Implementation of an Orthoradial Graph Drawing Algorithm

Bachelor Thesis of

Katharina Götz

At the Department of Informatics and Mathematics
Chair of Theoretical Computer Science

Reviewers: Prof. Dr. I. Rutter
Prof. Dr. C. Bachmaier

Time Period: 5st July 2018 – 2nd October 2018

Statement of Authorship

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

Passau, October 2, 2018

Abstract

The main objectives of this work are to give an overview of the theory for computing an orthoradial graph drawing and to provide an implementation of an orthoradial graph drawing algorithm. The process, from a connected, 4-planar graph to an orthogonal graph drawing, is described by the Topology-Shape-Metrics (TSM) framework. In three steps, the framework computes a topology, a shape and then the metrics for a drawing. A graph embedding describes the topology of a graph by storing the order of vertices relative to each other. The shape computed from the embedding describes the necessary angles in an orthogonal drawing with rotations and is stored in a graph representation. The metrics for the drawing, consisting of the lengths of the edges and the coordinates of vertices, are computed in the last step. The computation and an overview of each step are explained and summarized. Based on recent papers, the TSM framework can be extended for orthoradial graph drawings. The similarities and differences of the framework's steps for orthogonal and for orthoradial drawings are explained. The most important difference to the orthogonal case is the potential existence of monotone cycles in orthoradial representations. These cycles prevent the computation of an orthoradial drawing. The theory about the monotone cycles and the search process for them are contained. The TSM framework for orthoradial drawings is implemented as a functional prototype. The focus lies on the search of monotone cycles. Also, the processes of the implementation are shown on an example graph.

Deutsche Zusammenfassung

Die Hauptziele dieser Arbeit sind einen Überblick über die Theorie zur Errechnung einer orthoradialen Graphzeichnung zu geben und eine Implementation eines Algorithmus zum Zeichnen von orthoradialen Graphen bereit zu stellen. Der Prozess von einem 4-planaren, zusammenhängenden Graphen zu einer orthogonalen Zeichnung ist beschrieben durch das "Topology-Shape-Metrics (TSM)" Framework. In drei Schritten werden die Topologie (engl. topology), die Form (engl. shape) und die Metriken (engl. metrics) für die Zeichnung berechnet. Eine Einbettung des Graphen beschreibt die Topologie für den Graphen, indem die Ordnung der Knoten relativ zueinander gespeichert wird. Die Form wird basierend auf der Einbettung berechnet und beschreibt die benötigten Winkel für eine orthogonale Zeichnung als Rotationen. Diese wird als Graph Repräsentation gespeichert. Die Metriken für die Zeichnung, die aus den Längen der Kanten und den Koordinaten der Knoten bestehen, werden im letzten Schritt berechnet. Die Berechnungen und eine Übersicht der einzelnen Schritte zusammengefasst und erklärt. Basierend auf kürzlich erschienenen Arbeiten kann das TSM Framework auf das orthoradiale Graphzeichnen übertragen werden. Die Ähnlichkeiten und Unterschiede der Schritte des TSM Frameworks für orthogonale und orthoradiale Zeichnungen werden erklärt. Der wichtigste Unterschied zum orthogonalen Fall ist die mögliche Existenz von monotonen Kreisen in orthoradialen Repräsentationen. Diese Kreise verhindern die Berechnung einer orthoradialen Zeichnung. Die Theorie über monotone Kreise und der Prozess, um diese zu finden, werden beschrieben. Das TSM Framework für orthoradiale Zeichnung ist implementiert als ein funktionierender Prototyp. Der Schwerpunkt liegt hier auf der Suche nach monotonen Kreisen. Die Arbeitsabläufe der Implementierung werden an einem Beispielgraphen illustriert.

Contents

1. Introduction

The visualization of data as a graphic serves the purpose of quickly relaying information. The field of graph theory and graph drawing in computer science and mathematics researches the precise and clear presentation of such data. For example, a bus station map of a city is often abstracted to the outline of the city and the bus routes are simplified. Consider all bus stations and routes as the information of a map. This information can be saved as a graph, whose vertices are the bus stations and the edges are the bus routes connecting the vertices. Also, a city map containing streets and addresses can be saved as a graph. Figure 1.1 shows the city center of Passau as an example. A graph drawing algorithm computes a clear and precise drawing of the graph.

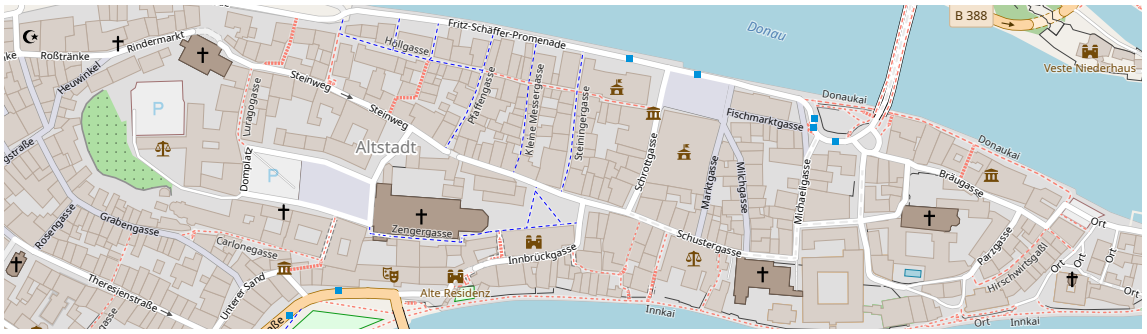


Figure 1.1: Map of the city center of Passau.[1]

Figure 1.2 shows a simple example graph in textual form on its left. A graph is structured with vertices, also known as nodes, and edges and does not contain any graphical information. A drawing is based on a graph and presents the graphs edges and vertices. Depending on the chosen arrangement of these, a graph can have many drawings. As an example, Figure 1.2 shows two possible drawings of the graph on its right.

Orthogonal drawings, whose lines are horizontal and vertical lines, are widely used in graph drawing. The graph drawing algorithm based on Tamassia's Topology-Shape-Metrics (TSM) framework is often referenced in literature about orthogonal drawings. Because the framework is a foundation of graph drawing, further research on optimization problems, like bend minimization, is available. In contrast to orthogonal drawings, graph

[1]Exported from: https://www.openstreetmap.org/export#map=16/48.5731/13.4647 (ODbL License)

Vertices: {0; 1; 2; 3; 4}
Edges: {(0,1);(1,2);(2,3);(3,4);(4,1)}

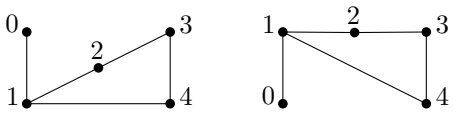


Figure 1.2: Example for a graph and a graph drawing.

drawing algorithms for other grids than the orthogonal one are less often found. However, orthogonal drawings have the disadvantage of being unable to represent concentric circles. An orthoradial drawing on the other hand has an underlying concentric grid allowing the representation of cycles. Figure 1.3 shows an orthogonal and an orthoradial drawing. Recent papers about orthoradial graph drawing contain the theory on how to obtain an orthoradial drawing. These papers prove the first steps to a drawing to be similar to the orthogonal steps. Barth et al. [BNRW17a] transfer the TSM framework onto orthoradial graph drawing and prove the framework to be applicable. Niedermann et al. [NRW13] build on their work and outline the problems for orthoradial graph drawing. The existence of monotone cycles in an orthoradial representation prevents the computation of a drawing and Niedermann et al. [NRW13] present a theoretical polynomial time algorithm for finding monotone cycles. This work gives an overview over the application of the TSM framework on orthogonal graph drawing. We use this as a foundation and provide an overview of the theory of applying the TSM framework on orthoradial graph drawing. As a second main topic, the theoretical algorithms are implemented to prove their applicability. This implementation is the first orthoradial graph drawing algorithm based on the TSM framework. Its focus lies on the algorithm for finding monotone cycles, which is also the first implementation of the algorithm.

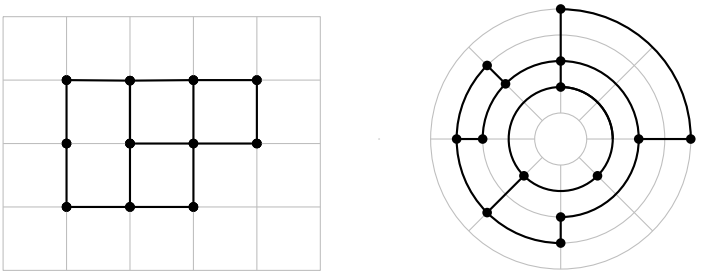


Figure 1.3: Orthogonal drawing (left) and orthoradial drawing (right) with their grids indicated by gray lines.

For orthoradial graph representations, we introduce a new attribute, called orientation, that describes the edges in addition to the rotations. Also, we amend a lemma that describes the dependence between vertex rotation and vertex degree and provide a proof. The missing details of the original papers for the face rotation of orthoradial graph drawings are provided. We introduce the process of splitting polygons in an orthoradial grid in order to get the internal angle sum of these polygons. The angle sum is the basis of the proof for the requirements on face rotation in an orthoradial representation. This work also includes the working method of the network flow algorithms for computing the coordinates and for computing the rotations with bend minimization. Also, exact details on finding the cycle direction and monotone cycles are summarized. The effects of monotone cycles on the augmentation of the representation and on its candidate edges are evaluated. The implementation of the orthoradial graph drawing is functioning as desired. Embeddings, orthoradial representations and also graph drawings can be computed. The execution of the program can start at each of these steps as another feature. Therefore, an input to our algorithm can either be a graph, an embedding or already a graph representation. The program also includes finding the faces of a representation for the augmentation and

network flow algorithms. After computing an orthoradial representation the graph is searched for monotone cycles. If any such cycle is found, it is repaired by inserting a spiral to correct the labels of the edges. The augmentation is also completely implemented. This includes the checks after inserting edges to the candidate edges and handling of special cases, for example, when a monotone cycle is inserted by the augmentation.

This work concentrates on orthogonal and orthoradial graph drawings, as these visualize information clearly and understandably. The concepts of paths, cycles and a vertex degree are assumed to be known. In this work, edges are considered bend-free, which means they are only portrayed with straight lines in the drawings and they do not change their direction. Chapter 2 provides an overview of the foundations. The concepts of planarity, graph embedding and facets are explained. Chapter 3 lists important related work to orthogonal and orthoradial graphs and their drawings. The main theory chapters of this work are Chapter 4 and 5. The first one describes the process from a graph to an orthogonal graph drawing. The second one explains the application of this process to the orthoradial case and lists the problems and the procedures to handle them. Chapter 6 contains information about the usage of the implementation and shows extracts of the procedures behind the implementation.

2. Preliminaries

This chapter explains the foundations about information encoded in graph drawings. A graph drawing is planar if it can be drawn with no crossings at all, which prevents overlapping edges [Wei01]. A graph is planar, if a planar graph drawing for the graph exists. In the following of this work all graphs are considered to be planar and to be 4-graphs, which defines a graph that includes only vertices with a degree of at most 4. Figure 2.1 shows a non-planar as well as a planar graph drawing. Also, the graphs are considered connected and therefore, no vertex has a vertex degree of 0.

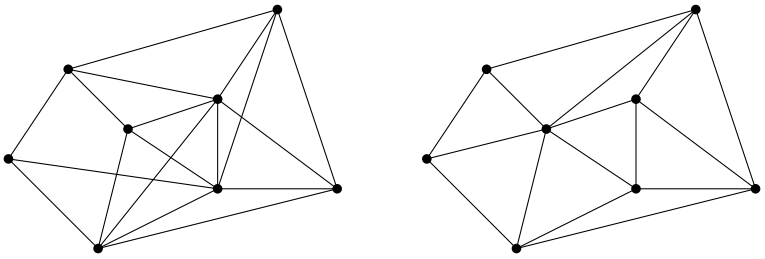


Figure 2.1: Non-planar (left) and planar (right) graph drawing of two different graphs.

When a planar graph drawing is available for a graph, we can extract information about faces. Given a graph drawing Γ for an arbitrary graph G, then, a *face* in G is defined as a connected component of $\mathbb{R}^2 \backslash \Gamma$, which excludes all edges and vertices and leaves the remaining areas as pairwise disjoint faces. Figure 2.2 shows an example graph drawing with its faces marked in different colors. These three faces are separated by the edges of the graph. Also, note that the faces f_1 and f_2 are completely enclosed by edges of the graph but the face f_0 fills the area around the drawing and is therefore unbounded. This face is called the *outer face* and all other faces are *regular faces*. A regular face can be uniquely described by the sequence of edges that are enclosing the face. The outer face is defined by the sequence of edges that are separating the drawing from the remaining space. The sequence of edges form the *facial cycle* around a face, which is directed so that the face lies on the right side of the cycle. The direction for regular faces is therefore in clockwise direction around the face and for the outer face in counter-clockwise direction. We define this sequence of edges as the *border* of a face. Note that not all compositions of edges define a face. Also, every face has exactly one border. Representing each face by its border is more space efficient than saving all faces in a graphic. The faces can then

be identified from the saved sequences afterwards. Figure 2.2 shows an example graph drawing with its faces and their borders marked.

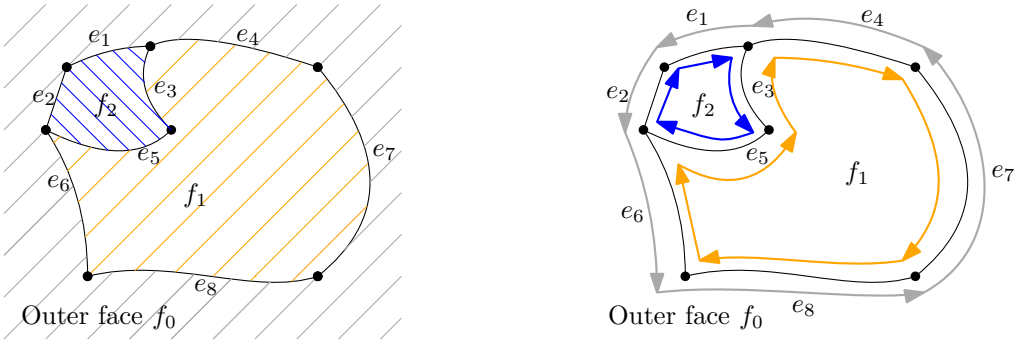


Figure 2.2: Graph drawing with faces f_0, f_1 and f_2. The corresponding sequence of edges are for f_0 $[e_1, e_2, e_6, e_8, e_7, e_4]$, for f_1 $[e_3, e_4, e_7, e_8, e_6, e_5]$ and for f_2 $[e_1, e_3, e_5, e_2]$.

The order of neighbor vertices around a vertex is saved in a so called adjacency list or adjacency entry. In the following the adjacency lists of an example graph drawing, shown in Figure 2.3, are extracted and explained. Each vertex v of a graph is assigned an adjacency list containing the order of neighboring vertices of v in clockwise direction. Therefore, it also describes the order of the edges connected to v. As this sequence is cyclic, the sequence adj(v)=$[u, w, t]$ for the vertices v, u, w, t is equivalent to the sequence adj(v)=$[t, u, w]$. When vertex v is drawn in a graph drawing, the neighboring vertices are then arranged around v in the listed order. Figure 2.3 shows an example drawing of a graph G. The adjacency lists for the vertices $v_1, ..., v_{11}$ of G can be easily extracted. For example, take vertex v_5 and all edges connected to it, which are v_5v_1, v_5v_4, v_5v_6 and v_5v_9. Because the only requirement for the sequence is the clockwise order, the adjacency list for v_5 is adj(v_5)=$[v_1, v_6, v_4]$ or any circular shifted version of it. The remaining adjacency lists for the other vertices are included in Figure 2.3.

Vertex	Adjacency entry
$v1$	$[v_2, v_5]$
$v2$	$[v_3, v_6, v_1]$
$v3$	$[v_7, v_2]$
$v4$	$[v_5, v_8]$
$v5$	$[v_1, v_6, v_4]$
$v6$	$[v_2, v_7, v_{10}, v_8, v_5]$
$v7$	$[v_3, v_{11}, v_6]$
$v8$	$[v_4, v_6, v_9]$
$v9$	$[v_{10}, v_8]$
$v10$	$[v_6, v_{11}, v_9]$
$v11$	$[v_{10}, v_7]$

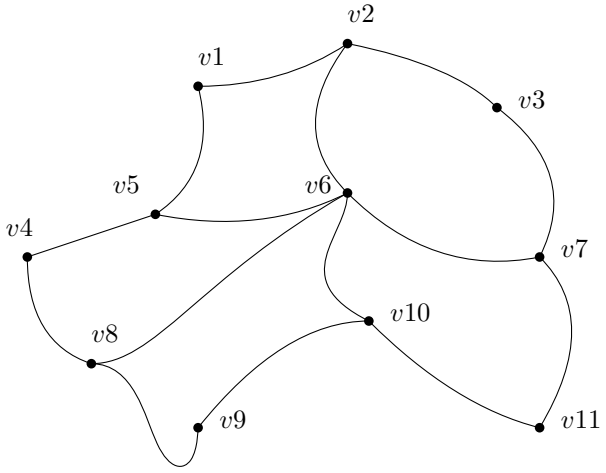


Figure 2.3: Graph drawing with embedding. Adjacency lists in clockwise direction.

Instead of saving both the sequence of edges for a face and also the adjacency list, only the latter one is needed. From the order of edges around a vertex the border of a face can be reconstructed. The adjacency lists are saved in a graph embedding. Therefore, faces of a graph are defined, when an embedding is fixed. Only the outer face needs to be specially specified with at least one edge that lies on its border, called *reference edge*.

In order to reconstruct a regular face we need to follow the edges in clockwise direction around a face. To find the edges of the outer face we need to follow the declared reference edge in counter-clockwise direction around the graph, so that the outer face is on the right side of these edges. The adjacency lists give the successive edge for traversing to the next edge of a face. The previous index of the adjacency list of a vertex v gives the next vertex u around a face and therefore, the next edge vu. This ensures the respective direction around a face of the found sequence, which corresponds to the innermost cycle bounding the face. Because the next edge has to be the next in counter-clockwise direction around a vertex v and the adjacency list is the order of these edges around v in clockwise direction, the previous index is desired. The search for the next edge is iterated until the first edge is found again. The Figure 2.4 includes a graph with a face f and its border $[vu, uw, wt, tv]$, which is marked in blue.

