A Generic Framework for the Topology-Shape-Metrics Based Layout

Paul Klose

Master Thesis
October 2012

Christian-Albrechts-Universität zu Kiel
Real-Time and Embedded Systems Group
Prof. Dr. Reinhard von Hanxleden

Advised by: Dipl-Inf. Miro Spönemann

Abstract

Modeling is an important part of software engineering, but it is also used in many other areas of application. In that context the arrangement of diagram elements by hand is not efficient. Thus, layout algorithms are used to create or rearrange the diagram elements automatically in order to free users from this. However, different types of diagrams require different types of layout algorithms.

Planarity and orthogonality are well-known drawing conventions for many domains such as UML class diagrams, circuit schemata or entity-relationship models. One approach to arrange such diagrams is considered by Roberto Tamassia and is called Topology-Shape-Metrics approach, which minimizes edge crossings and generates compact orthogonal grid drawings. This basic approach works with three phases: the planarization, the orthogonalization, and the compaction. The implementation of these phases are considered in this thesis in detail with respect to a generic, extensible architecture, such that every phase can be exchanged with different alternatives. This leads to a considerable amount of flexibility and expandability. Special handling of high-degree nodes has been implemented based on this architecture. Moreover, approach and implementation proposals for interactive planarization and for handling edge labels, hypergraphs and port constraints are presented.

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Contents

List of Figures

List of Tables

Abbreviations

TSM	Topology-Shape-Metrics
KIEL	Kiel Integrated Environment for Layout
KIELER	Kiel Integrated Environment for Layout Eclipse Rich Client
KIML	KIELER Infrastructure for Meta Layout
KLay	KIELER Layout Algorithms
OGDF	Open Graph Drawing Framework
UML	Unified Modeling Language
MCF	Minimum Cost Flow
SSP	Successive Shortest Path
BFS	Breadth First Search
DFS	Depth First Search
DFI	DFS Index

Chapter 1

Introduction

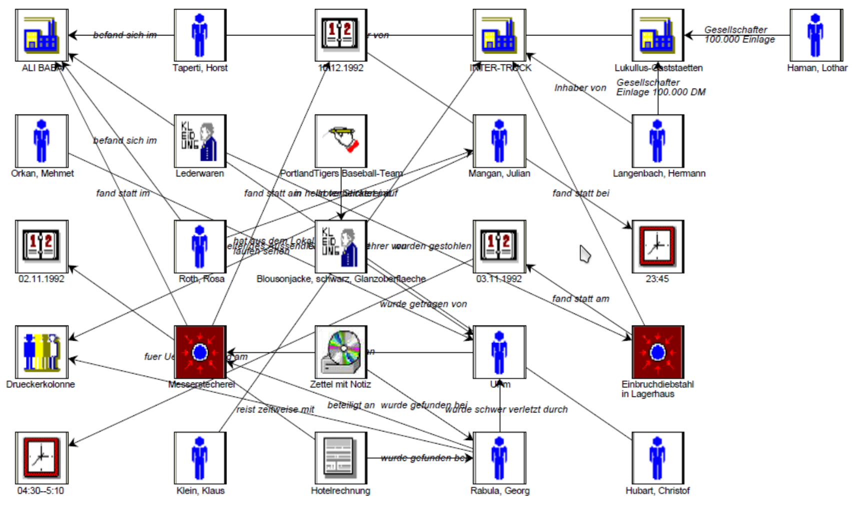
"The usefulness of a drawing of a graph depends on its readability that is, the capability of conveying the meaning of the graph quickly and clearly." Roberto Tamassia [Di +99].

Computer systems as well as the hard- and software development usually consist of large complex conceptual structures. In many cases, people that hold different experiences and knowledge backgrounds are involved in these processes. Thus, the design of such structures needs to be comprehensible to let all participants work in the same, desired direction.

