms*, 17(2):237–250.

Eppstein, D. (1995). Dynamic euclidean minimum spanning trees and extrema of binary functions. *Discrete & Computational Geometry*, 13(1):111–122.

Eppstein, D. (1996). Spanning trees and spanners. Technical report.

Eppstein, D. (1998). Geometric lower bounds for parametric matroid optimization. *Discrete & Computational Geometry*, 20(4):463–476.

Eppstein, D. (2021). A stronger lower bound on parametric minimum spanning trees. *arXiv preprint arXiv:2105.05371*.

Frederickson, G. N. (1985). Data structures for on-line updating of minimum spanning trees, with applications. *SIAM Journal on Computing*, 14(4):781–798.

Fu, J.-J. and Lee, R. C. T. (1991). Minimum spanning trees of moving points in the plane. *IEEE Transactions on Computers*, 40(01):113–118.

Georgakopoulos, G. and Papadimitriou, C. H. (1987). The 1-steiner tree problem. *Journal of Algorithms*, 8(1):122–130.

GEOS contributors (2021). *GEOS coordinate transformation software library*. Open Source Geospatial Foundation.

Gillies, S. et al. (2007–). Shapely: manipulation and analysis of geometric objects.

Gower, J. C. and Ross, G. J. (1969). Minimum spanning trees and single linkage cluster analysis. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 18(1):54–64.

Guibas, L. J., Mitchell, J. S., and Roos, T. (1991). Voronoi diagrams of moving points in the plane. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 113–125. Springer.

Harel, D. and Tarjan, R. E. (1984). Fast algorithms for finding nearest common ancestors. *siam Journal on Computing*, 13(2):338–355.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, 585(7825):357–362.

Henzinger, M. R. and King, V. (1997). Maintaining minimum spanning trees in dynamic graphs. In *International Colloquium on Automata, Languages, and Programming*, pages 594–604. Springer.

Hunter, J. D. (2007). Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95.

Hurtado, F., Korman, M., van Kreveld, M., Löffler, M., Sacristán, V., Shioura, A., Silveira, R. I., Speckmann, B., and Tokuyama, T. (2018). Colored spanning graphs for set visualization. *Computational Geometry*, 68:262–276.

Hurtado, F., Korman, M., Van Kreveld, M., Löffler, M., Sacristán, V., Silveira, R. I., and Speckmann, B. (2013). Colored spanning graphs for set visualization. In *International Symposium on Graph Drawing*, pages 280–291. Springer.

Jarník, V. (1930). O jistém problému minimálním.(z dopisu panu o. borůvkovi).

Jordahl, K., den Bossche, J. V., Fleischmann, M., Wasserman, J., McBride, J., Gerard, J., Tratner, J., Perry, M., Badaracco, A. G., Farmer, C., Hjelle, G. A., Snow, A. D., Cochran, M., Gillies, S., Culbertson, L., Bartos, M., Eubank, N., maxalbert, Bilogur, A., Rey, S., Ren, C., Arribas-Bel, D., Wasser, L., Wolf, L. J., Journois, M., Wilson, J., Greenhall, A., Holdgraf, C., Filipe, and Leblanc, F. (2020). geopandas/geopandas: v0.10.2.

Karger, D. R., Klein, P. N., and Tarjan, R. E. (1995). A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM (JACM)*, 42(2):321–328.

Katoh, N., Tokuyama, T., and Iwano, K. (1995). On minimum and maximum spanning trees of linearly moving points. *Discrete & Computational Geometry*, 13(2):161–176.

Korte, B. and Nešetřil, J. (2001). Vojtěch jarník'work in combinatorial optimization. *Discrete Mathematics*, 235(1-3):1–17.

Korte, B. H. and Vygen, J. (2008). *Combinatorial optimization*, volume 1, chapter 21: Travelling Salesman Problem, pages 527–562. Springer.

Kruskal, J. B. (1956). On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50.

Loberman, H. and Weinberger, A. (1957). Formal procedures for connecting terminals with a minimum total wire length. *Journal of the ACM (JACM)*, 4(4):428–437.

Lundh, F. (1999). An introduction to tkinter. *URL: www. pythonware. com/library/tkinter/introduction/index. htm.*

Meulemans, W., Riche, N. H., Speckmann, B., Alper, B., and Dwyer, T. (2013). Kelpfusion: A hybrid set visualization technique. *IEEE transactions on visualization and computer graphics*, 19(11):1846–1858.

Meulemans, W., Speckmann, B., Verbeek, K., and Wulms, J. (2018). A framework for algorithm stability and its application to kinetic euclidean msts. In *Latin American Symposium on Theoretical Informatics*, pages 805–819. Springer.

Monma, C. and Suri, S. (1992). Transitions in geometric minimum spanning trees. *Discrete & Computational Geometry*, 8(3):265–293.

Nešetřil, J., Milková, E., and Nešetřilová, H. (2001). Otakar borůvka on minimum spanning tree problem translation of both the 1926 papers, comments, history. *Discrete mathematics*, 233(1-3):3–36.

O'Searcoid, M. (2006). *Metric spaces*, chapter 1.1 Metric Spaces, pages 2–8ss. Springer Science & Business Media.

Prim, R. C. (1957). Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401.

Rahmati, Z. and Zarei, A. (2012). Kinetic euclidean minimum spanning tree in the plane. *Journal of Discrete Algorithms*, 16:2–11.

Sedgewick, R. and Wayne, K. (2011a). *Algorithms*, chapter 2.2 Mergesort, pages 270 – 287. Addison-wesley professional, fourth edition edition.

Sedgewick, R. and Wayne, K. (2011b). *Algorithms*, chapter 2.4 Priority Queues, pages 308 – 335. Addison-wesley professional, fourth edition edition.

Shamos, M. I. and Hoey, D. (1975). Closest-point problems. In *16th Annual Symposium on Foundations of Computer Science (sfcs 1975)*, pages 151–162. IEEE.

Sleator, D. D. and Tarjan, R. E. (1983). A data structure for dynamic trees. *Journal of computer and system sciences*, 26(3):362–391.

Tarjan, R. E. (1979). Applications of path compression on balanced trees. *Journal of the ACM (JACM)*, 26(4):690–715.

Tarjan, R. E. (1983a). *Data structures and network algorithms*, chapter 6 Minimum Spanning Trees, pages 71–83. SIAM.

Tarjan, R. E. (1983b). *Data structures and network algorithms*, chapter 2 Disjoint Sets, pages 23–31. SIAM.

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., Carey, C. J., Polat, İ., Feng, Y., Moore, E. W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E. A., Harris, C. R., Archibald, A. M., Ribeiro, A. H., Pedregosa, F., van Mulbregt, P., and SciPy 1.0 Contributors (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272.

Yao, A. C.-C. (1982). On constructing minimum spanning trees in k-dimensional spaces and related problems. *SIAM Journal on Computing*, 11(4):721–736.

Yu, M., Hillebrand, A., Tewarie, P., Meier, J., van Dijk, B., Van Mieghem, P., and Stam, C. J. (2015). Hierarchical clustering in minimum spanning trees. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 25(2):023107.