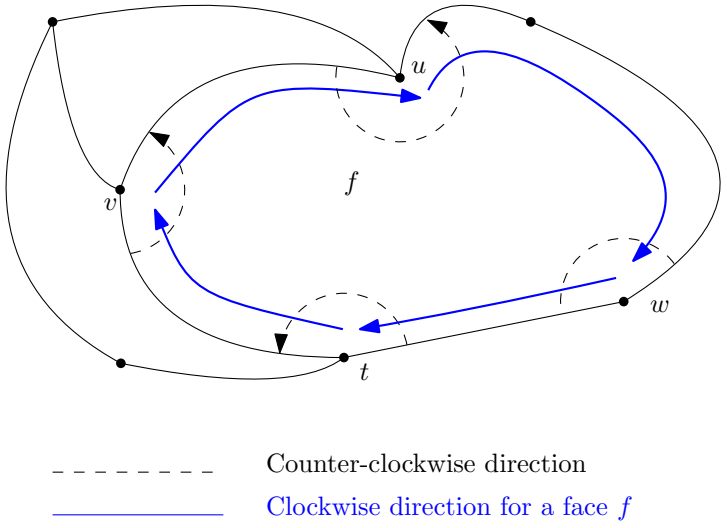


Figure 2.4: Finding the next edge around the face f. Surrounding edges $[v, u, w, t]$

3. Related work

The following publications about graph theory form a foundation for working with orthoradial graph drawings. The collection from Tamassia in 2013 [Tam13] contains important subjects for our work. The book of Battista et al. [BETT98] is another big collection of subjects of graph theory. Many of the other reference publications are also contained in one of these collections. Other subjects and problems in graph drawing are also introduced and discussed in the included papers, however, only a small part is outlined in this work.

Drawing a graph as an orthogonal drawing has two major approaches. The first one tries to arrange the angles for each vertex v one by one with the aim of assigning preferable large angles between the edges around v. The second approach specifies a grid in which the graph has to be embedded. The set of possible angles is chosen beforehand and the graph can only contain these. Eiglsperger et al. [EFK01] explain the relationship between angles in a graph and their influence on a graph's readability. They choose to work with orthogonal graph drawings, as these are considered well-readable and clear. Figure 1.3 in Chapter 1 shows an example orthogonal drawing. The angles in the grid, which are multiples of $90°$, make the edges easily distinguishable from each other. As a result of the set of possible angles, graphs admitting an orthogonal drawing are 4-graphs that limit the vertices to have a maximum degree of 4.

Orthogonal graph drawings all have the characteristic of planarity. Planar graphs and planar drawings are defined in Chapter 2. A planar drawing can be drawn with no crossings, which is desired, as edge crossings or intersections reduce the readability. Weiskircher [Wei01] deals with the definition of planarity, planarity testing and drawing algorithms for planar graphs. Also, Patrignani [Pat13] summarizes algorithms for efficiently testing planarity and computing planar embeddings. The paper classifies linear-time planarity algorithms into two categories, cycle-based algorithms and vertex-addition algorithms, and explains them. Besides testing the planarity, the subject of crossing minimization is also a main aspect in graph theory. The paper of Buchheim [BCG$^+$13] contains more details on the crossing minimization problem, which is an NP-hard optimization problem. The paper discusses different approaches that try to solve or approximate this problem.

The basic concepts of graph drawing are outlined in Tamassia's paper of automatic graph drawing [TDBB88]. Firstly, it describes the difficulties of placing a set of textural data onto a fixed grid to get a logical and well-readable graphic without extra information. Secondly, the paper summarizes graph drawing algorithms for planar graphs and other graph categories. A complete introduction to graphs and more current versions of graph drawing algorithms are contained in the Graph Drawing Handbook [Tam13].

An important component of a graph drawing algorithm is the computation of the graph representation, which fixes the rotations between consecutive edges around a vertex and the bends on the edges of the graph. The definition of the rotations can be found in the work of Bläsius et al. [BKRW11]. Whether a planar graph can be drawn without bends is a NP-complete problem. Also, bend minimization is an optimization problem that is NP-hard if no embedding is chosen for a graph. If an embedding is fixed, an algorithm for bend minimization, introduced in the work of Tamassia [Tam87] and the book of Battista et al. [BETT98], is running in polynomial time. The need for bends and positioning of these are computed with a minimum cost flow algorithm that distributes the rotations as flow units. More information about network flow algorithms is included in the work of Duncan et al. [DG13]. Details about the running time of minimal cost flow algorithms can be found in the work of Garg et al. [GT97] and in the work of Cornelsen et al. [CK11]. A more detailed version of the application of the network flow is contained in the work of Battista et al. [BETT98][Chapter 5]. Tamassia's paper of 1987 [Tam87] also contains the computation of the angles by a network flow algorithm. Additionally, it contains the theory for proving an orthogonal graph representation to be drawable by augmentation, also known as rectangulation. For a drawing of an orthogonal representation to exist once the embedding and representation are fixed, two rotation requirements have to be fulfilled. These allow the angles to be converted successfully.

A very similar category of drawings to orthogonal ones are orthoradial drawings, as these are also based on planar 4-graphs. The recent theorems in the paper of Barth et al. [BNRW17a] for orthoradial graphs are transferring the TSM framework of Tamassia to orthoradial graph drawing. The TSM framework is explained in Tamassia's paper in 1987 [Tam87]. However, orthoradial representations are not drawable despite the requirements for the rotations being fulfilled. Hasheminezhad et al. [HHT09] characterize drawable cycles. If a representation contains a non-drawable cycle, it prevents the computation of a drawing. Based on this, Barth et al. [BNRW17a] define these cycles as monotone cycles and describe the labeling of a cycle to classify a cycle to be monotone. The manuscript of Niedermann et al. [NRW13] introduces an algorithm for finding monotone cycles in a graph, which is included as an important feature in the implementation. Both papers of Barth et al. [BNRW17a] and Niedermann et al. [NRW13] also contain details on the augmentation of orthoradial representations. Note that there also exists a revised conference version [BNRW17b] of the paper of Barth et al. [BNRW17a] cited in this work. However, as conference versions are often shortened, all citations of the publication of Barth et al. point to the extensive version [BNRW17a].

The Open Graph Drawing Framework (OGDF) [CGJ+13, Chapter 17] is an important feature for the implementation and is included in the collection of 2013 [Tam13]. The Open Graph Drawing Framework (OGDF)[1] is described as "a self-contained C++ class library for the automatic layout of diagrams." on the official website. This open source project is a combined work of many universities and free to use for everyone. Reading and writing GML files, classes to represent graphs and many algorithms are only a small part of its features. The paper of Chimani et al. [CGJ+13] is a summary of its functionalities and structure and explains the featured graph algorithms. Also noteworthy is the GML [2] file format used to save a graph in textual form. Graphs saved in GML format can be easily transformed into a graphic because the format can also contain graphical information like coordinates of vertices.

[1] http://www.ogdf.net

[2] https://www.fim.uni-passau.de/fileadmin/files/lehrstuhl/brandenburg/projekte/gml/gml-technical-report.pdf

4. Orthogonal Graph Drawing

To understand orthoradial graph representations we first look at orthogonal ones. Tamassia [Tam87] summarizes orthogonal drawings and representations and his work is the basis for this section. The pipeline of this algorithm for getting an orthogonal drawing contains three steps. The first step takes a graph, described as textual information, as input and constructs a graph embedding for the graph. In this step the order of edges, incident to a vertex v around itself, is fixed. The second step takes a graph embedding and constructs a graph representation. This step fixes the angles between the edges around v. These angles are represented by rotations. The graph representation also contains information about the needed bends on the edges for the next step. The last step takes a graph representation and fixes the coordinates of the vertices and the lengths of the edges based on the rotations of the representation. Figure 4.1 visualizes this pipeline. When calculating the graph drawing along the pipeline the steps are dependent on each other and build up on one another. It is only possible to define a graph representation if the graph embedding was defined beforehand, as it is an extension. When reversing the pipeline the previous step can be easily deduced because of this dependency.

Figure 4.1: Steps for an algorithm to draw an orthogonal or orthoradial graph.

The pipeline describes the Topology-Shape-Metrics framework of Tamassia [Tam87]. The first step receives only the graph's vertices and edges and, to get to the second, we need to define a topology, which is defined by the graph embedding. It fixes the circular order of edges around a vertex. However, we are not getting any geometric information like the coordinate of a vertex, but we get first constraints for the graph drawing. For example, if the order around a vertex v is adj(v)=$[e_1, e_2, e_3, e_4]$ in clockwise direction and the resulting graph drawing places e_1 relatively downwards from v, then we can conclude that e_2 is to the left, e_3 is upwards and e_4 is to v's right. In this case e_3, for example, cannot be placed to the right of v as it would break the listed order. Note that the outer face is also fixed when the embedding is chosen. If a topology is fixed, the drawing's shape can be described with a graph representation, which specifies the angles at the vertices between the neighboring edges. At the second step the embedding gives the orders of the edges around each vertex and the angles between consecutive edges in the circular order around a vertex are fixed. Additionally, information about the bends in the graph drawing and

their direction is contained to satisfy orthogonality. Figure 4.2 shows two graph drawings that are based on the same graph but different embeddings that differ at a vertex v.

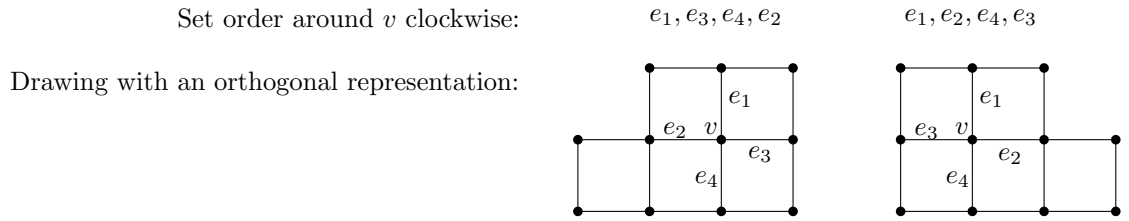


Figure 4.2: Graph drawings with different embeddings at vertex v.

For the second step, the graph representation needs to be fixed. In the Graph Drawing book [BETT98] the representation is described as an orthogonal shape, which considers the angles but neglects the edge lengths. The type of representation can be distinguished by the set of potential angles. A precondition for an orthogonal or orthoradial representation is a 4-graph and the corresponding angle set for orthogonal or orthoradial is {90°, 180°, 270°, 360°}. When the embedding and hence, the order of the edges around the vertices is fixed, the angles between consecutive edges in the given order at a vertex can be determined. Figure 4.3 shows the drawings of two different representations that are based on the same embedding. At vertex v, the rotations and therefore, the angles are chosen differently.

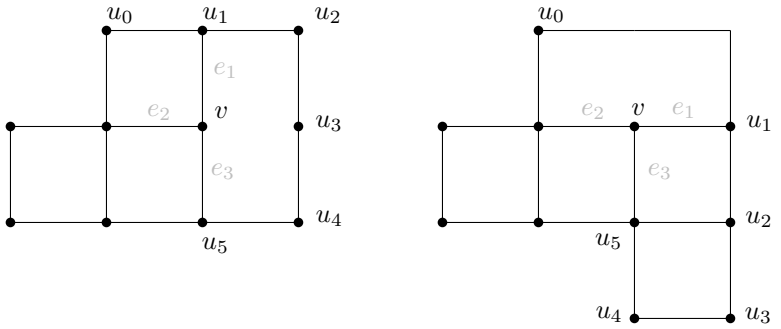


Figure 4.3: Graph Drawings with rotation at vertex v. On the left angle$(e_3, e_1) = 180°$ and angle$(e_1, e_2) = 90°$. On the right angle$(e_3, e_1) = 90°$ and angle$(e_1, e_2) = 180°$.

The third step defines the metrics which are composed of the length of the edges and the coordinates of the vertices. Afterwards, the necessary information for a drawing are computed and it can be drawn. In order to understand the importance of this pipeline and the connection between the steps, we reverse the pipeline and illustrate the steps on an example graph. The step backwards to a graph representation from a graph drawing is the first in the reversed pipeline and is explained in the next Section 4.1. The next step, the extraction of a graph embedding from a graph representation, is illustrated in the example in Chapter 2. The order around each vertex is saved in adjacency lists in the embedding. In the example in Chapter 2 the embedding is extracted from a drawing. When only a graph representation is available, instead of a drawing, the adjacency lists can be easily determined by considering the provided angles given by the representation. The step from a graph embedding to a graph is executed by ignoring the adjacency lists, as only the vertices and edges are needed. Therefore, only the step from a graph drawing to a graph representation is explained in Section 4.1.

Section 4.2 presents another approach for computing a drawing for a graph representation. Tamassia [Tam87] proves that a drawing exists for a graph representation with only rectangles as faces, which have a face rotation of 4. An arbitrary graph representation can be checked for rectangles and its faces can be converted to rectangles by augmenting

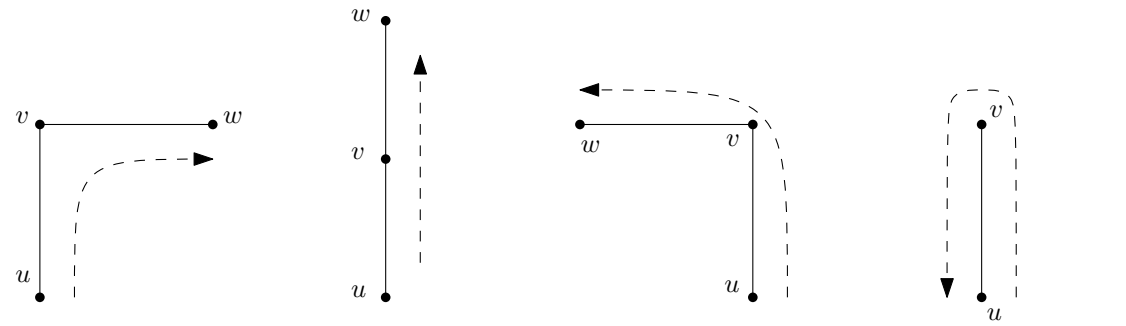
with fictitious edges and vertices. The Section 4.3 explains the idea for computing a graph drawing from a graph representation and the used network flow algorithm.

4.1 From a Graph Drawing to a Graph Representation

This section contains examples for determining the graph representation, the third step of the pipeline, from a graph drawing, which is the end of the pipeline. As explained before, a graph representation includes the information about the angles around the vertices of a graph drawing. An orthogonal representation describes the angles of a drawing of a graph embedded into an orthogonal grid, which only has horizontal and vertical segments. The angles of an orthogonal drawing are stored in rotations, which are explained in Subsection 4.1.1. The extraction of the rotations from a drawing is shown in Subsection 4.1.2. Also, representations can be drawable, which is defined in Section 4.1.3. If an orthogonal representation is drawable, its rotations fulfill certain requirements, which are also included in Section 4.1.3.

4.1.1 From Angles to Rotations

In this section the definition of rotation for a path and a face is reused from the work of Bläsius et al. [BKRW11] and more detailed information can be found in their work. The rotation for a vertex and a cycle can derived from the rotation of a path with length 2, which describes the angle formed by two neighboring edges in counter-clockwise direction at a vertex. For consecutive edges in circular order around a vertex a rotation value is stored in the graph representation so that the angle between these edges in a graph drawing is fixed. When considering a drawing to be existent, the geometric requirements for the drawing result in each vertex having 360° distributed around itself. Therefore, the sum of the angles between consecutive edges in circular order around each vertex is exactly 360° and a vertex has, per precondition of 4-graphs, up to four neighbors. Simplified, we have four equal units of 90° to distribute at each vertex. If a vertex has less than four neighbors one or more angles will get two or more units of 90°, so that 360° are distributed around the vertex. For the definition of rotations, the angle between two edges uv, vw that share a common vertex v is defined as angle(uv, vw), which gives the geometric angle in degree enclosed by the edges in a drawing in counter-clockwise direction.



angle$(uv, vw) = 90°$ rot$(uv, vw) = 1$

angle$(uv, vw) = 180°$ rot$(uv, vw) = 0$

angle$(uv, vw) = 270°$ rot$(uv, vw) = -1$

angle$(uv, vw) = 360°$ rot$(uv, vu) = -2$

Figure 4.4: Rotations and their drawings.

The rotation between two edges uv, vw, defined as rot(uv, vw), describes the angle that is enclosed by the edges starting from uv in counter-clockwise direction to vw. Because of the angle set of orthogonal drawings the edges can have an angle of 90°, 180° or 270°. If the rotation between an edge and itself is measured, we have an angle of 360°. The

possible angles {90°, 180°, 270°, 360°} of an orthogonal drawing are abstracted to the rotations $\{1, 0, -1, -2\}$. Figure 4.4 shows the corresponding rotations and angles. A 90° angle starting from a vertex u over v to w shapes a right bend and is represented with a rotation of 1. Therefore, the number of rotations of value 1 in a graph represents the number of 90° angles. A 270° angle starting from a vertex u over v to w forms a left bend and is represented with a rotation of -1. Therefore, also the number of left bends can be easily found when counting rotations of -1. The rotation 0 equals an angle of 180° and represents a straight line form u to w, which does not contribute to the number of bends. A 360° angle shapes a turn-around, which can also be interpreted as two left turns around v. This angle is represented with rotation -2, which represents two rotations of -1. Given a graph G containing the vertices v, u and w and the undirected edges uv, vw, we define the rotation between two edges uv and vw as

$$rot(uv, vw) = \begin{cases} 1 & \text{if angle}(uv, vw) = 90° \\ 0 & \text{if angle}(uv, vw) = 180° \\ -1 & \text{if angle}(uv, vw) = 270° \\ -2 & \text{if angle}(uv, vw) = 360° \end{cases}. \tag{4.1}$$

The rotation can also be calculated with the formula

$$\text{rot}(uv, vw) = \frac{180° - \text{angle}(uv, vw)}{90°}. \tag{4.2}$$

Niedermann et al. [NRW13] represent the edges used in $\text{rot}(e_1, e_2)$ through their vertices, which are the source of e_1, the common vertex of the edges and the target of e_2. They define the rotation as $\text{rot}(u, v, w)$ for the edges uv and vw, which does not alter the rotation definition. The rotation $\text{rot}(uv, vw)$ for two edges uv, vw defines the rotation of a path of length 2. The definition of rotation can be also extended to arbitrary paths. To get the rotation of a longer path we accumulate the rotations of consecutive edge pairs. The *path rotation* of a path P with edges $e_1, ..., e_k$ and length k is defined as

$$\text{rot}(P) = \sum_{i=1}^{k-1} \text{rot}(e_i, e_{i+1}). \tag{4.3}$$

Figure 4.5 shows on its left a drawing of an example path P. The graph drawing is based on a graph G with edges $e_1, ..., e_8$ and the path $P = [e_1, ..., e_8]$ has length 8. The rotation of P is $\text{rot}(P) = -2$, based on the formula for a path rotation. Note that the rotation 0 is negligible as 180° angles represent no bend. Also, the path P contains two more left bends than right ones, which is also represented by the path rotation of -2. Therefore, the rotation of a path P can also be defined as the number of right bends minus the number of left bends.

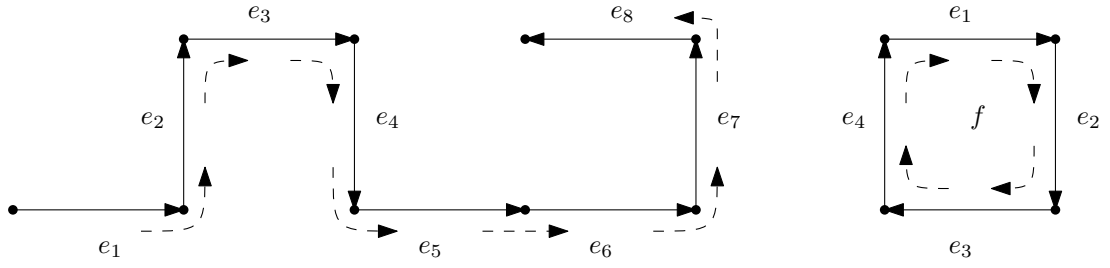


Figure 4.5: Path P with $\text{rot}(P) = -2$ (left) and cycle C with $\text{rot}(C) = 4$ (right).