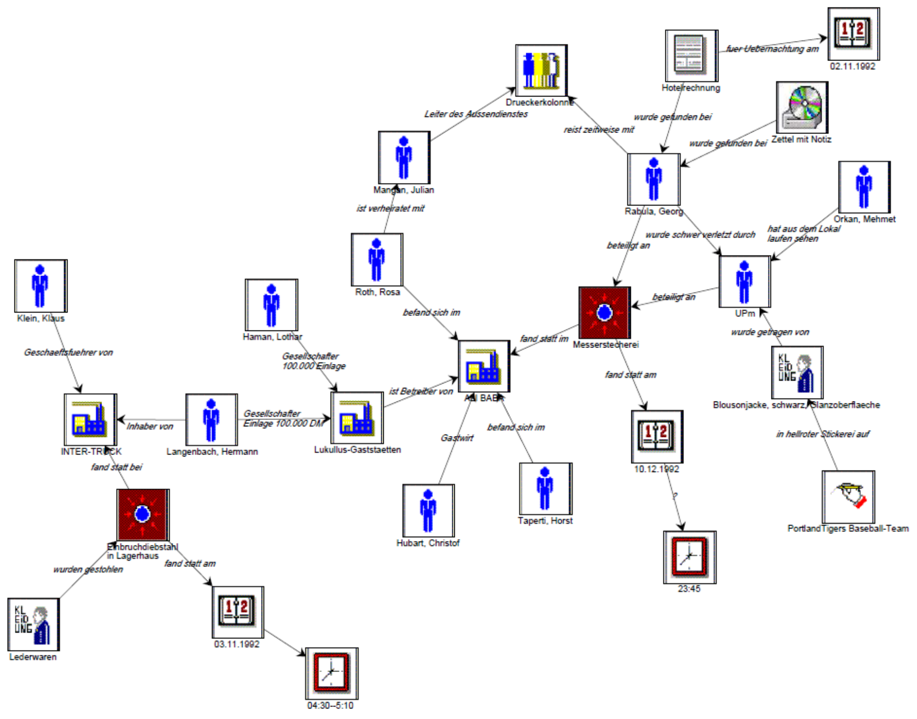
Frequently, graphical representations are used to model relations in such areas of application. Especially graphs are used to represent relations between different objects in diagrams. Examples of such representations are circuit schematics, state diagrams, and database dependencies for application developers.

A readable drawing is important. Figure 1.1 shows the same diagram with different layouts. The first one in Figure 1.1a is hard to read because of the many edge crossings, though it is in compact shape. A force-based layout algorithm is applied to the drawing with the result that no crossings remain (see Figure 1.1b). The relations between the elements of the drawing are much clearer, but they need more drawing space. Usually these diagrams are two-dimensional, and they can become large quite fast. The manual maintenance and manual expansion of such diagrams is expensive, hence tools that are able to do the layout automatically are desired.

Different types of drawings require different types of layout algorithms. Each layout algorithm provides different aesthetics criteria. There are general criteria such as edge crossings and total edge lengths, and there are criteria that depend on the user preference and the use case. In this thesis, the considered layout algorithm provides orthogonal graph drawings. Orthogonality is ensured if all the segments of the edges in a graph are drawn horizontally or vertically. These types of drawings are widely used in circuit schematics and in software diagrams



a. Result of a database query.



b. Same drawing with force-based layout.

Figure 1.1. Different layouts for the same diagram [Mut05].