To extend the definition of rotation to cycles consider a path with the same vertex as the source vertex of the first edge and as the target vertex of the last edge. Additionally to

the path rotation, the rotation between the first and last edge is accumulated. The *cycle rotation* is defined for a cycle $C = [e_1, ..., e_k]$ with length k as

$$\text{rot}(C) = \sum_{i=1}^{k} \text{rot}(e_i, e_{i+1}), \text{ where } e_{k+1} = e_1. \tag{4.4}$$

Figure 4.5 shows on its right a drawing of an example cycle. The graph drawing is based on a graph G with edges $e_1, ..., e_4$ and the cycle $C = [e_1, ..., e_4]$ has length $k = 4$. The cycle rotation is therefore $\text{rot}(C) = 4$. The *face rotation* of a face f is defined as the rotation of the cycle C that bounds f in clockwise direction that is

$$\text{rot}(f) = \text{rot}(C). \tag{4.5}$$

Figure 4.5 also shows on its right side an example face f, which is bounded by the cycle C. Because face f lies on the right side of cycle C and the rotation of C is $\text{rot}(C) = 4$ the rotation of f is also $\text{rot}(f) = 4$. Additionally, the rotation for a vertex is needed for the drawable characteristic explained in Section 4.1.3. A *vertex rotation* for a vertex v is defined as the summation over the rotations between consecutive edges in the circular order around v. For a vertex v with $\text{adj}(v)=[u_1, ..., u_k]$ we define the vertex rotation as

$$\text{rot}(v) = \sum_{i=k+1}^{1} \text{rot}(u_i v, v u_{i-1}), \text{ where } v u_{k+1} = v u_1.$$

4.1.2 Reading Rotations from a Graph Drawing

The first step of the reversed pipeline generates a graph representation $R(\Gamma)$ consisting of rotations, which is defined by a drawing Γ. When reading out a rotation between two edges uv, vw from a drawing, the angle between the vertices u and w at vertex v in counter clockwise direction starting from uv is crucial. For drawing a graph the rotation between neighboring edges is important to determine the distance with the vertices' coordinates. The rotation between not-consecutive edges is also important, for example, for determining the rotation of a path. For this subsection the graph in Figure 4.6 serves as an example. The figure contains two drawings that are based on the same orthogonal representation and therefore, also on the same graph.

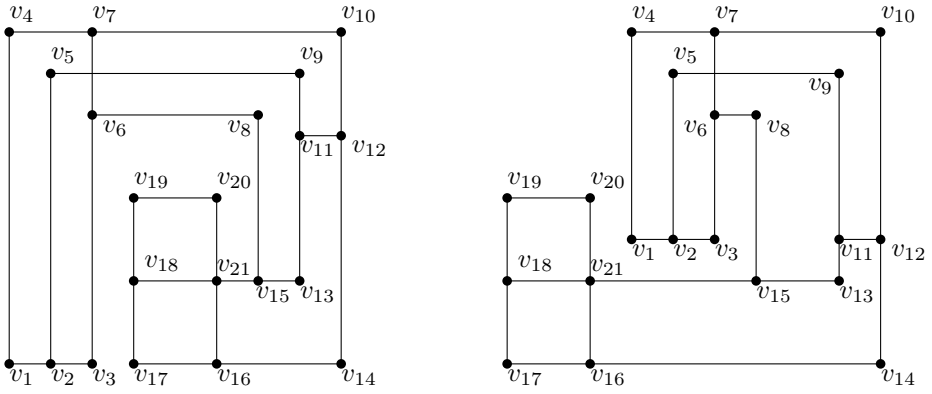


Figure 4.6: Two graph drawings based on the same graph representation.

The rotation between two consecutive edges can be easily extracted from the drawings. For example, the rotation between the edges v_2v_1, v_1v_4 is calculated by inserting the $\text{angle}(v_2v_1, v_1v_4) = 90°$ into the alternative Formula 4.2: $\text{rot}(v_2v_1, v_1v_4) = 1$. A rotation of

-1 can be found for the two edges $v_{12}v_{10}, v_{10}v_7$ and a rotation of 0 can be found for the two edges $v_{14}v_{12}, v_{12}v_{10}$. The path P, which is marked in blue in Figure 4.7, has two rotations accumulated and the rotation $\text{rot}(P) = -1$. The figure also contains a cycle C, marked in orange, with length $k = 7$. The rotation of C is the sum of rotation between consecutive edge pairs and is therefore $\text{rot}(C) = 4$. The face f, which is marked in brown, has the same rotation as its bounding cycle that is $\text{rot}(f) = 4$.

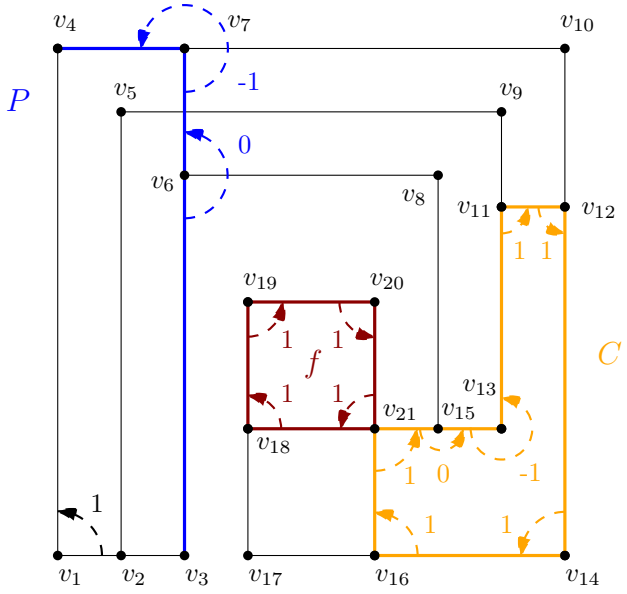


Figure 4.7: Drawing from Figure 4.6 with marked rotations for the examples.

4.1.3 Drawable Orthogonal Representations

A representation R is called *drawable* if an orthogonal drawing Γ exists so that $R = R(\Gamma)$. If the representation $R(\Gamma)$, which is extracted from Γ, is the same as the representation R, then R is drawable. So, a representation R is drawable if and only if G admits a drawing with the specified rotations. When an orthogonal representation is drawable, the characteristics introduced in the following two lemmas for rotations apply.

In the first lemma, requirements for the rotations around a vertex are set. In an orthogonal drawing Γ the angles between two consecutive edges around a vertex v sum up to 360°. As a result, in a drawable, orthogonal representation $R(\Gamma)$, the sum of rotations around a vertex have to apply to certain requirements, as they are representing these angles.

Lemma 4.1. *Let G be a planar, connected 4-graph and R be an orthogonal representation of G. If R is drawable, then every vertex v of G with degree deg(v) has the vertex rotation*

$$\text{rot}(v) = 2(\deg(v) - 2).$$

Proof. Since R is drawable, there exists a drawing Γ of G with $R = R(\Gamma)$. Let v be a vertex of G with degree $\deg(v)$. As the representation is orthogonal and the degree of 0 is not considered, v can have a degree between 1 and 4.

If $\deg(v) = 1$, v has only one edge e, which has an angle of 360° to itself around v. The rotation between the edges and also the rotation for the vertex is $\text{rot}(e, e) = \text{rot}(v) = -2$, which is the same as the degree inserted into the formula of the lemma: $2(\deg(v) - 2) = 2(1 - 2) = -2$.

If $\deg(v) = 2$, then two edges e_1, e_2 are incident to v. There are three possible angle combinations, which are shown in Figure 4.8, because the 360° angle around v is split into

the angles between the edges. The first two scenarios have each one angle of 90° and one angle of 270° and are illustrated on the left and on the right of the figure. In the middle of the figure is the third scenario shown, which has two 180° angles. In all scenarios the sum of the rotations is $\text{rot}(v) = 0$. Therefore, for a degree of 2 all possible angle distributions fulfill the lemma, as $2(\deg(v) - 2) = 2(2 - 2) = 0$.

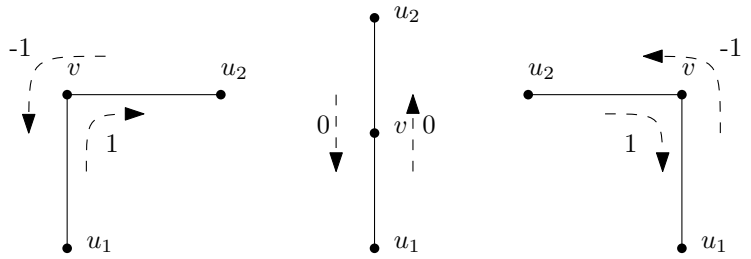


Figure 4.8: Possible rotations for a vertex v with $\deg(v)$=2.

If $\deg(v) = 3$, then the vertex v is incident to three edges e_1, e_2, e_3. In the drawing, the 360° around v can be distributed to one 180° angle and two 90° angles. This leads to three possible drawings of v, which result in the same rotation sum for v, namely $\text{rot}(v) = 1 + 1 + 0 = 2$. The possible drawings are shown in Figure 4.9. The rotation of vertex v then equals the formula of the lemma: $2(\deg(v) - 2) = 2(3 - 2) = 2$.

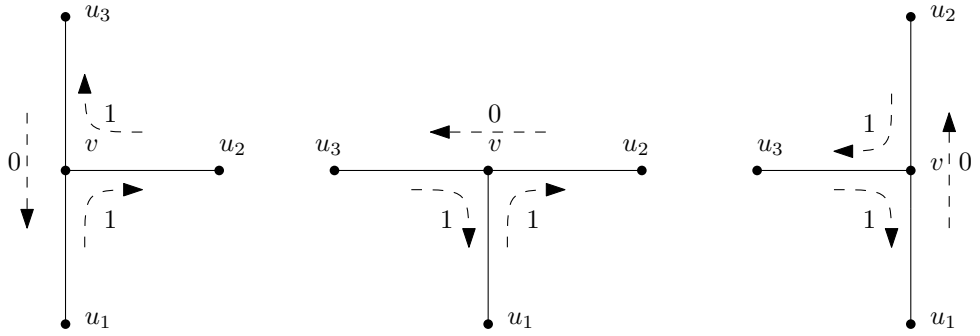


Figure 4.9: Possible rotations for a vertex v with $\deg(v)$=3.

If $\deg(v) = 4$, then vertex v is incident to four edges e_1, e_2, e_3, e_4. Therefore, the drawing contains four angles, which sum up to 360° and hence, each have 90°. The angles between the consecutive edges equal to rotation 1 and so, the rotation of v is $\text{rot}(v) = 4$, which is shown in Figure 4.10. The degree inserted into the formula of the lemma also results in: $2(\deg(v) - 2) = 2(4 - 2) = 4$. So, also the last case for the degree of v fulfills the lemma.

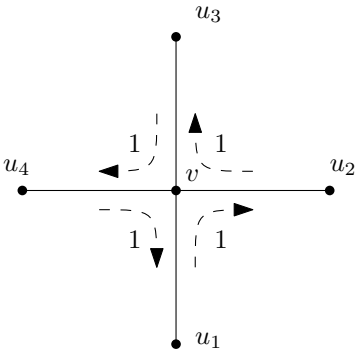


Figure 4.10: Possible rotations for a vertex v with $\deg(v)$=4.

□

The second lemma gives information about the rotation of the faces in a drawable orthogonal representation. In an orthogonal drawing a face is bounded by a polygon with k vertices. The sum of internal angles at the k vertices can be calculated with $\sum_{i=1}^{k} \alpha_i = (k-2) \cdot 180°$. The rotations of a drawable orthogonal representation represent this internal angle sum and therefore, they have to fulfill the requirements contained in the following lemma. Recall that the formula to convert an angle $\alpha \in \{90°, 180°, 270°, 360°\}$ to a rotation is

$$\text{rot}(\alpha) = \frac{180° - \alpha}{90°}. \tag{4.6}$$

Lemma 4.2. *Let G be a planar, connected 4-graph and R be an orthogonal representation of G. If R is drawable, then every regular face f in G has the face rotation* $\text{rot}(f) = 4$ *and the outer face F has the face rotation* $\text{rot}(F) = -4$.

Proof. Since R is drawable, there exists a drawing Γ of G with $R = R(\Gamma)$. Let f be an arbitrary regular face f and let F be the outer face. Because of the orthogonality, f is bounded by a polygon with k corners and its internal angle is calculated with the sum over the angles at the corners $\sum_{i=1}^{k} \alpha_i = (k-2) \cdot 180°$. The regular face f has rotation $\text{rot}(f) = 4$ because of the following equation.

$$\sum_{i=1}^{k} \text{rot}(\alpha_i) = \sum_{i=1}^{k} \frac{180° - \alpha}{90°} = \frac{1}{90°}\left(\sum_{i=1}^{k} 180° - \sum_{i=1}^{k} \alpha\right) = \frac{1}{90°} \cdot \left(k \cdot 180° - (k-2) \cdot 180°\right) =$$

$$= \frac{1}{90°} \cdot 2 \cdot 180° = 4$$

For the outer face the external angles at the corners of the border of F are deciding for the face rotation. As the outer face F is a polygon in the drawing of R , it contains k corners and the sum of its external angles is calculated with $\alpha_i = \overline{\alpha_i} = 360° - \alpha_i$ inserted in the equation above.

$$\sum_{i=1}^{k} \text{rot}(\overline{\alpha_i}) = \sum_{i=1}^{k} \frac{180° - \overline{\alpha}}{90°} = \sum_{i=1}^{k} \frac{180° - (360° - \alpha)}{90°} = \sum_{i=1}^{k} \frac{-180° + \alpha}{90°} =$$

$$= \frac{1}{90°}\left(\sum_{i=1}^{k} -180° + \sum_{i=1}^{k} \alpha\right) = \frac{1}{90°} \cdot \left(-k \cdot 180° + (k-2) \cdot 180°\right) = \frac{1}{90°} \cdot -2 \cdot 180° = -4$$

□

4.2 Augmenting Orthogonal Representations

Tamassia [Tam87] introduced an approach for proving a representation to be drawable based on the two lemmas in Section 4.1.3. Consider an orthogonal representation R. By requiring all faces to be rectangles, which require four rotations of 1, a drawing for R can be calculated with a flow network [Tam87]. An orthogonal representation that satisfies for each vertex v the rotation of Lemma 4.1 and for each face f the rotation of Lemma 4.2 has for each face a rectangle as the facial cycle. Therefore, for each representation that fulfills both lemmas a drawing exists.

The last step for constructing a drawing from a representation is computing the lengths of its edges and the coordinates of its vertices. If all faces are rectangles the edges fit into the grid layout for orthogonal drawings and the lengths of the edges can be easily computed with a flow network. Therefore, orthogonal representations with rectangles as faces are drawable.

To convert non-rectangular faces into rectangles, the representation is augmented with fictitious edges and vertices [Tam87]. As a rectangle contains four rotations of value 1, the conversion needs to extinguish all rotations of value -1, which form a left bend along the face boundary. The augmentation checks each regular face f of a graph G for the rotation sequence of -1,1,1, which is called a *U-Shape*, in the bounding cycle of f in clockwise direction. Vertices with rotation 0 can exist between the vertices of the U-Shape. Figure 4.11 shows a U-Shape and its augmented version.

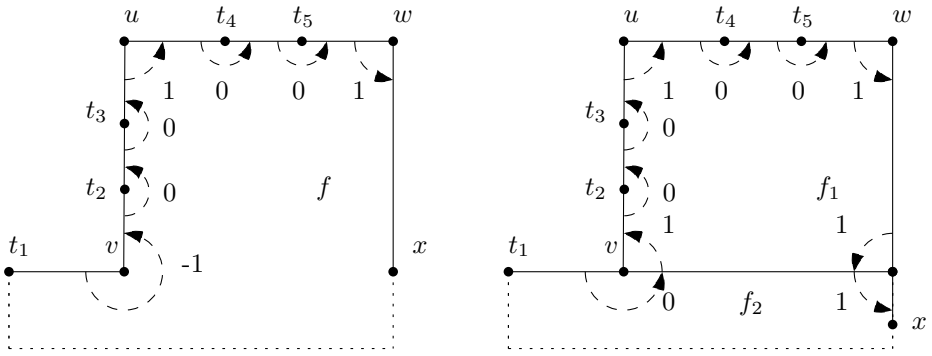


Figure 4.11: Augmenting a U-Shape.

The rotation sequence of -1,1,1 is contained in the border of f if it contains a sequence of edges with the rotations $\text{rot}(t_1v, vt_2) = -1$, $\text{rot}(t_3u, ut_4) = 1, \text{rot}(t_5w, wx) = 1$ in this order. Between these three rotations, the sequence can contain an arbitrary number of edges with rotation 0. This rotation sequence represents a left bend followed by two right bends that shape the face as a polygon and not a rectangle. By inserting a new edge from vertex v to a new vertex on wx, the original face f is split into two new faces, which need to be checked for U-Shapes respectively. If a face doesn't contain any more U-Shapes it contains only rotations of value 1 or 0 and is therefore a rectangle.

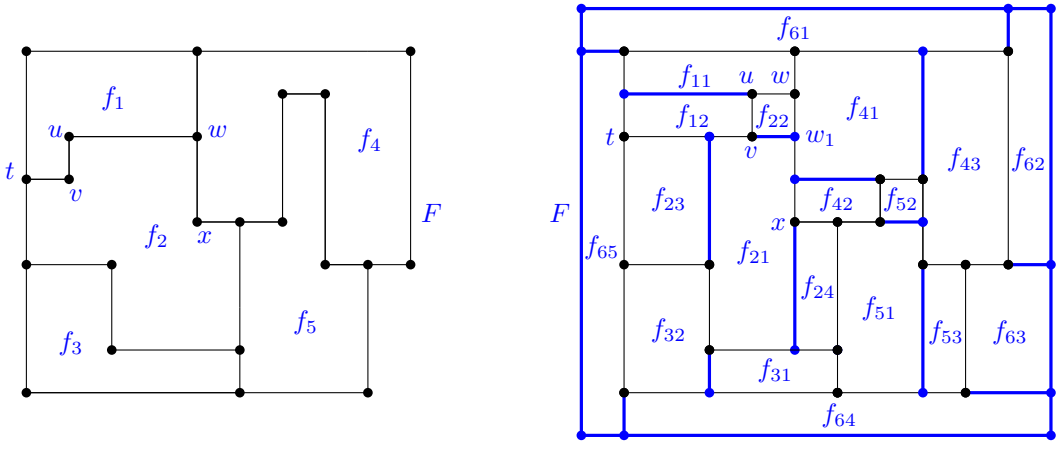


Figure 4.12: Augmenting a graph representation: drawing of the representation before (left) and after (right) the augmentation.

Figure 4.12 contains a drawing of an orthogonal representation R for a graph G on the left. A drawing of the possible result of the augmentation is shown on the right. Depending on the order of faces for checking and augmenting U-Shapes, other edges and vertices can be inserted during the augmentation. The inserted edges and vertices are marked in blue in the drawing. Although a drawing is included here for illustrations, note that the representation is not proven to be drawable until the augmentation is finished. The graph G contains in particular the vertices t, v, u, w, x. When augmenting the face f_2 a left turn

is found at tv, vu and two right turns for vu, uw and uw, wx. These edges form a U-Shape. A new vertex w_1 is inserted at the edge wx between the vertices w and x and a new edge vw_1 is inserted. The face f_2 is now split into two new faces f_{21} and f_{22}. Note also that, as the drawing of the original representation is only anticipated, the position of vertices can change. On the right side of Figure 4.12, the vertex v has other coordinates than in the left drawing because the new edge splitting f_{21} and f_{23} was inserted.

The outer face of an orthogonal representation R is a special case when augmenting R. The shortest way to form the border of the outer face to a rectangle is to insert edges around the former border of the outer face and create a new border as a rectangle. It is important to connect the new border with the former border, as only connected graphs are proven drawable with the augmentation. When all faces are checked for U-Shapes and are augmented, then the representation contains only rectangles as faces. Two flow networks are used to compute the lengths of the edges of the graph, one for the horizontal edges and one for the vertical ones. The coordinates for the vertices can be chosen afterwards based on the lengths of the edges and the existence of a drawing is proven.

4.3 Computing a Graph Drawing

The last step of the pipeline computes a drawing for a graph representation. More exact explanations are included in the paper of Battista et al. [BETT98]. Let R be a drawable orthogonal representation. The metrics for an orthogonal drawing are the lengths of the edges and the coordinates for the vertices. The basic idea is computing the lengths for the horizontal and vertical edges separately and afterwards based on the lengths the coordinates. For this step we need an augmented graph representation as presented in the section before, so that the faces are all forming rectangles. Figure 4.13 shows the two network flow algorithms to compute the lengths of the edges. Note that a new sink vertex and new source vertex are needed in both computations. The flows over the network flow edges marked in blue give the length of an edge relative to the other edges. Afterwards, an exact length can be used for one unit of flow. The coordinates are then assigned for one vertex and recursively distributed based on the lengths of the edges to the next vertices.

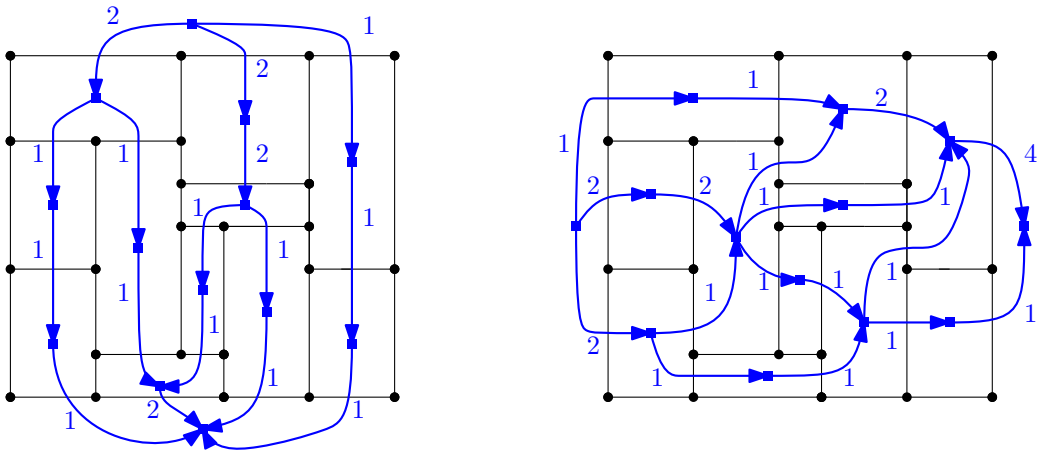


Figure 4.13: Computing the lengths of the edges with network flow algorithms.

5. Orthoradial Graph Drawing

The pipeline of the TSM framework can be transferred on orthoradial graph drawing. This chapter gives an overview over the similarities and differences to the orthogonal case. Important aspects as finding and repairing monotone cycles, as well as the effects of these cycles on the augmentation are explained. This chapter is based on the works of Barth et al. [BNRW17a] and Niedermann et al. [NRW13]. Orthoradial graph drawings are embedded in an orthoradial grid, which is formed by M concentric circles and N spokes, where $M, N \in \mathbb{N}$. The grid and its center, which is labeled with 'C', are shown on the right side of Figure 5.1. Vertical and horizontal edges as a classification also appear in orthoradial drawings. The edges that are on a part of a concentric circle drawn are horizontal edges. The straight spokes represent the vertical edges.

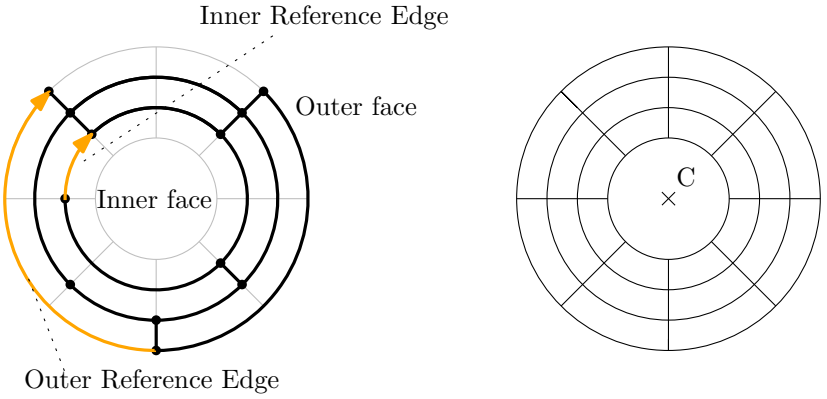


Figure 5.1: On the left, inner and outer face and their reference edges. On the right, orthoradial grid with its center marked.

Orthoradial representations have an outer face and also an inner face, which is placed at the center of the graph drawing. The inner face's border is the innermost cycle, while the outermost cycle is the facial cycle for the outer face. The inner and outer face are each described by a selected edge, a *reference edge*. Figure 5.1 shows on its left an example drawing with its inner and outer face and the reference edges marked. On the right side of the *inner reference edge* lies the inner face and on the left side of the *outer reference edge* lies the outer face.

The coordinates for the vertices in an orthoradial drawing are more practical in the polar coordinate system than the Cartesian coordinate system. Figure 5.2 shows the

interpretation of the polar coordinates for vertices. Instead of the x and y coordinates, we have an angle from the normally positive x-axis and a radius from the grid's center for each vertex. The start axis for the angle is marked in blue. Starting from the axis the angle to a vertex in counter-clockwise direction is the desired angle. The orange lines in the figure show the radius for both example vertices.

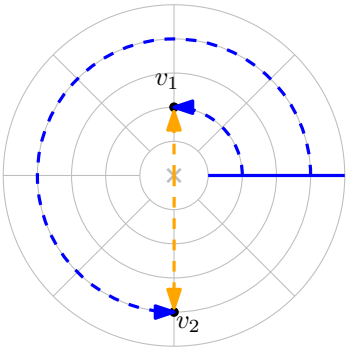


Figure 5.2: Polar coordinates of two vertices.

In a 3-dimensional drawing the polar coordinates are also identifying explicit coordinates and the drawing can be shown on the surface of a 3-dimensional cylinder. The radius from the center's grid is the respective height and the angle remains the same. Figure 5.3 shows an orthoradial drawing on the orthoradial 2-dimensional grid and on the surface of a 3-dimensional cylinder.

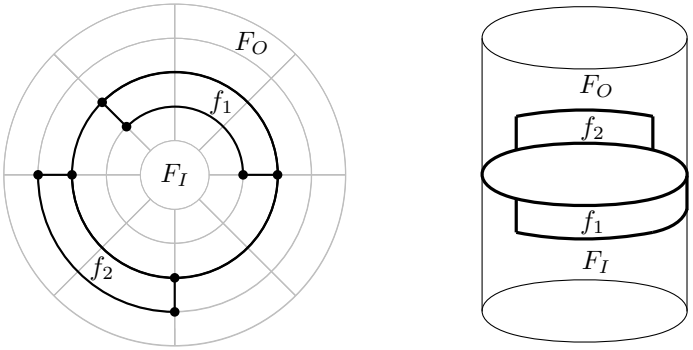


Figure 5.3: Orthoradial drawing on a grid and on the surface of a cylinder. The inner face is marked with F_I, the outer with F_O.

The orientation of edges is an additional characteristic that represents the direction relative to the outer reference edge. The orientation distinguishes horizontal and vertical edges and is useful for the following sections. It also represents the information whether an edge is directed in clockwise or counter clockwise direction. The outer reference edge is always horizontal and directed clockwise. Depending on the rotation between two adjacent edges, the orientation of the second edge can be computed if the orientation of the first is known. An edge has one of the four possible orientations that are $\{0, 1, 2, 3\}$. The even numbers represent the horizontal lines and the uneven vertical ones. The orientation 0 is therefore horizontal and additionally, in clockwise direction. The orientation 2 is horizontal and in counter-clockwise direction. The vertical edges that are directed towards the inner face have orientation 1. The vertical edges that are directed towards the outer face have orientation 3. The orientations are shown in Figure 5.4.

For computing an orthoradial graph drawing, the pipeline for orthogonal drawings introduced in Chapter 4 can be used as a template. The steps from a graph up to an orthoradial

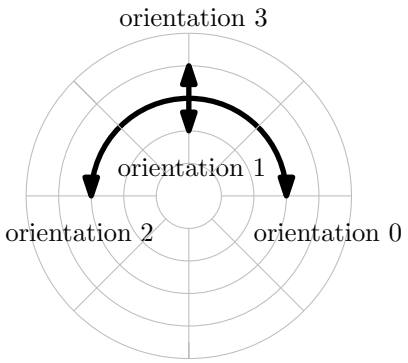


Figure 5.4: Orientation for edges.

graph representation are similar to the orthogonal pipeline. The detailed computation differs slightly as the requirements for the face rotations are not equal to the orthogonal case. However, the structure of the embedding or representation respectively are identical. In the orthoradial and orthogonal case the first step from a graph to a graph embedding chooses the order of edges around a vertex, so that the planarity of the graph is given. The embedding can be chosen with a standard planar embedding algorithm, which are included in the work of Patrignani [Pat13] and not further explained in this work. The second step from a graph embedding to a graph representation in the orthoradial case needs to compute the rotations for each consecutive edge pair around each vertex of the graph. For the orthoradial case the face rotation differs from the orthogonal ones, as we have an outer and an inner face. In Section 5.1 the requirements for a representation to be drawable are explained for the orthoradial case. However, the definition of a drawable representation differs between the orthoradial and the orthogonal case. Another criterion is needed for the representation to be drawable. An orthoradial drawing cannot be computed from a representation if the latter contains so called *monotone cycles*. Section 5.2 explains monotone cycles and Section 5.3 shows how to find them. Also, Section 5.4 explains how to fix a monotone cycle.

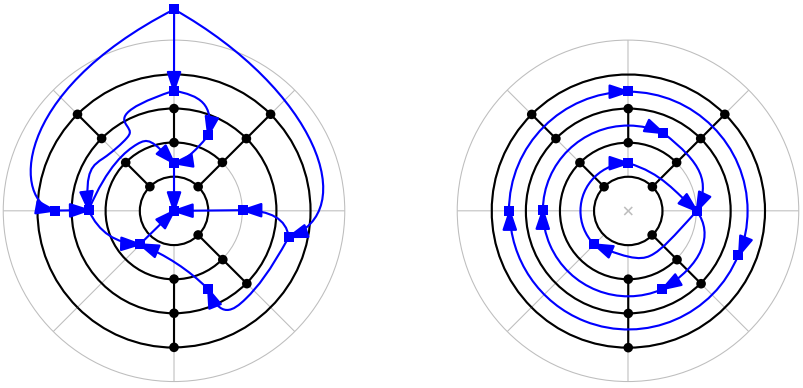


Figure 5.5: Computing the lengths of the edges.

The last step of the pipeline computes the metrics for an orthoradial drawing. The edges' lengths need to be fixed. Like in the last step of the orthogonal pipeline, two network flow calculations can be used, one for horizontal and one for vertical edge lengths. An important difference is, when the flow is computed, the angle covered by the horizontal edges on a concentric circle need to sum up to 360°. Therefore, we search for the concentric circle with the most flow units on its horizontal edges and split the 360° through this number. Another difference is that the vertical flow network needs no new sink or source as it is circled in itself. The Figure 5.5 shows the network flow graphs.

5.1 Drawable Orthoradial Representations

The computation of a graph representation in the orthoradial case needs to fix the rotations for each consecutive edge pair around each vertex. For the representation to be drawable similar lemmas to Section 4.1 are needed. We define $R(\Gamma)$ again as the representation that is extracted from a drawing Γ. An orthoradial representation R is drawable if an orthoradial drawing Γ exists so that $R = R(\Gamma)$.

In orthoradial representations the definition of rotations from orthogonal representations can be reused. The rotations of an orthoradial representation are also $\in \{1, 0, -1, -2\}$ and describe the same angles in the drawing. The vertex rotation, face rotation, path rotation and cycle rotation are defined as in Section 4.1.1. Additionally, the Lemma 4.1 of Section 4.1.3 also applies to orthoradial representations, because the geometric situations are identical. In particular the angles between consecutive edges around a vertex also sum up to 360°. The proof for the new Lemma 5.1 is structured as the proof for Lemma 4.1.

Lemma 5.1. *Let G be a planar, connected 4-graph and R be an orthoradial representation of G. If R is drawable, then every vertex v of G with degree* $\deg(v)$ *has the vertex rotation*

$$\mathrm{rot}(v) = 2(\deg(v) - 2).$$

The second lemma for drawable orthogonal representations, which concludes the face rotation for each face, can not be transferred to the orthoradial case directly. This is the case, because in contrast to the orthogonal case, an orthoradial representation has an inner and an outer face. If the inner and outer face are the same face, then the representation equals an orthogonal one and Lemma 4.1 applies. In the following the inner and outer face are always considered as different faces. In order to present and prove a new lemma for the face rotation for the orthoradial case, following lemma allows to compute the internal angle of a polygon in the orthoradial grid. It is based on the angles $\alpha_i = \mathrm{angle}(v_{i-1}v_i, v_iv_{i+1})$ for the vertices $v_1, ..., v_k$ of the polygon, where $v_0 = v_k$ and $v_{k+1} = v_1$.

Lemma 5.2. *A polygon P with k vertices $v_1, ..., v_k$ as corners in the orthoradial grid has the internal angle sum defined by the formula*

$$\sum_{i=1}^{k} \alpha_i = \begin{cases} k \cdot 180° & \textit{if the grid's center lies in the interior of } P \\ (k-2) \cdot 180° & \textit{otherwise.} \end{cases}.$$

Proof. We prove the formula by induction over the number of vertices of a polygon, as the internal angle sum formula depends on the inclusion of the center of the grid.

Base cases: This induction has two base cases, as it depends on the inclusion of the grid's center in the polygon. Figure 5.6 shows the two base cases. If the grid's center is not contained, then the base case is a polygon with four vertices v_1, v_2, v_3, v_4. Less than four vertices can not form a polygon in the orthoradial case with the center excluded. All four vertices have an internal angle of 90°.

$$\sum_{i=0}^{k} \alpha_i = 90° \cdot 4 = 360° = (4-2) \cdot 180° = (k-2) \cdot 180°$$

The other base case contains the grid's center. The polygon forms a circle around the center and can have a variable number k of vertices, which all have 180°.

$$\sum_{i=0}^{k} \alpha_i = k \cdot 180°$$

In either case the formula holds.

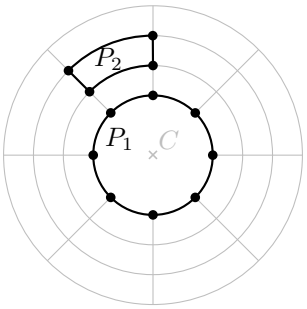


Figure 5.6: Base cases for the induction, polygons P_1, P_2. The grid's center lies in the interior of P_1, but not in the interior of P_2.

The inductive step needs beforehand the procedure for splitting a polygon into two strictly smaller polygons. Let P be an arbitrary polygon in the orthoradial grid. Then P can be split into two truly smaller polygons P_1, P_2. Let $P = [u, ..., v]$ be the outermost, counter-clockwise directed path, i.e. the vertices u, v have the maximal radius to the grid's center. Also, P is chosen as long as possible. We now look at a cone in the drawing formed by the grid's center and u and v. However, the edge uv is excluded. Figure 5.7 shows the cone for an example polygon on the left in brown. Then, choose a vertex t that lies in the cone. Therefore, if the angle for v is θ_1 and the angle for u is θ_2, then the angle θ_3 of t is $\theta_3 \leq \theta_1$ and $\theta_3 \geq \theta_2$. For the scenario that uv intersects with the start axis of the polar coordinates' angles, that means $\theta_2 > \theta_1$, take $\theta_1 = \theta_1 + 360°$ and $\theta_3 = \theta_3 + 360°$. Also, vertex t is chosen so that t has the greatest angle and radius among all vertices of P in the cone. Note that the radius of t to the grid's center is smaller than the one of v or u, as these vertices were chosen with the greatest radius and the edge uv is excluded from the cone. Then a new edge tw can be inserted to a vertex w that has the same radius as t to the grid's center and the second highest angle between v and u. The vertex w can be inserted on the edge from u towards the center if it was not existent before. If vertex t is found at this edge from u, then a new vertex w can be inserted, so that t and w are switched from the scenario above and edge wt is inserted. The two new polygons P_1, P_2 are truly smaller, as both have fewer vertices than P. Let c be $c = 1$ if a new vertex for t or w was inserted to the polygon and $c = 0$ otherwise. The following formula, where $k(P)$ gives the number of vertices of a polygon P, is valid, that is

$$k(P_1) + k(P_2) = k(P) + 2 + c \tag{5.1}$$

If both vertices t and w existed before splitting P, then they are counted twice, once for each new polygons P_1, P_2. Therefore, $k(P_1) + k(P_2)$ has two more vertices. If either t or w did not exist in P, then this new vertex is present in both new polygons. Therefore, the formula for the new polygons has three more vertices.