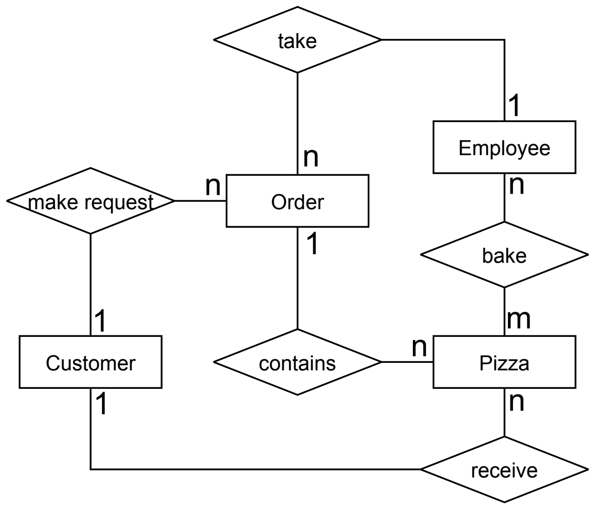


Figure 1.2. Entity-relationship diagram.

like the Unified Modeling Language (UML) class diagrams or entity-relationship models. Figure 1.2 shows an orthogonal drawing of an entity-relationship model. Every edge segment is bounded by a bend or an endpoint.

Orthogonal drawings can be realized with a basic layout algorithm, namely the Topology-Shape-Metrics (TSM) approach. Roberto Tamassia introduced this algorithm in 1987 [Tam87]. Its task is to compute compact orthogonal grid drawings with few edge crossings and a few number of edge bends.

To do that, firstly, the algorithm calculates a planar embedding of an input graph. If no planar embedding exists dummy nodes are introduced to obtain one. Then, the algorithm computes a bend-minimal and compact orthogonal grid drawing with respect to the input embedding. If it is allowed to change the embedding of the input graph, the problem becomes NP-hard [GT02]. The approach is divided into three parts, namely planarization, orthogonalization, and compaction. This allows a good exchangeability and expandability.

The completion of the implementation of that algorithm in a modeling tool as well as the consideration of extensions are the main contributions of this thesis.

1.1 Related Work

Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) is an academical research project that improves the graphical model-based design of complex systems with automatic layout [Fuh11]. Miro Spönemann et al. created a framework for the integration of different layout algorithms to support automatic layout for the diagrams modeled in KIELER [SFH09; Fuh+10].

Claußen, Döhring, and Kutschmar worked at the implementation of the TSM algorithm in 2010 [Cla10; Döh10; Kut10]. They realized the planarization part of the algorithm including two different planarity testing techniques. Moreover, Claußen implemented most parts of the orthogonalization phase of the TSM approach. In that context, the authors implemented various shortest path finding algorithms and the creation of flow networks together with a method to solve the minimum cost flow problem. Thus, they created some important foundations for this thesis.

Other related works are documents and workgroups that consider the TSM approach. Basics of orthogonal drawing and the TSM algorithm can be found in various publications of Tamassia et al. [TDBB88; Di +94; Di +99]. Besides these papers the institutes Technische Universität Dortmund, the Friedrich-Schiller-University Jena, the University of Cologne, the University of Sydney, and oreas GmbH developed and still support the Open Graph Drawing Framework (OGDF)[1], which provides several layout algorithms. In particular, the TSM algorithm as well as a lot of extensions and variations of that are part of its library. Many ideas of their approaches were used in this thesis [Chi+07; GM04; JLM98; Ker07; KM98; MM96]. The OGDF provides a wide range of planarization algorithms. Their algorithms are suitable and efficient in generating and arrangement layouts. Thus, the following question comes up: Why implement an own TSM algorithm? Even if the framework is open, the algorithms of OGDF are written in C++, so that their source-code could not be integrated in the KIELER Layout Algorithms (KLay) which is a Java-based project. The goal is to research new variants and extensions of the different parts of the generic TSM approach. Hence, a new Java-based implementation is essential.

Eiglsperger et al. considered in 2003 the TSM approach with respect to different layout approaches of UML diagrams [EKS03]. In that context he considered mixed upward planarization to allow directed edges in the diagrams. Chimani, Gutwenger, Mutzel, and Wong et al. investigated upward planarization in 2008 as well [Chi+10]. Their approach produces much better results for upward crossing minimization.

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

1.2 Contributions

The focus of this work lies on the embedding of the existing parts of the TSM algorithm into a generic architecture and on the completion of the implementation of the algorithm as well as the evaluation and partial implementation of possible extensions.

In that context some changes are done to extend the implementation of the existing planarization and orthogonalization. Additionally, the implementation for finding a suitable external face was considered.

The compaction with all its intermediate processors was implemented to finalize the whole algorithm in order to layout first diagrams. In that context, the compaction phase itself was created. It consists of the creation of a flow network and its solution in order to achieve small edge lengths. Moreover, the technique for the creation of grids and the mapping of the graph elements on these grids were implemented.

An extension of the compaction was the realization of more general input graphs, especially graphs that are not in rectangular shape. A mechanism was introduced that adjusts input graphs with dummy nodes to be in rectangular face. Secondly, a process was considered that subdivides edges with bends into edges with dummy nodes at the positions of the bends. Furthermore, a method was implemented to remove all dummies of the graph correctly in order to ensure the graph represents its original.

In addition, the algorithms were extended to allow cutedges and cutvertices. Furthermore, methods to allow graphs with high-degree nodes are evaluated and two of them are implemented, namely the Giotto approach and the Quod approach. Additional possible extensions of that algorithm were evaluated like interactive planarization and planar drawing alternatives. Besides all that, the handling of directed edges, port-constraints and edge labels are considered. Implementation ideas for handling self-loops and multi-edges were investigated.

1.3 Overview

In Chapter 2 basics of graph theory are presented. Furthermore, flow networks are described as well as graph drawing conventions. Different aesthetics criteria of drawings are considered, and the TSM layout approach is presented. The implementation of that algorithm is embedded in the research project KIELER.

Finally, the general architecture of the layout algorithms implemented in KIELER is presented.

Afterwards the three parts of the TSM algorithm are considered. Firstly, the planarization is presented in Chapter 3. In that context, the planarity testing and the additional edge insertion phase are described. Chapter 4 forms the orthogonalization. Here, Tamassia's algorithm for setting angle- and bend-data to the graph is considered. In that process the creation of a flow network in the context of orthogonalization and its solution is presented. Finally, the step to calculate the external face of the input graph is discussed.

Chapter 5 considers among others the main contributions of the this thesis. First the compaction phase itself is discussed. The creation and the solving of flow networks to minimize the lengths of the graph edges are considered. Additionally, a lot of steps around that phase to allow more general input graphs are discussed. A part of these steps is to bring the input graph in rectangular shape. In that process the general method to transform the internal and external faces of a graph in rectangular shape is investigated. Moreover, the implementation of that method is presented. Additionally, the special case for handling cutvertices and cutedges is presented. After the compaction phase has calculated the relative edge length the technique for adding graph nodes on positions into a grid is shown. Afterwards the removal of the different dummies of the graph is discussed.

The second main chapter forms Chapter 6, which includes the evaluation and the presentation of the implementation of some extensions. Especially the evaluation and implementation of the Giotto approach and the Quod approach to allow graphs with higher degree than four are described. Besides that approach, some methods to draw planar graphs are presented, and the possible realization of the interactive planarization in the implemented algorithm is shown. Furthermore, the finding of an optimal embedding to let the external face be suited is presented. Some additional desired extensions are hyperedges, edge labels and port-constraints which are considered as well. Additionally, the possible implementation of some smaller basic extensions are discussed, such as multi-edges and self-loops.

Finally, a conclusion is given in Chapter 7 to complete the thesis. In that process an evaluation of the different layout algorithms of KLay is presented.

Chapter 2

Preliminaries

This chapter considers definitions to set the base for the graph drawing environment. The essentials are introduced here bases mainly on [Di +99] and [KW01]. Furthermore, flow networks are introduced. Then, aesthetics criteria of graph drawing are presented, and the TSM approach is discussed. Moreover, the KIELER as a platform for implementing and testing layout algorithms is introduced, which includes an implementation of the TSM algorithm. This chapter ends with the description of the generic architecture for the different layout algorithms used in KIELER with special focus on the TSM algorithm.