Inductive step: Given a polygon P with k vertices. Then P can be split into two new polygons P_1, P_2 as explained above, so that P_1 and P_2 are both truly smaller than P. If one vertex for t or w was inserted while splitting, then the internal angle sums of the polygons P1, P2 include an additional angle, which is not present in P. As the new vertex in P has an angle of 180°, we have to subtract 180° from the formula if a new vertex for t or w was inserted. Let c be again $c = 1$ if a new vertex was inserted to the polygon and $c = 0$ otherwise. The internal angle sum of P is computed by the formula, where $e_i = v_i v_{i+1}$, that is

$$\sum_{v_i \in P} \alpha_i = \sum_{v_i \in P_1} \alpha_i + \sum_{v_i \in P_2} \alpha_i - c \cdot 180°. \tag{5.2}$$

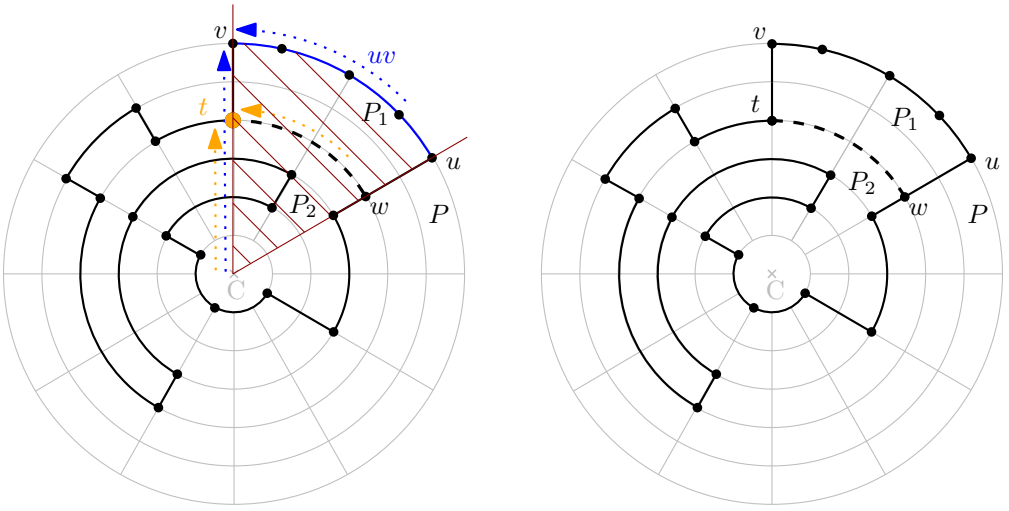


Figure 5.7: Splitting a polygon in an orthoradial grid with a new edge tw. On the left, finding uv and t. On the right, result after splitting.

Also, it has to be differentiated whether P contains the grid's center or not. If P does not contain the center, then neither do P_1 or P_2. Then, Formula 5.1, by the inductive hypothesis, and Equation 5.2 yield the following

$$\sum_{v_i \in P} \alpha_i = (k(P_1) - 2) \cdot 180° + (k(P_2) - 2) \cdot 180° - c \cdot 180 = (k(P_1) + k(P_2) - 4 - c) \cdot 180° =$$

$$= (k(P) - 2) \cdot 180°$$

If P does contain the center, then, without loss of generality, P_2 also contains the center. Equation 5.2 yields the following

$$\sum_{v_i \in P} \alpha_i = (k(P_1) - 2) \cdot 180° + k(P_2) \cdot 180° - c \cdot 180° = (k(P_1) + k(P_2) - 2 - c) \cdot 180° = k(P) \cdot 180°$$

□

Lemma 5.3. *Let G be a planar, connected 4-graph and let R be an orthoradial representation of G. If R is drawable, then every regular face f in R has the face rotation* $\mathrm{rot}(f) = 4$ *and the inner and outer face F_I, F_O have the face rotation* $\mathrm{rot}(F_I) = \mathrm{rot}(F_O) = 0$.

Proof. Since R is drawable, there exists a drawing Γ of G with $R = R(\Gamma)$. Let f be a regular face and let F_I and F_O be the inner face and the outer face. The face f is bounded by a cycle C, which forms a polygon P in a drawing. The number of vertices as the corners of the polygon is $k = k(P)$. Recall that the formula to convert an angle $\alpha \in \{90°, 180°, 270°, 360°\}$ to a rotation is

$$\mathrm{rot}(\alpha) = \frac{180° - \alpha}{90°}. \tag{5.3}$$

As the polygon P does not contain the center of the grid, by Lemma 5.2 the face rotation is $\mathrm{rot}(f) = 4$ as

$$\sum_{i=1}^{k} \mathrm{rot}(\alpha_i) = \sum_{i=1}^{k} \frac{180° - \alpha_i}{90°} = \frac{1}{90°} \cdot (\sum_{i=1}^{k} 180° - \sum_{i=1}^{k} \alpha_i) = \frac{1}{90°} \cdot (k \cdot 180° - (k-2) \cdot 180°) = 4.$$

The inner face is bounded by a cycle C_I, which again bounds a polygon P_I containing the grid's center in a drawing. Therefore, by Lemma 5.2 the face rotation $\text{rot}(F_I)$ is

$$\sum_{i=1}^{k} \text{rot}(\alpha_i) = \sum_{i=1}^{k} \frac{180° - \alpha_i}{90°} = \frac{1}{90°} \cdot (\sum_{i=1}^{k} 180° - \sum_{i=1}^{k} \alpha_i) = \frac{1}{90°} \cdot (k \cdot 180° - k \cdot 180°) = 0.$$

The outer face F_O is bounded by a cycle C_O. Because the opposite angles of C_O are the internal angle of a polygon that contains the grid's center, by Lemma 5.2 the rotation of F_O is

$$\sum_{i=1}^{k} \text{rot}(\alpha_i) = \sum_{i=1}^{k} \frac{180° - \alpha_i}{90°} = \frac{1}{90°} \cdot (\sum_{i=1}^{k} 180° - \sum_{i=1}^{k} (360° - \overline{\alpha_i})) =$$

$$= \frac{1}{90°} \cdot (k \cdot 180° - k \cdot 360° + \sum_{i=1}^{k} \overline{\alpha_i}) = \frac{1}{90°} \cdot (k \cdot 180° - k \cdot 360° + k \cdot 180°) = 0.$$

□

5.2 Monotone cycles

For an orthogonal representation, Lemma 4.1 and Lemma 4.2 are sufficient for the representation to be drawable. The equivalent requirements for an orthoradial representation are included in Lemma 5.1 and Lemma 5.3. This section deals with whether these lemmas are sufficient for orthoradial representations. For this consider as an example an orthoradial graph representation R for a graph G with all rotations complying with the two lemmas for rotations. Given three edges e_1, e_2, e_3, we set the rotation for these edges: $\text{rot}(e_1, e_2) = -1$, $\text{rot}(e_2, e_3) = 1$, $\text{rot}(e_3, e_1) = 0$. Consider the cycle $C = [e_1, e_2, e_3]$ as an essential cycle that is directed in clockwise direction. When drawing the cycle C in an orthoradial grid, the edge e_1 has a left turn to e_2 and e_2 a right turn to e_3. The edge e_3 then goes without a bend to e_1 although they would be on different concentric circles of the grid. Because we have the requirement of bend free edges and the edge e_3 needs to be drawn on two different concentric circles, there exists no drawing for this representation. Figure 5.8 shows a sketch for this graph.

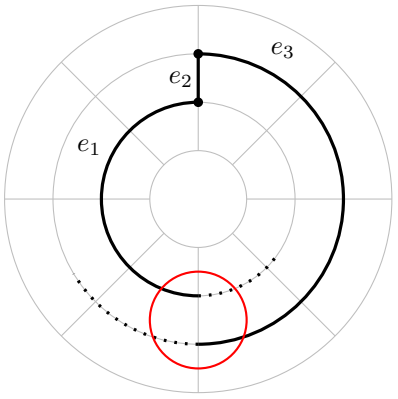


Figure 5.8: Problem with the levels of the concentric circles in cycle $C = [e_1, e_2, e_3]$.

So, orthoradial representations are not always drawable if the two lemmas for the rotations are fulfilled. On closer inspection, these problems occur, when the representation contains special cycles. These fulfill the requirements for the rotations like the example above does. Hasheminezhad et al. [HHT09] characterize drawable cycles and Barth et al. [BNRW17a]

establish based on their work the following definitions. An orthoradial representation has two kinds of cycles, the *regular cycles* and the *essential cycles*. An essential cycle has the inner face in its interior enclosed and has a cycle rotation of 0. A regular cycle does not enclose the inner face in its interior and has a cycle rotation of 4. Figure 5.9 shows two cycles C_1, C_2 that are representing these two categories. In the following an essential cycle is always considered directed so that the interior of the cycle is on the right side of each edge, which is equivalent to a clockwise directed cycle.

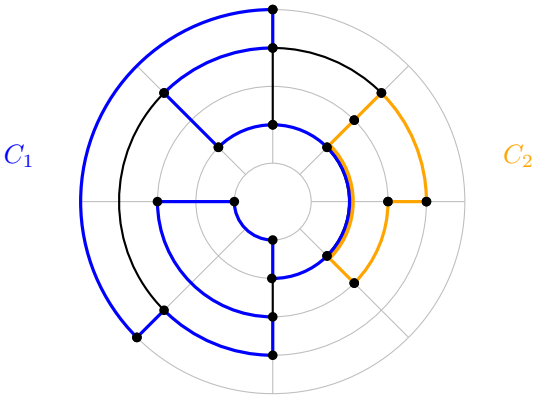


Figure 5.9: Orthoradial drawing with an essential cycle C_1 and a regular cycle C_2.

Essential cycles can be drawable and not drawable, so two subcategories have to be distinguished again. Essential Cycles can be categorized into *monotone* and *non-monotone cycles*. The example above contains a monotone cycle. In order to draw an orthoradial representation, it has to be free of monotone cycles because these cannot be drawn. A monotone cycle can be contained in a representation even if all rotations are fulfilling the two lemmas for rotations.

Monotone cycles can either be *increasing* or *decreasing*. The cycle $C = [e_1, e_2, e_3]$ in the figure above is an increasing cycle. When going along the monotone cycle in clockwise direction and switching to a concentric circle that is closer to the outer face, the cycle is increasing. When going along the monotone cycle in counter-clockwise direction and switching to a concentric circle that is further away from the outer face, the cycle is decreasing. However, this does not mean that the graph is not drawable. It only means that the chosen representation with the rotations above leads to a monotone cycle. Another orthoradial representation for the same graph embedding may be drawable. If for the example cycle C above all rotations are chosen with rotation 0, then C is a non-monotone cycle.

A cycle can be characterized as increasing or decreasing by *labeling* the edges of the cycle. Let e^* be the outer reference edge and let s be the target vertex of e^*. Let C be an essential cycle in the graph. Then, there exists an elementary, arbitrary path P from s to a vertex v on C. Recall that the path P is elementary if it intersects C only at its endpoints. Let $P_1 + P_2$ be the concatenation of two paths P_1, P_2. We define for an edge e on the cycle C the function for labeling $l_C^P(e) = rot(e^* + P + C\,[v, e])$, where $C\,[v, e]$ describes the path from v to e on the cycle C. Barth et al. [BNRW17a] proved the labeling to be independent of the choice of path P. A label on an edge depends on the outer reference edge of the underlying representation and the rotations of the cycle's edges. Figure 5.10 shows the labeling of a cycle C. A decreasing cycle has only non-negative labels and at least one strictly positive label on its edges. An increasing cycle has only non-positive labels and at least one strictly negative label. If the labels on a cycle are all 0 or the cycle contains both strictly positive and strictly negative labels, then the cycle is non-monotone.

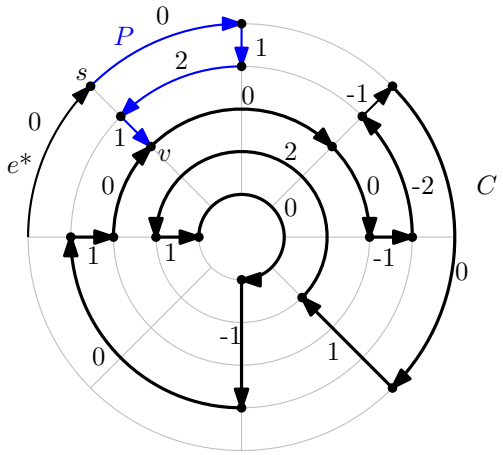


Figure 5.10: Graph Drawing with an essential cycle C, the reference edge e^* and a path P from e^* to C. The label of the edges are included.

5.3 Finding monotone cycles

An orthoradial representation needs to be checked for each essential cycle if it is monotone. However, there may be exponentially many monotone cycles contained. Niedermann et al. [NRW13] introduced an algorithm to check for monotone cycles. Instead of checking each cycle, each edge of the representation is examined for a decreasing cycle that contains the edge. The increasing cycles can be found, when testing for decreasing cycles again in the mirrored representation, as an increasing cycle is representing a decreasing cycle in the mirrored representation and vice versa. Because all edges are reversed in the mirrored representation and then tested again, it is sufficient to test a cycle only in clockwise direction, once in the normal and once in the mirrored representation, to find all monotone cycles.

Given a graph G with an orthoradial representation R. The algorithm for finding the outermost decreasing cycle introduced in the paper of Niedermann et al. [NRW13] finds a decreasing cycle if the representation R contains one. This process searches for each edge e of G the outermost, decreasing cycle that contains e, where the label of e is the smallest of the labels on the cycle. A decreasing cycle is the outermost one if it is not contained in the interior of any other decreasing cycle. Niedermann et al. [NRW13] prove the fact that the outermost cycle is unique. The outermost, decreasing cycle for an edge e can be found with a left-first DFS that visits each vertex at most once. Starting with e, the algorithm visits the next vertex via the next outgoing edge in clockwise direction from the target vertex of e. However, certain outgoing edges are excluded for the DFS because these would create negative labels on the cycle and it would not be a decreasing cycle. Also, these labels would be smaller than the label of e, which is assumed to have the smallest label. To determine these excluded edges a search label is calculated for choosing the next outgoing edge. Consider the DFS searching the next outgoing edge from the edge e_{ref}. An outgoing edge e from e_{ref} has the search label $\mathrm{sl}(e) = \mathrm{sl}(e_{ref}) + \mathrm{rot}(e_{ref}, e)$. The edges with a negative search label are filtered out, as these edges would have a smaller label than e and e is assumed to have the smallest label. Note that the first edge of the DFS gets search label 0. The routes of the search tree of the DFS that contain the start edge are *candidate cycles*. A candidate cycle is a decreasing cycle if it is essential, all labels on the cycle are non-positive and it has at least one strictly positive label. If the search tree contains no decreasing cycle, then the start edge of the DFS is not contained by a decreasing cycle, where it has the smallest label.

Lemma 5.4. *Let G be a planar, connected 4-graph and R be an orthoradial representation of G. If R contains a decreasing cycle, the algorithm finds an decreasing cycle even if only the edges on the essential cycles of G are tested that are directed clockwise.*

5.4 Computing a Drawable Orthoradial Representation

This section deals with computing a drawable orthoradial representation. Firstly, an orthoradial representation is calculated with a network flow algorithm from an embedding, so that the two Lemmas 5.1 and 5.3 from Section 5.1 are fulfilled for each vertex and face. This calculation can also be transferred to the orthogonal case, when working with the face rotations for orthogonal representations. Secondly, the orthoradial representation needs to be checked for monotone cycles. If it does not contain one, then it is drawable. If a monotone cycle exists, a simple extension on the cycle that is explained later in this section can convert the cycle to non-monotone.

When a graph embedding is given, the rotations for a graph representation can be computed with a network flow calculation. This step minimizes the bends in the resulting representation. The idea is to consider the two Lemmas 5.1 and 5.3 and to distribute rotations to vertices and faces, so that the lemmas are fulfilled. The input for the network flow algorithm is a graph. The vertices have a *demand* or *supply* of units, here a unit is a rotation, and the edges allow the units to be relocated. Let E be an embedding for a graph G. A new graph based on G with additional vertices and edges is used as an input graph for computing the rotations. The additional vertices, which are called *face vertices* in the following, represent the faces of G. The additional edges are inserted for the flow of rotation between a face vertex and a vertex or between face vertices. Therefore, the first step for getting an orthoradial representation is to build this new graph for the network flow algorithm. Note that finding the faces can be done by considering the embedding of the graph. Figure 5.11 shows an input graph for the network flow split into two drawings. The blue vertices are the face vertices and all blue edges in both drawings are the additional, inserted edges to G. The left drawing only shows the new edges between face vertices. For each edge of G separating an adjacent face pair a new edge is inserted. The flow on these edges represents the bends to be inserted at the edge separating the faces. Note that two face vertices v_1, v_2 have as many edges connecting them as edges separating the original faces. The right drawing shows the new edges between the original vertices and the face vertices. New edges are inserted from every vertex of G to their incident faces.

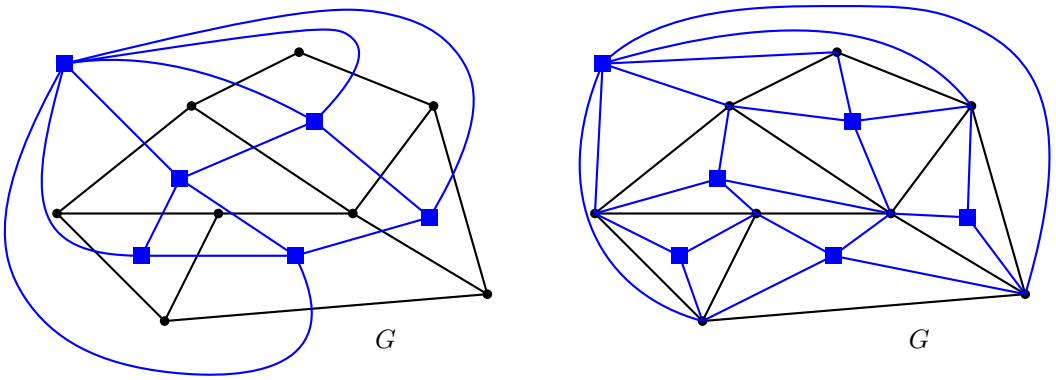


Figure 5.11: Graph G extension for calculating the rotations with a network flow algorithm. On the left, edges between face vertices. On the right, edges between vertices of G and face vertices.

The vertices in a network flow graph have the property of supply or demand. Because of Lemma 5.1, a vertex of G needs a rotation value based on its own degree, which is then

its supply. For example, let vertex v have a degree of 2, then its supply is 0. Because of the degree of 2, v is incident to two faces, which then get, for example, both a flow of 0 from v. The rotation of v at these faces is then 0. The face vertices have a demand of 0, if they represent the outer or inner face, or a demand of 4 for a regular face. This is the case, because of Lemma 5.3 and its requirements for face rotations. The inner and outer face need to be fixed before the network flow algorithm, so that their demands for the face rotation are set correctly.

The edges in a network flow graph have the properties of cost, lower bound and upper bound. The cost of an edge represents the cost of routing one flow unit along that edge if a unit flows over that edge. The upper and lower bound represent the capacity of an edge. The original edges of G have a cost of 0 and a lower and upper bound of 0. Therefore, no rotation unit can flow over the original edges. The inserted edges between original vertices and face vertices have a cost of 0, a lower bound of 0 and an upper bound of 1, so a face vertex can get a rotation of 0 or 1. The rotation -1 is represented with a flow of one unit from a face vertex to an original vertex. The edges between face vertices represent bends on the edges separating original faces. They have a cost of 1 and each flow unit over one of these edges is converted into a bend. The upper bound of the edges between face vertices is chosen as high as possible, because more than one bend can be necessary. The lower bound is again 0. Figure 5.12 shows on its left a part of an input graph, which consists of two face vertices v_1, v_2 and the vertices u_1, u_2, u_3 around v_1. As the face of v_1 has three incident vertices, it gets three rotation units from these. Because v_1 has a demand of 4, the result is a flow from the face vertex v_2 to the face vertex v_1. Consider the flow to be valid. The right side of the figure shows the original graph with the computed rotations and a new bend represented by a new vertex t.

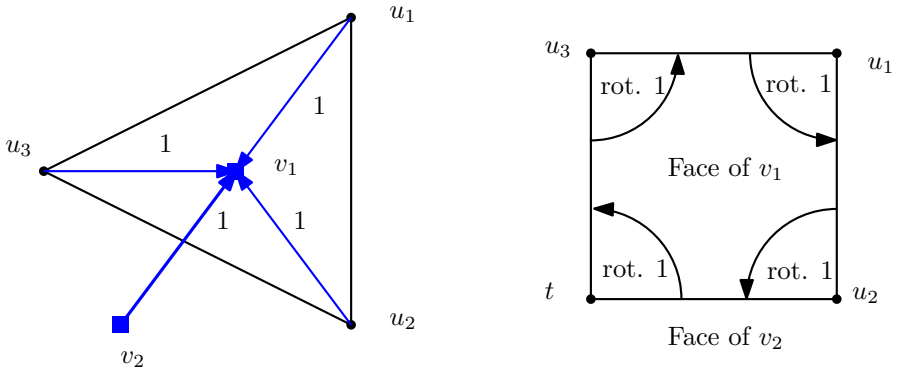


Figure 5.12: Flow between two face vertices on the left and the resulting bend on the right.

The used flow is a minimum-cost flow to get a graph with a minimum number of bends. A minimum-cost flow problem can be solved in polynomial time. Cornelson et al. [CK11] present an algorithm to prove a min-cost flow problem on a planar bidirected graph with bounded costs and face sizes solvable in $O(n^{3/2})$ time.

After the rotations are fixed, the orthoradial representation has to be checked for monotone cycles. In order to make the representation drawable, the monotone cycle can be made to non-monotone by adding new edges and vertices. When considering the labeling for a monotone cycle, it either contains only positive labels with at least one strictly positive one or only negative labels with at least one strictly negative one. The idea of the fix is to insert edges and vertices formed as a spiral until the cycle also contains labels with the missing sign, as a cycle that contains both strictly positive and negative labels is non-monotone. The fix is explained in the following for a decreasing cycle C in an orthoradial representation R. As C is decreasing, it only contains positive labels and at least one strictly positive one. Therefore,

it has a smallest label on an edge vu that has label 0 or greater. The edge vu is then split into five new edges $vw_1, w_1w_2, w_2w_3, w_3w_4, w_4u$. The new edges get following rotations: $\text{rot}(vw_1, w_1w_2) = -1, \text{rot}(w_1w_2, w_2w_3) = 1, \text{rot}(w_2w_3, w_3w_4) = 1, \text{rot}(w_3w_4, w_4u) = -1$. Therefore, with the new bends in the order left, right, right, left the labels on C need to be recalculated. Figure 5.13 shows this process on the left. Let the label on the edge vu be x, then the label on vw_1, w_2w_3, w_4u are also x, but on w_1w_2 it is $x-1$ and on w_3w_4 it's $x+1$. If $x = 0$, then C already has a strictly negative label on w_1w_2 and is non-monotone. If $x > 0$, then we have to repeat this process of adding in edges and vertices x times to get a negative label. The second repetition splits the edge w_1w_2 into four new edges, where only the last rotation to w_2w_3 is now 0. The second one of the new edges of the second repetition can then be split again into four new edges until the repetition is at step x. The first and second repetition are shown in Figure 5.13. The result after five repetitions is shown in Figure 5.14.

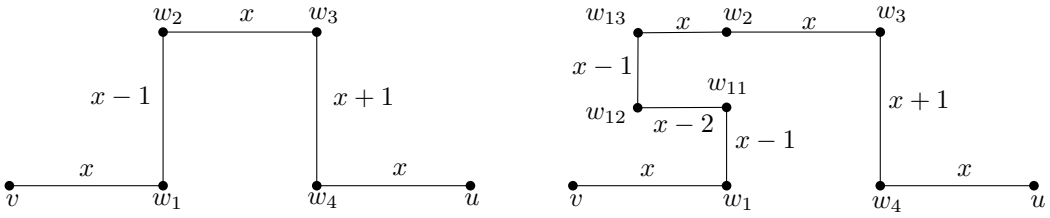


Figure 5.13: First and second repetition of inserting a spiral with the labels of the edges.

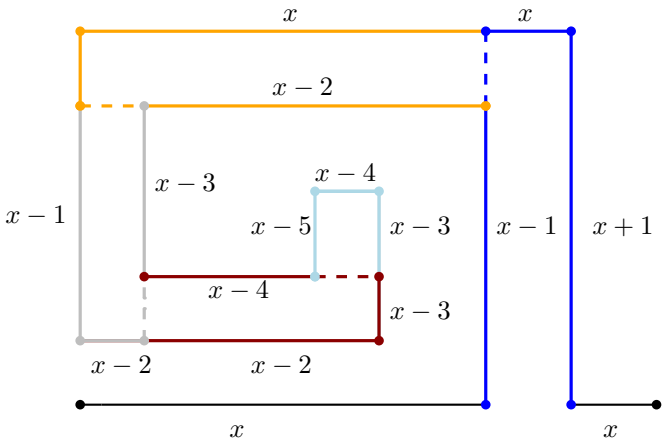


Figure 5.14: Spiral with the labels for the edges. The five repetitions are respectively marked with different colors.

5.5 Augmentation

This section transfers the augmentation of orthogonal representations to the orthoradial ones. The augmentation for orthogonal ones inserts edges so that each regular face is a rectangle. Similarly, we augment an orthoradial representation, so that each regular face fulfills Lemma 5.3. Assuming that the outer and inner face also fulfill the lemma and each vertex fulfills Lemma 5.1, then the representation is drawable if no monotone cycles are contained.

Let R be an orthoradial representation for a graph G. For the augmentation we insert a new border for the outer and inner face as in Section 4.2 explained. Figure 5.15 shows the new borders for an example graph marked in blue. When the new borders are inserted, the faces of R are checked for U-Shapes. If a U-Shape is found in a face f, then

the facial cycle of f contains a sequence of edges with the rotations $\text{rot}(t_1v, vt_2) = -1$, $\text{rot}(t_3u, ut_4) = 1, \text{rot}(t_5w, wx) = 1$ in this order. Between these three rotations, the sequence can contain an arbitrary number of edges with rotation 0. The augmentation for orthogonal representation inserts a new edge from v to a new vertex z on wx. However, in an orthoradial representation it can form a monotone cycle. Figure 5.15 shows on its left a U-Shape in an orthoradial drawing with the rotation sequence -1,1,1 found for the edges tv, vu, uw, wx. If now a new edge vz is inserted, whereby z lies on wx, the representation is no longer drawable, as it contains a new decreasing cycle $[vz, zx, xt, tv]$. It is also possible to create an increasing cycle with a new edge. However, in the case, where the new inserted edge is a vertical edge, no monotone cycle is produced by the augmentation. On the right of Figure 5.15 a vertical augmentation is shown. Because of monotone cycles, the augmentation differs from the orthogonal augmentation. In the following we assume the representations to be augmented to be free of monotone cycle before the augmentation.

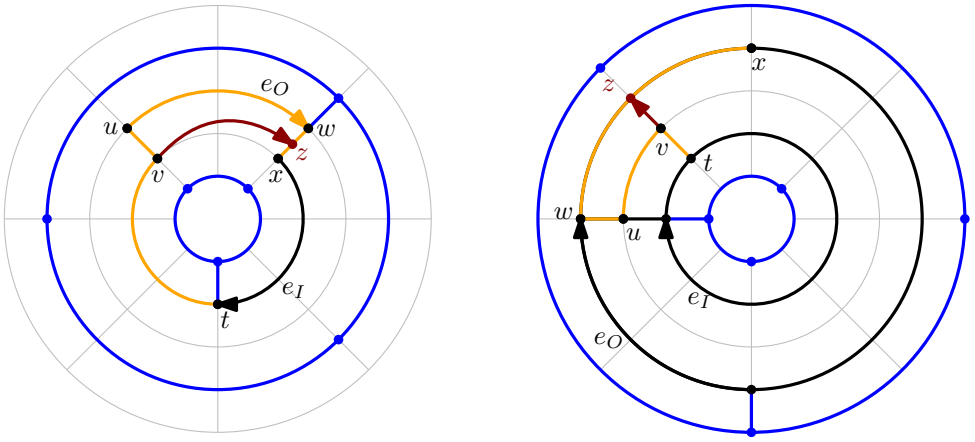


Figure 5.15: On the left: Augmentation that creates a decreasing cycle. On the right: vertical augmentation.

In an orthoradial representation, a U-Shape can be augmented like in the orthogonal case if the inserted edge is vertical. If a U-Shape requires a new horizontal edge, the augmentation performs following steps. Firstly, so called *candidate edges* are searched for the U-Shape that can substitute edge wx. These candidate edges are all contained in the face of the U-Shape and require the same orientation as the edge wx, so that the possible, inserted edges all have the same orientation. To find the candidate edges in the face of the U-Shape, the edges starting from edge uw, which has the second right turn of the U-Shape, are traversed until finding vertex v again. Figure 5.16 shows the possible candidates for an example U-Shape. Note that the drawing is only temporary because a drawing is only computed, when the augmentation is finished.

Secondly, after the candidate edges are found, these edges are tested if they produce a monotone cycle in the order they were found. An inserted edge vz can be tested with the algorithm for finding monotone cycles in the previous Section 5.3, where vz has the smallest label. If either of the tests is positive, the new edge has to be removed and the augmentation then tries to insert a new edge to the next candidate edge.

Barth et al. [BNRW17a] prove following properties for the candidate edges. The augmentation to the first candidate never produces a monotone cycle if the first candidate is a horizontal edge [BNRW17a][Lemma 21], which is equal to the inserted edge being vertical. Also, if the candidates are horizontal, then the first candidate never produces an increasing cycle and augmenting with the last candidate never produces a decreasing cycle [BNRW17a][Lemma 22, Lemma 24]. Consider the candidate edge w_ix_i producing a decreasing cycle and the next candidate producing an increasing one, then the augmentation

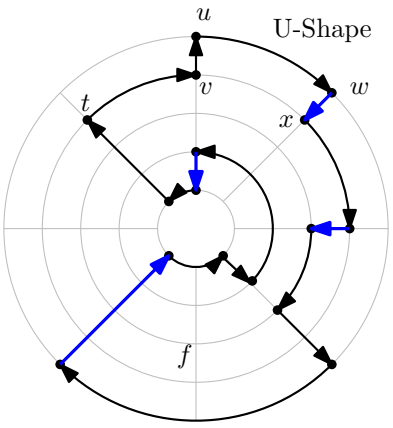


Figure 5.16: Finding the candidate edges (in blue) for a U-Shape $[t, v, u, w, x]$ in a face f.

inserts a new edge vx_i, which starts from the left bend at vertex v, and vx_i does not produce a decreasing or increasing cycle [BNRW17a][Lemma 25, Lemma 26]. Figure 5.17 shows this scenario, where the first candidate is wx and a new edge vx can be inserted. The first candidate would result in the new edge e_1, which produces a decreasing cycle. The second candidate would result in the new edge e_2, which produces an increasing cycle. Therefore, the edge $e_3 = vx$ can be inserted and it does not produce a monotone cycle.

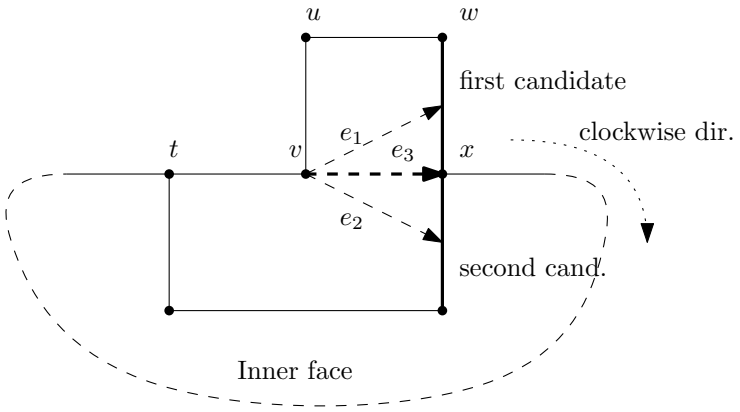


Figure 5.17: Augmentation and monotone cycles.

6. Implementation

This chapter describes the implementation of the drawing algorithm and gives an overview over noteworthy implementation details. In the end, the execution of the implementation is illustrated on an example input graph. The implementation is written in C++ and the library for working with graphs is the Open Graph Drawing Framework[1] (OGDF), which is described in the work of Chimani et al. [CGJ+13]. The Graph Markup Language (GML)[2] describes a graph in textual form. Graphs saved in a GML file can be read and converted with OGDF. Also, a graph in the program can be written as output as a GML file or as an image file in SVG format. The framework also provides graph algorithms, for example, for computing a planar embedding. However, OGDF has no classes for orthoradial graph embeddings or representations, as this is the first implementation of an orthoradial graph drawing algorithm. Therefore, attributes like rotation or orientation are not in the standard classes for the graph attributes. This implementation uses an extended version of the GraphAttributes class of OGDF, which is called RGraphAttributes. This subclass inherits all functionality for graphs that is already provided. We added the corresponding attributes for the outer and inner reference edge, the rotation and the orientation. Additionally, fields, which save the polar coordinates of the vertices, were added because of the concentric grid for orthoradial drawings.

The input of the implementation is a GML file. To avoid the handling of special cases we impose several constrictions on the input. The inserted graph is required to have a vertex degree between 2 and 4. Also, the graph has to contain the inverted edge vu for every edge uv. Although this implementation draws undirected graphs, the attributes of the edges, for example the rotations, need to be considered for each direction separately. Therefore, the input graph is expected to have each direction uv and of each edges vu. Another precondition for the input of the implementation are the reference edges. They have to always be given. The outer reference edge is marked with the red stroke color "#FF0000". The outer face then lies on the left side of the reference edge. The inner reference edge is marked with the green stroke color "#00FF00". The inner face then lies on the right side of the reference edge. These colors are also included in the RGraphAttributes class.

The input graph can be read with a given representation or a given embedding or without further information. The program then executes the yet needed steps for computing a

[1] http://www.ogdf.net

[2] https://www.fim.uni-passau.de/fileadmin/files/lehrstuhl/brandenburg/projekte/gml/gml-technical-report.pdf

drawing. However, like OGDF, GML does not support the rotation attribute for the edges or the order of the edges around a vertex. If a fixed representation is desired, it has to be encoded in the GML file or an extra file. We chose the first version and encoded the rotations between two consecutive edges in circular order around a vertex as the labels of the edges. Given two edges uv, vw that are consecutive in counter-clockwise direction around v, then the label of uv gives the rotation $\text{rot}(uv, vw)$. Figure 6.1 shows an example for interpreting the labels. If the GML file contains labels for all edges, they are interpreted as the rotations and therefore, if they are not formatted as $\{1, 0, -1, -2\}$ the graph is rejected. If the graph file does not contain labels or one label is missing, it is interpreted as a graph without a given representation.

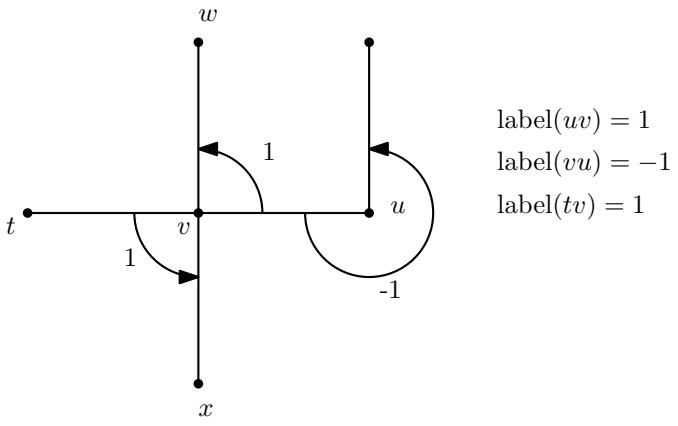


Figure 6.1: Graph drawing demonstrating saving the representation as labels.

Only the rotations between two consecutive edges in circular order around a vertex are saved within the labels. The rotations of non-consecutive adjacent edges, paths, cycles and faces are calculated when needed. The class RGraphAttributes provides methods to get the rotation between consecutive and non-consecutive edges. The rotation of a path, a cycle or a face can be calculated as in the formulas introduced in Section 4.1.1. The rotation between non-consecutive edges incident to a vertex can be calculated when considering the adjacency list of v. Let vertex v have an adjacency list $\text{adj}(v) = [u_1, u_2, u_3, u_4]$, then the rotation is $\text{rot}(u_1v, vu_4) = -1$. This is the case as all consecutive edges in circular order can only have rotation 1. Therefore, three 90° angles are represented and the angle between u_1v and vu_4 is $\text{angle}(u_1v, vu_4) = 270°$, which is a rotation of -1. Let vertex v have an adjacency list $\text{adj}(v) = [u_1, u_2, u_3]$ or $\text{adj}(v) = [u_1, u_2, u_3, u_4]$, then the rotation between u_1v and vu_3 is

$$\text{rot}(u_1v, vu_3) = -\big((\text{rot}(u_1v, vu_2) + \text{rot}(u_2v, vu_3))\%2\big).$$

This formula equals the cases

$$\text{rot}(u_1v, vu_3) = \begin{cases} 0 & \text{if } \text{rot}(u_1v, vu_2) = 1 \text{ and } \text{rot}(u_2v, vu_3) = 1 \\ -1 & \text{if } \text{rot}(u_1v, vu_2) = 1 \text{ and } \text{rot}(u_2v, vu_3) = 0 \\ -1 & \text{if } \text{rot}(u_1v, vu_2) = 0 \text{ and } \text{rot}(u_2v, vu_3) = 1 \end{cases}.$$

An embedding is also required for the representation, as it determines the two edges in circular order for a rotation. Also, the program can be started with a given embedding for a graph. The embedding is represented by the coordinates of the vertices in the Cartesian coordinate system. Let G be the graph shown in Figure 6.1. Then the adjacency list for v is $\text{adj}(v) = [t, w, u, x]$. When building a GML file and choosing the coordinates for the vertices it is recommended to adjust to the shape of the orthoradial grid. On the left of

Figure 6.2 the grid is shown. As an example, the orthoradial drawing on the right of the figure can have the embedding specified by the coordinates, as shown in the middle of the figure, if the rotations are chosen respectively.