2.1 Basics of Graph Drawing

Definition 2.1 (Graph, Subgraph)**.** A *graph* $G = (V, E)$ consists of a finite set of *vertices* V and a finite set of *edges* E. An edge $e \in E$ with $e = (v, w)$ represents a pair of nodes where $v, w \in V$. e is a denoted as a *self-loop* if $v = w$. A graph $G' = (V', E')$ is a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$.

Definition 2.2 (Degree)**.** The *degree* of a vertex v, called $deg(v)$, is the number of edges incident to the vertex. Self-loops count twice.

Definition 2.3 (Directed, Undirected)**.** A graph is *directed* if all pairs of E are ordered. v is called *source* and w is called *target* of a directed edge (v, w). If all edges are unordered the graph is called *undirected*. A *mixed graph* contains directed and undirected edges.

Definition 2.4 (Completeness)**.** An undirected graph G is *complete* if every pair of its distinct vertices is connected by an unique edge.

Definition 2.5 (Bipartiteness)**.** A graph is a *bipartite* if its vertices can be divided into two disjoint sets U and V such that every edge connects a vertex in U to one in V; that is, U and V are each independent sets.

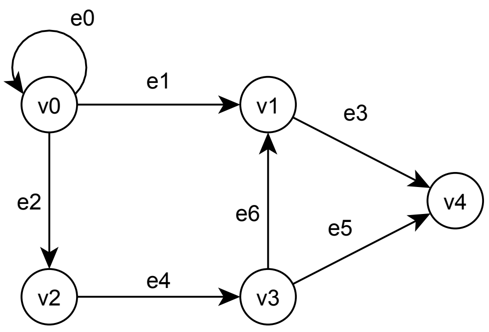
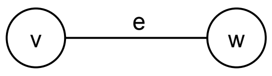


Figure 2.1. Directed graph with self-loop.

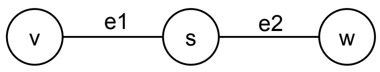
Definition 2.6 (Drawing)**.** A *drawing* Γ of a graph G is a mapping of each vertex v to a distinct point $\Gamma(v)$ and each edge (v, w) to a simple open Jordan curve with endpoints $\Gamma(v)$ and $\Gamma(w)$. A directed edge is often drawn with an arrow.

Figure 2.1 shows a drawing of a directed graph $G = (V, E)$ with the described definitions where $V = \{v_0, v_1, v_2, v_3, v_4\}$, $E = \{e_0, e_1, e_2, e_3, e_4\}$. For example, v_0 is source and v_1 is target of e_1. $e_0 = (v_0, v_0)$ is a self-loop on v_0 and e. g., $deg(v_0) = 4$ and $deg(v_1) = 3$.

Definition 2.7 (Path)**.** A (directed) *path* in a (directed) graph $G = (V, E)$ is a sequence $(v_0, v_1, ..., v_n)$ of distinct vertices of G with $(v_i, v_{i+1}) \in E$ for $i \in 0, \dots, n-1$. A path is a (directed) *cycle* if $(v_n, v_1) \in E$. A directed graph is *acyclic* if it has no directed cycles.



a. Simply connected edge.



b. Subdivision of the edge of Figure 2.2a.

Figure 2.2. Subdivision example.

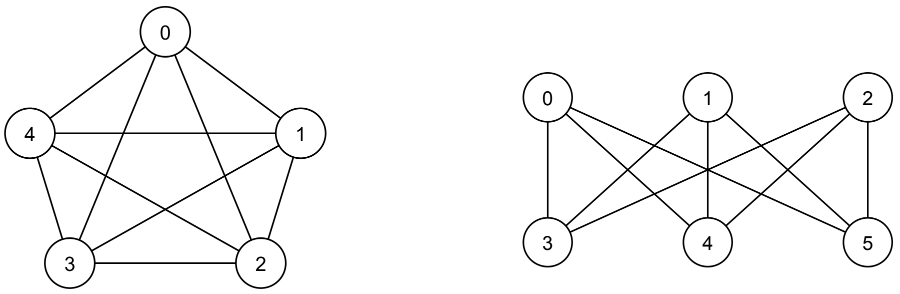
In the following, an important term of the context of this thesis is introduced, the *planarity*.