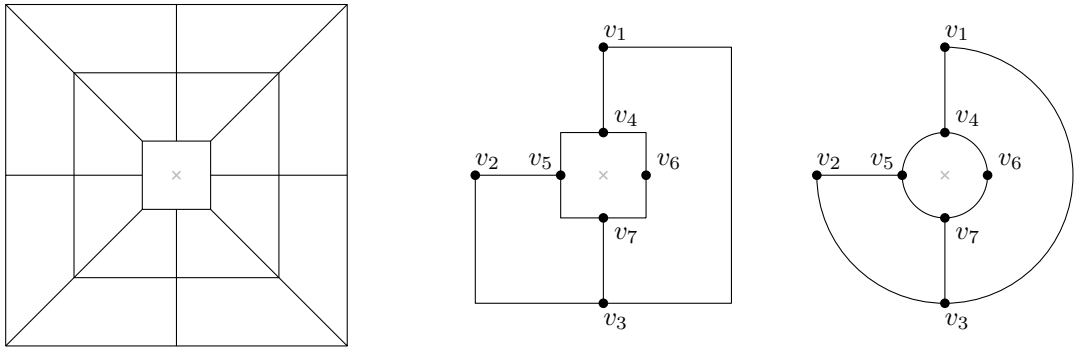


Figure 6.2: On the left, the grid for the embedding. In the middle the embedding of the orthoradial drawing on the right.

Extracting the adjacency lists from the coordinates repeats for each vertex v a comparison of the x and y coordinates of v to each adjacent vertex. As we have a 4-planar graph, we differentiate between four positions relative to a vertex v, south, north, west and east of v. For example, in Figure 6.1 is vertex x south of v and v south of w. If an edge contains a bend, for example the edge v_4v_6 in Figure 6.2, then the position of the bend before the vertex is decisive for the ordering. Take the edge v_4v_6 of the figure, then for the adjacency list of v_4 the vertex v_6 is to v_4's east. The vertex v_6 has v_4 to its north. With these relative positions, the order around a vertex can be easily determined.

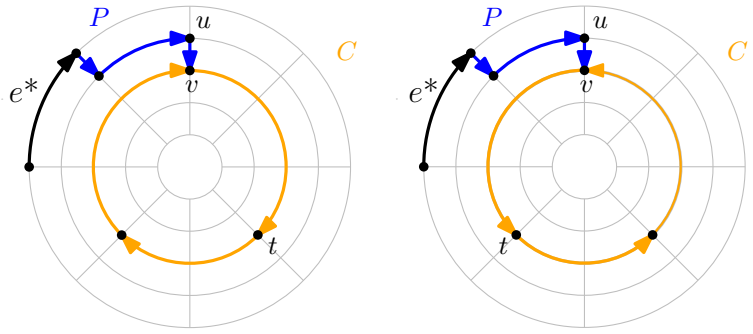
When a graph is read into the implementation without an embedding, OGDF inserts a default value for the coordinates of the vertices, which is (0,0). In order to check if an embedding is given and specified by the input it is checked if more than one vertex has the coordinates (0,0). Note that the coordinates in the input graph are not the final coordinates for the output graph drawing but are only for the encoding of the embedding in a GML file.

The program can be started with only a graph without a given embedding or representation. The implementation then computes a planar embedding with the functionality of OGDF. The program aborts if it is not possible to embed the graph in a planar way. A representation is computed afterwards with a network flow algorithm as explained in Section 5.1.

The source code of the implementation is split into multiple directories so that the functionalities are grouped. The partitioning resulted from implementing the features piece by piece. The basic file operations, like reading a graph from a file, and the computation of rotations or orientations are collected in the directory "readWriteGraph". It contains methods for reading the input graph, converting the labels, reading the reference edges, writing the graph to a file and more. The RGraphAttributes class can be found in a directory named "types". Debug methods, geometrical computations and search algorithms such as the DFS or BFS are contained in the directory "util". The algorithms using a network flow, which are the rotation and coordinate computations, can be found in the directory "networkFlow". Both computations use the minimum cost flow algorithm provided by OGDF. Furthermore, there exists a directory "augmentation" for the augmentation procedure and a directory "cycleMethods" for the monotone cycle tests.

6.1 Determining the Direction of a Cycle

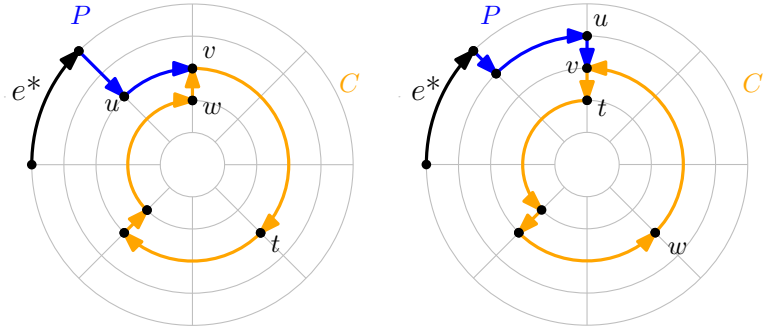
The direction of a cycle can be distinguished because the outer reference edge is always directed clockwise as a precondition and has the inner face to it's right. If the outer reference edge is contained in the cycle, the cycle direction is clockwise. If the inversed edge of the outer reference edge is part of the cycle, then the cycle direction is counter-clockwise. If the cycle does not contain the outer reference edge nor its inversed edge, a path P from the outer reference edge to the cycle has to be determined. Let e^* be the outer reference edge and let s be the target vertex of e^*. Let C be an essential cycle in the graph. Then, there exists an elementary, arbitrary path P from s to a vertex v on C. The path P is elementary if it intersects C only at its endpoints. The rotation between the last edge uv of the path and the first directed edge on the cycle vt indicates the cycle direction. If the rotation $\text{rot}(uv, vt)$ is 1, then the cycle is directed in counter-clockwise direction. If the rotation is -1, then the cycle is directed in clockwise direction. The Figure 6.3 shows these two scenarios. Note that only the cycle on the left is desired, as it is directed clockwise.



$\text{rot}(uv, vt) = -1$ $\text{rot}(uv, vt) = 1$

Figure 6.3: On the left: cycle C that is clockwise directed. On the right: inverted cycle $\overline{C}$.

If the rotation between the last edge uv of path P and the first edge of the cycle vt is $\text{rot}(uv, vt) = 0$, then the cycle can be directed clockwise and also counter-clockwise. The Figure 6.4 shows these two scenarios, where the rotation is 0. Then the rotation between the last edge of the path uv and the last edge vw of the cycle, when walking along the cycle starting from v, is decisive for the cycle direction. If the rotation is $\text{rot}(uv, vw) = 1$, then the cycle is directed clockwise and if the rotation is $\text{rot}(uv, vw) = -1$, then the cycle is directed counter-clockwise.



$\text{rot}(uv, vt) = 0$ $\text{rot}(uv, vt) = 0$
$\text{rot}(uv, vw) = 1$ $\text{rot}(uv, vw) = -1$

Figure 6.4: On the left: cycle C that is clockwise directed. On the right: inverted cycle $\overline{C}$.

6.2 Finding monotone cycles

The implementation needs the test for monotone cycles for checking if a representation is drawable and also for the augmentation. Therefore, finding the monotone cycles is a central feature of the implementation. The process for finding the monotone cycles is split into two methods - testing a specific edge for an increasing or decreasing cycle. Independent of these tests, another method is implemented for checking if a cycle is essential and the labeling is also separately implemented. These functionalities are collected in the directory "cycleMethods".

The implementation of the decreasing cycles test is structured like the explained procedure in Section 5.2. Firstly, the outermost cycle for an edge e is searched so that e has the smallest label on the cycle. This is implemented with a left-first-DFS, which chooses the next unvisited edge with a positive search label. If a cycle was found, it is tested whether it is essential and then, whether it is decreasing.

For the augmentation both tests for monotone cycles are necessary. When a U-Shape in the orthoradial representation is searched for and needs to be augmented, like explained in Section 5.5, we search for candidate edges. Then, a new edge is inserted from the left bend of the U-Shape to one of the candidate edges so that no new monotone cycle exists afterwards. When a candidate is tested, a temporary edge to the candidate from the left bend of the U-Shape is inserted. This new edge is then tested whether it is the edge with the smallest label on a decreasing cycle. If the tested candidate is not the first and is not decreasing, it is tested for an increasing cycle. Barth et al. [BNRW17a] prove that there exists a horizontal path between two candidates to the left bend if the first of the two produces a decreasing cycle and the other an increasing one [BNRW17a][Lemma 25]. Therefore, the test for an increasing cycle, with the precondition that the previous candidate produces a decreasing cycle, searches for a horizontal path, which consecutive edges all have rotation 0. Figure 5.17 shows this scenario as a sketch.

6.3 Augmentation

The augmentation of an orthoradial representation is implemented as explained in Section 5.5. All faces of the graph are iterated through and checked for U-Shapes. If a U-Shape is found, it is augmented with a new edge. Firstly, all candidate edges with the needed orientation are searched. Secondly, they are tested in the order they were found whether a monotone cycle exists after inserting a temporary edge to the candidate. If a monotone-cycle was found, the temporary edge is removed and the next candidate is tested. Otherwise, the temporary edge is accepted and the U-Shape is augmented. If the orientation of the candidate edges is vertical, the process of searching and testing all candidate edges can be skipped, as the first vertical candidate is proven to never produce a monotone cycle. After a U-Shape was augmented, the face, which contained the U-Shape, is again checked for one. The new face is also checked for U-Shapes afterwards.

In the augmentation step new borders for the outer and inner face are inserted in the given representation. Let R be an orthoradial graph representation. The new borders have to preserve the rotations and orientations of the original graph and also have correct rotations and orientations themselves. The extension of R with new borders for the outer and inner face is explained on the outer face. Let e^* be the outer reference edge. A new edge e_1 is inserted, so that the rotation $\text{rot}(e^*, e_1) = -1$. Let e be the next edge in clockwise direction after e^*, then the rotation between e and e_1 is

$$rot(e, e_1) = \begin{cases} 0 & \text{if } \text{rot}(e^*, e) = 1 \\ 1 & \text{if } \text{rot}(e^*, e) = 0 \end{cases}.$$

If $\text{rot}(e^*, e) = -1$ applied, then the new edge can not be inserted so that $\text{rot}(e^*, e_1) = -1$. We then search starting from e for the next edge pointing outwards that has no neighbor also pointing outwards and insert the new edge e_1 to this edge with rotation 0. Figure 6.5 shows two of these scenarios for the insertion of e_1. When e_1 was inserted, three or two more edges are needed to form a cycle. The implementation inserts three edges e_2, e_3, e_4 so that $\text{rot}(e_1, e_2) = 1$ and $\text{rot}(e_4, e_1) = 1$ and all other rotations between e_2, e_3 and e_4 are 0.

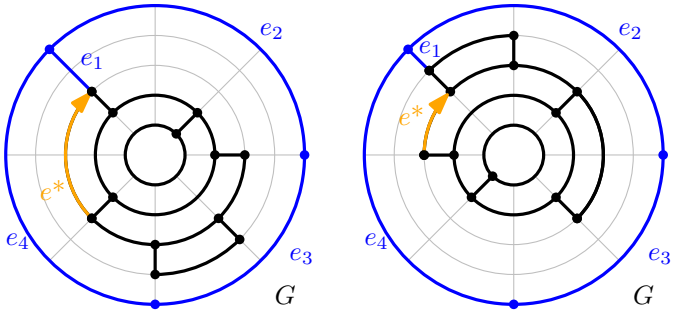


Figure 6.5: Extending the outer face.

The code for the augmentation is collected in a directory called 'augmentation', which then contains multiple C++ files. For example, the file "extendGraph" contains the methods for inserting the new borders of the outer and inner face .

6.4 Drawing Arcs

Orthoradial graph drawings differ from orthogonal ones in the need of circular arcs. Only the vertical edges of an orthoradial drawing are straight lined strokes. The last step of the implementation constructs a orthoradial drawing by bending the horizontal edges as circular arcs. The OGDF library provides only straight lined strokes, so the implementation has to insert bends on the horizontal edges until the edge resembles a circular arc. The edges of the graph are traversed through with a right first DFS starting from the outer reference edge, which is known to be horizontal and directed in clockwise direction. A horizontal edge then gets bent. The bend points are computed by vertex rotation. The code for this procedure can be found in the file "convertToRadial" in the directory "readWriteGraph".

The angle that is covered by an edge e is needed to insert the bends. Let e be a horizontal edge with start vertex s and target vertex t. The function $\theta(v)$ of a vertex v gives the angle of v in the polar coordinate system. The angle that is between s and t is $\gamma(s,t) = \theta(t) - \theta(s)$. However, this has not to be the angle that edge st covers, as it depends on the orientation of st in the representation and the positions of the vertices. Note that $\gamma(s,t) = 0$ implies that edge st is vertical and therefore, is excluded in the following. We can differentiate between two cases, in which edge st has either orientation 0 or 2. Therefore, edge st is either clockwise or counter-clockwise directed. Figure 6.6 shows the possible orientations in two drawings. Depending on the angle between s or t and the orientation of edge st, the computation of the angle $angle(st)$ of edge st differs. If $\gamma(s,t) = \theta(t) - \theta(s) > 0$, then the angle covered by edge st is

$$\text{angle}(e) = \begin{cases} 360° - \gamma(s,t) & \text{if orientation of } e \text{ is } 0 \\ -\gamma(s,t) & \text{if orientation of } e \text{ is } 2 \end{cases}. \tag{6.1}$$

If $\gamma(s,t) = \theta(t) - \theta(s) < 0$, then the angle covered by edge st is

$$\text{angle}(e) = \begin{cases} -\gamma(s,t) & \text{if orientation of } e \text{ is } 0 \\ 360° + \gamma(s,t) & \text{if orientation of } e \text{ is } 2 \end{cases}. \tag{6.2}$$

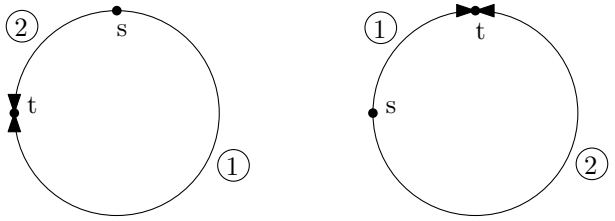


Figure 6.6: Finding the angle covert by an edge st with $\gamma(s,t) > 0$ on the left and $\gamma(s,t) < 0$ on the right. ① = orientation 0. ② = orientation 2.

In the implementation for a concentric circle that is further away from the center of the drawing more bends are inserted, so that it appears as circular. The conversion is shown in Figure 6.7 with one or two bends per edge. Depending on the angle that a horizontal edge covers and the radius from the drawing's center, the number of bends for a horizontal edge should at least be

$$steps(e) = \begin{cases} radius/10 & \text{if angle}(e) < 90° \text{ and angle}(e) > -90° \\ 2 \cdot radius/10 & \text{if angle}(e) > 90° \text{ and angle}(e) < 180° \\ & \text{or angle}(e) < -90° \text{ and angle}(e) > -180° \\ 3 \cdot radius/10 & \text{if angle}(e) > 180° \text{ or angle}(e) < -180° \end{cases}. \tag{6.3}$$

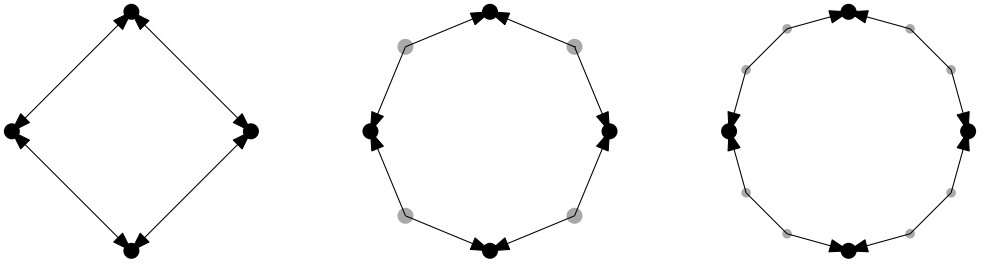


Figure 6.7: Converting an edge line to an arc.

Figure 6.8 shows a result output of the implementation, painted with the yEd-Editor[3]. The number of bends on the horizontal edges is computed with the formula and the horizontal edges form accurate arcs. The implementation assigns a fixed radius to each concentric circle, i.e. the radius increases in steps of 100 units from the center outwards. Therefore, the radius increases linearly. As the number of bends $steps(e)$ for an edge e are dependent on its radius, the number of bends also scales linearly with the radius. The distance between two bends is therefore in every concentric circle the same and the horizontal edges appear to be arcs.

[3]https://www.yworks.com/products/yed

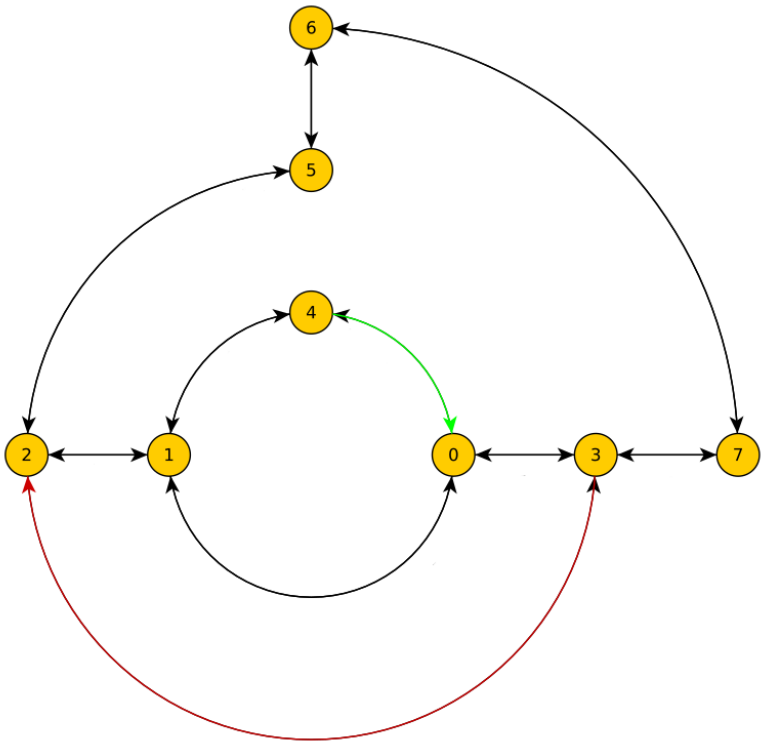


Figure 6.8: Result of bending the horizontal edges.

6.5 Example Graph

In this section the execution of the implementation is shown for a graph G. The graph contains eight vertices $v_0, ..., v_7$ and following list of edges that is

$$[v_0v_1, v_0v_4, v_0v_3, v_1v_2, v_1v_4, v_2v_3, v_2v_5, v_3v_7, v_5v_6, v_6v_7].$$

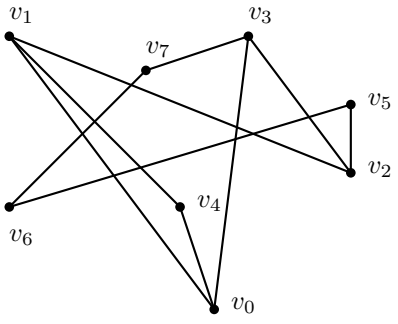


Figure 6.9: Example Graph G.

Figure 6.9 shows a drawing of G. As the input graph needs the inverted edge uv for every edge vu, the list above is in fact not accepted by the implementation. The given list of the edges is shortened for better display. In the following all inversed edges are considered contained. The first step of the implementation is the search for the reference edges, which checks the stroke color for each edge. Consider the edge v_4v_0 the inner reference edge and the edge v_3v_2 the outer reference edge. The implementation then searches for the complementary edge pair. Afterwards, the implementation checks whether

coordinates for the vertices are given and uses them as the embedding of the graph. For this example, consider no embedding or representation given in the input graph. If an embedding is given, the implementation checks if a representation is also given. If not, then an orthoradial representation is calculated with a network flow. For the input graph a new planar embedding is computed with OGDF. Figure 6.10 shows the resulting order for the edges on its left and a drawing for the graph at this step on the right, although no actual drawing exists yet.

Vertex	Adjacency list
v_0	$[v_4, v_1, v_3]$
v_1	$[v_2, v_0, v_4]$
v_2	$[v_1, v_5, v_3]$
v_3	$[v_2, v_7, v_0]$
v_4	$[v_1, v_0]$
v_5	$[v_2, v_6]$
v_6	$[v_5, v_7]$
v_7	$[v_6, v_3]$

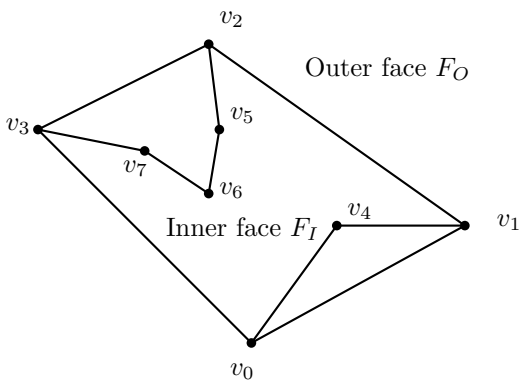


Figure 6.10: Graph after the embedding has been computed.

Based on this embedding the rotations are calculated with a network flow algorithm. Figure 6.11 shows on its left a drawing of the graph representation with the computed rotations. Note that a new vertex v_8 was inserted for the face rotation of face $[v_0v_4, v_4v_1, v_1v_8, v_8v_0]$, acting as a bend, so that it has a face rotation of 4. Then, the orientations of the edges are extracted from the rotations based on the fact that the outer reference edge has orientation 0. Following formula is provided in the implementation by the class RGraphAttributes. Let the edge uv have the orientation orientation(uv), then the orientation for an edge vw is

$$\text{orientation}(vw) = (\text{orientation}(uv) + \text{rot}(uv, vw) + 4)\%4. \tag{6.4}$$

In the next step, the implementation tests the representation for a decreasing cycle. As in Section 5.3 explained, each edge of the graph is tested whether it is contained in a decreasing cycle. Let e be an edge of the graph. The implementation searches for the outermost cycle containing e so that e has the smallest label on the cycle. The outermost cycle has all other cycles containing e enclosed in its interior. Then, the found cycle is tested whether it is decreasing. In the example, the first hit is the edge v_0v_3, which lies on its outermost decreasing cycle $C = [v_0v_3, v_3v_2, v_2v_1, v_1v_8, v_8v_0]$. As the outer reference edge v_3v_2 is contained in the cycle we have the labels: label$(v_0v_3) = 0$, label$(v_3v_2) = 0$, label$(v_2v_1) = 0$, label$(v_1v_8) = 0$, label$(v_8v_0) = 1$. The decreasing cycle is then repaired with an inserted spiral as explained in Section 5.4. On its right Figure 6.11 shows the representation after the insertion of the spiral, which is marked in blue. The new cycle that is the corresponding one to C is $[v_0v_{12}, v_{12}v_{11}, v_{11}v_{10}, v_{10}v_9, v_9v_3, v_3v_2, v_2v_1, v_1v_8, v_8v_0]$ and has a negative label on $v_{12}v_{11}$. The graph contains no other decreasing cycle.

The next step is the augmentation of the graph. Firstly, the outer and inner face get new borders. Then all faces are checked for U-Shapes. The new borders are marked in blue. Figure 6.12 shows a temporary result after the explained steps of augmentation in the following. The other U-Shapes and their augmentation are shown in Figure 6.13, which is also the final output of the program. The bend represented by the new vertex v_8 is colored in bright red. The inserted spiral at edge v_3v_2 is marked in brown. The inserted edges and vertices by the augmentation are orange. In the representation seven U-Shapes were found.

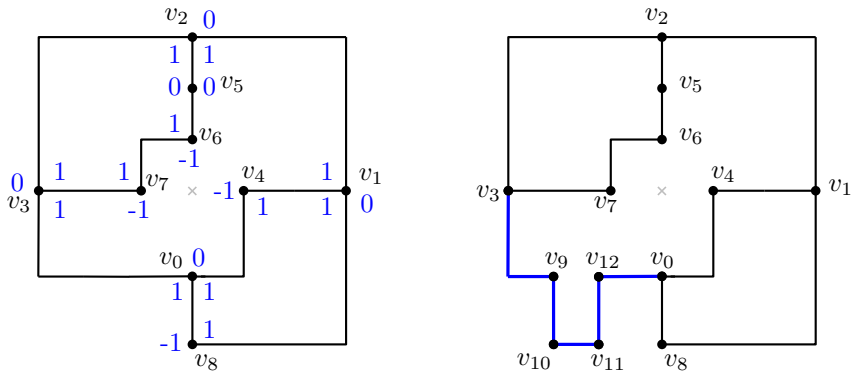


Figure 6.11: Result of rotations on the left. Result of fixing the decreasing cycle on edge v_0v_3 on the right.

Note that the resulting graph may differ based on the order in which the U-Shapes are found. The first U-Shape with rotation sequence -1,1,1 was formed by the edges v_7v_6, v_5v_2, v_2v_1. Note that the edge v_6v_5 was skipped as it has rotation 0. The candidate edges for this U-Shape are v_1v_4, v_0v_{16} and $v_{10}v_9$. The test with the first candidate v_1v_4 resulted in no new monotone cycle and therefore, the edge v_6v_{19} was inserted. The next U-Shape found is $v_0v_{12}, v_{12}v_{11}, v_{11}, v_{11}v_{10}$. Here, the first candidate $v_{10}v_9$ produces a decreasing cycle, as well as the second candidate v_3v_7 and third candidate $v_{19}v_4$. The augmentation with the last candidate v_0v_{16}, which is part of the extension of the inner face's border, produces an increasing cycle. Because of the switch from decreasing to increasing cycle, a new edge $v_{12}v_4$ can be inserted to the target vertex v_4 of the last candidate $v_{19}v_4$, which produces a decreasing cycle. This procedure is known to not insert a monotone cycle.

In the last step of the implementation, the length of the edges and the coordinates of the vertices are computed. The horizontal edges are bent to be concentric arcs. The fat marked lines are the edges of the original graph with the spiral.

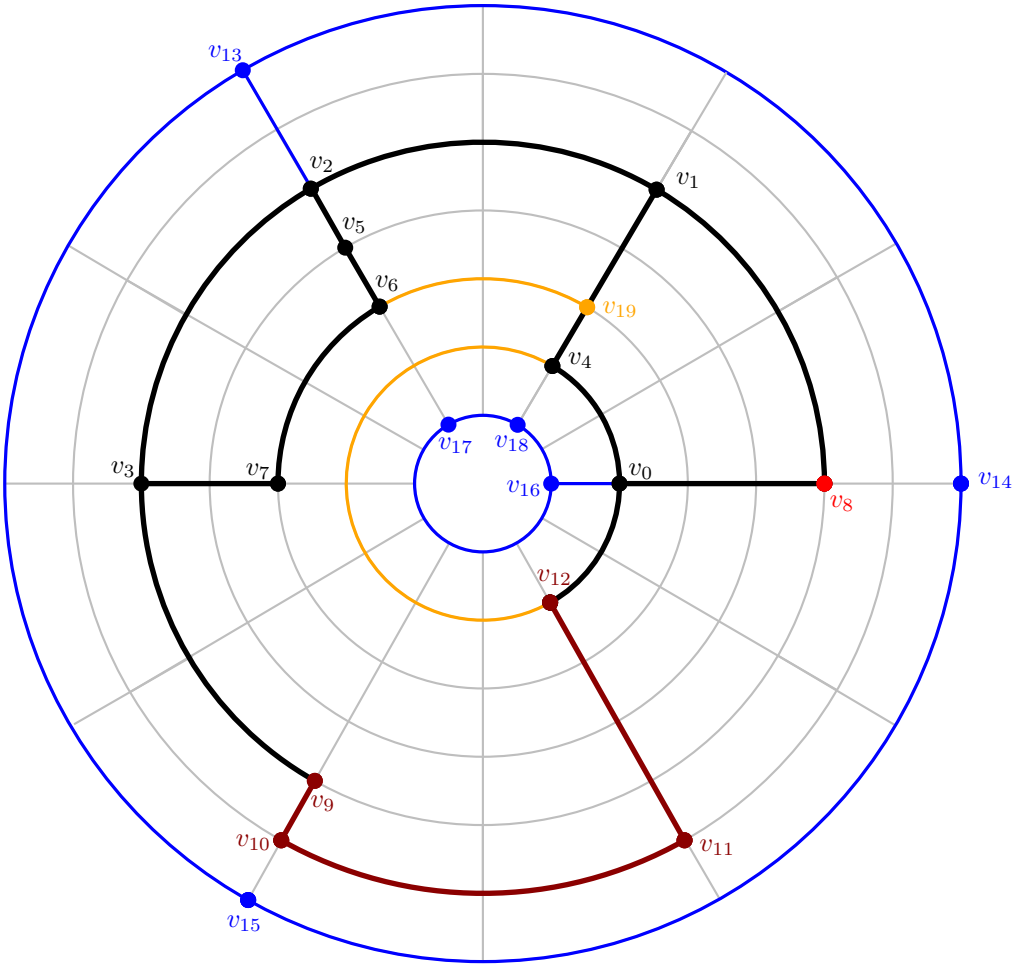


Figure 6.12: First two augmentation steps. In blue, the new borders for the outer and inner face. In orange, the augmentations. In bright red, inserted bend. In brown, the inserted spiral for the decreasing cycle.

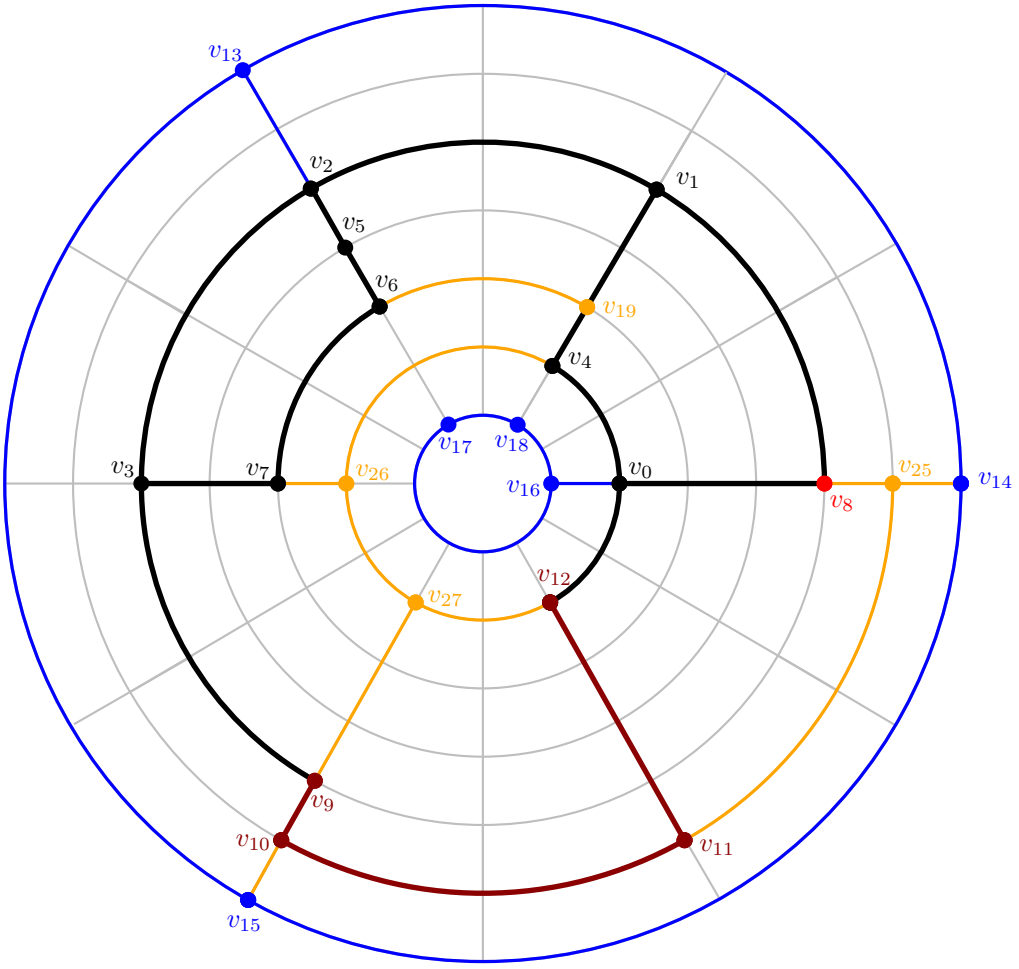


Figure 6.13: Result orthoradial drawing for the example graph in Figure 6.9. In blue, the new borders for the outer and inner face. In orange, the augmentations. In bright red, inserted bend. In brown, the inserted spiral for the decreasing cycle.

7. Conclusion

This work explains the theory of orthogonal and orthoradial graph drawing. In the orthoradial drawing theory the remaining proofs, that were not yet included in other works, are explained. The dependence between vertex rotation and vertex degree is described in a new lemma and a proof is provided. Additionally, missing details of the original papers for the face rotation of orthoradial graph drawings are shown in detail. For the proof the splitting of polygons in an orthoradial grid and the computation of the internal angle sum of these polygons are defined. The rotation and coordinates computations on the orthoradial case are also included in detail. To simplify the descriptions of the algorithm, a new orientation attribute for orthoradial representations is introduced. Also, exact details for finding the cycle direction and monotone cycles are summarized. The effects of monotone cycles on the augmentation are immense, as each augmentation now needs several candidate edges and needs to check for new inserted monotone cycles after each step. The implementation of the augmentation confirmed the behavior of the insertion between monotone cycles, when a change from decreasing to increasing cycle was found.

The implementation of the orthoradial graph drawing algorithm is the first prototype. It is functional and can compute an orthoradial drawing for a connected, 4-planar graph. However, the process of implementation was problematic. Small errors in the rotations led to program failures or infinite loops. Additionally, the debugging of a program part requested a self built specific graph and the debugging of all parts before the specific problematic part that was intended to be debugged. This is the case because the graph changes or is extended in each step and a drawing can only be illustrated after the execution has finished successfully. Therefore, rotations may change before a specific point in the execution and have to be recorded manually. Often, an error in a later program step is a result of incorrectly assigned rotations or orientations in a previous step. The input of the program is restricted to 4-planar connected graphs that have at least a vertex degree of 2 at each vertex. Due to this restriction, special cases, which are too time consuming to handle separately, are avoided.

As the bend minimization is still an unsolved optimization problem, it may be interesting to research if an exploitation of the characteristics of orthoradial drawings are useful for the bend minimization problem. Some graphs may need less bends in an orthoradial representation than in an orthogonal one. Another topic for future work may include a comparison of time complexity of common bend minimization algorithms applied on orthoradial representations or of the graph drawing algorithm in general.

Bibliography

[BCG+13] Christoph Buchheim, Markus Chimani, Carsten Gutwenger, Michael Jünger, and Petra Mutzel. Crossings and planarization. In Roberto Tamassia, editor, *Handbook of Graph Drawing and Visualization*, pages 43–86. CRC Press, 2013.

[BETT98] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G Tollis. *Graph drawing: algorithms for the visualization of graphs*. Prentice Hall PTR, 1998.

[BKRW11] Thomas Bläsius, Marcus Krug, Ignaz Rutter, and Dorothea Wagner. Orthogonal graph drawing with flexibility constraints. In Ulrik Brandes and Sabine Cornelsen, editors, *Graph Drawing*, pages 92–104. Springer, 2011.

[BNRW17a] Lukas Barth, Benjamin Niedermann, Ignaz Rutter, and Matthias Wolf. Towards a topology-shape-metrics framework for ortho-radial drawings. *arXiv preprint arXiv:1703.06040*, 2017.

[BNRW17b] Lukas Barth, Benjamin Niedermann, Ignaz Rutter, and Matthias Wolf. Towards a topology-shape-metrics framework for ortho-radial drawings. In *33rd International Symposium on Computational Geometry, SoCG 2017, July 4-7, 2017, Brisbane, Australia*, pages 14:1–14:16, 2017.

[CGJ+13] Markus Chimani, Carsten Gutwenger, Michael Jünger, Gunnar W Klau, Karsten Klein, and Petra Mutzel. The Open Graph Drawing Framework (OGDF). In Roberto Tamassia, editor, *Handbook of Graph Drawing and Visualization*, pages 543–569. CRC Press, 2013.

[CK11] Sabine Cornelsen and Andreas Karrenbauer. Accelerated bend minimization. In Marc van Kreveld and Bettina Speckmann, editors, *Graph Drawing*, pages 111–122. Springer, 2011. International Symposium on Graph Drawing.

[DG13] Christian A Duncan and Michael T Goodrich. Planar orthogonal and polyline drawing algorithms. In Roberto Tamassia, editor, *Handbook of Graph Drawing and Visualization*, pages 223–246. CRC Press, 2013.

[EFK01] Markus Eiglsperger, Sándor P Fekete, and Gunnar W Klau. Orthogonal graph drawing. In Michael Kaufmann and Dorothea Wagner, editors, *Drawing Graphs: Methods and Models*, pages 121–171. Springer, 2001.

[GT97] Ashim Garg and Roberto Tamassia. A new minimum cost flow algorithm with applications to graph drawing. In Stephen North, editor, *Graph Drawing*, pages 201–216. Springer, 1997. Symposium on Graph Drawing.

[HHT09] Mahdieh Hasheminezhad, S Mehdi Hashemi, and Maryam Tahmasbi. Ortho-radial drawings of graphs. *Australasian J. Combinatorics*, 44:171–182, 2009.

[NRW13] Benjamin Niedermann, Ignaz Rutter, and Matthias Wolf. Efficient algorithms for ortho-radial graph drawing. Manuscript, 2013.

[Pat13] Maurizio Patrignani. Planarity testing and embedding. In Roberto Tamassia, editor, *Handbook of Graph Drawing and Visualization*, pages 1–42. CRC Press, 2013.

[Tam87] Roberto Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM Journal on Computing*, 16(3):421–444, 1987.

[Tam13] Roberto Tamassia, editor. *Handbook of graph drawing and visualization*. CRC press, 2013.

[TDBB88] Roberto Tamassia, Giuseppe Di Battista, and Carlo Batini. Automatic graph drawing and readability of diagrams. *IEEE Transactions on Systems, Man, and Cybernetics*, 18(1):61–79, 1988.

[Wei01] René Weiskircher. Drawing planar graphs. In Michael Kaufmann and Dorothea Wagner, editors, *Drawing Graphs: Methods and Models*, pages 23–45. Springer, 2001.