Definition 2.8 (Planarity)**.** A drawing Γ is *planar* if no two distinct edges cross. A graph is planar if there is a planar drawing of G.

A graph is called to be maximal planar if it is planar but adding any edge would destroy that property.

Definition 2.9 (Subdivision)**.** A *subdivision* of a graph G is a graph resulting from the subdivision of edges in G. The subdivision of an edge $e = (v, w)$ yields a graph containing one new vertex s with a set of edges replacing e with two new edges, $e_1 = (v, s)$ and $e_2 = (s, w)$.

Figure 2.2 illustrates an example for a subdivision. The edge $e = (v, w)$ can be subdivided into two edges, $e_1 = (v, s)$ and $e_2 = (s, w)$, connecting to a new vertex s.



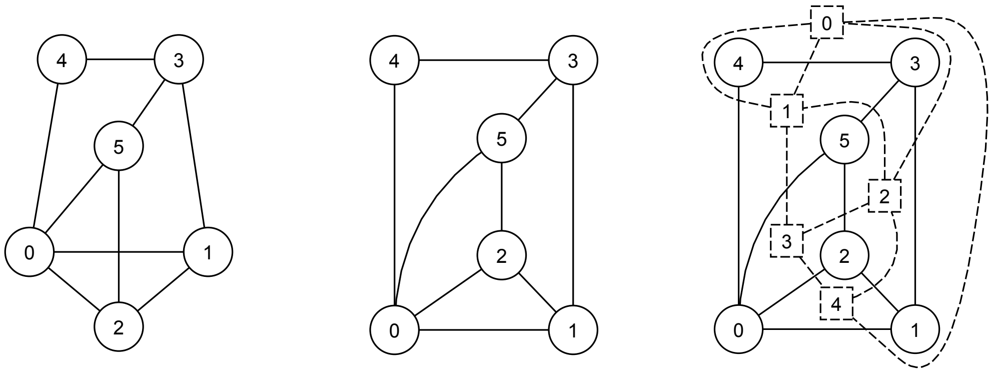
a. Drawing of K_5. **b.** Drawing of $K_{3,3}$.

Figure 2.3. Drawings of the smallest non-planar graphs.

Figure 2.3 shows drawings of two special graphs, the K_5 and the $K_{3,3}$. K_5 is a complete graph with 5 vertices. $K_{3,3}$ is a complete bipartite graph, which is divided into two sets. Each set consists of three elements. Both graphs are not planar, since there is no representation of that graph that is planar. They are even the smallest graphs that are not planar.

Theorem 2.10 (Kuratowski[1])**.** *A finite graph is planar if and only if it does not contain a subgraph that is a subdivision of K_5 and $K_{3,3}$.*

Definition 2.11 (Embedding, Equivalent)**.** A (planar) *embedding* of G is an equivalence class of drawings that is defined by the circular order of the adjacent edges of each vertex v of a graph. If each vertex v of G has drawings with same circular sequence of adjacent edges around v for two (planar) embeddings, they are called to be *equivalent*.



a. Non-planar drawing.

b. Planar drawing of Figure 2.4a with another embedding.

c. The dual graph of the drawing of Figure 2.4b.

Figure 2.4. Different representations of a graph.

An embedded planar graph is usually called *plane* graph. Figure 2.4a and Figure 2.4b present two different drawings for the same graph. The drawing of Figure 2.4a is not planar since the edges $(0, 1)$ and $(2, 5)$ cross. Changing the embedding of that graph leads to a planar drawing as shown in Figure 2.4b. The example graph is planar since there is at least one planar drawing.

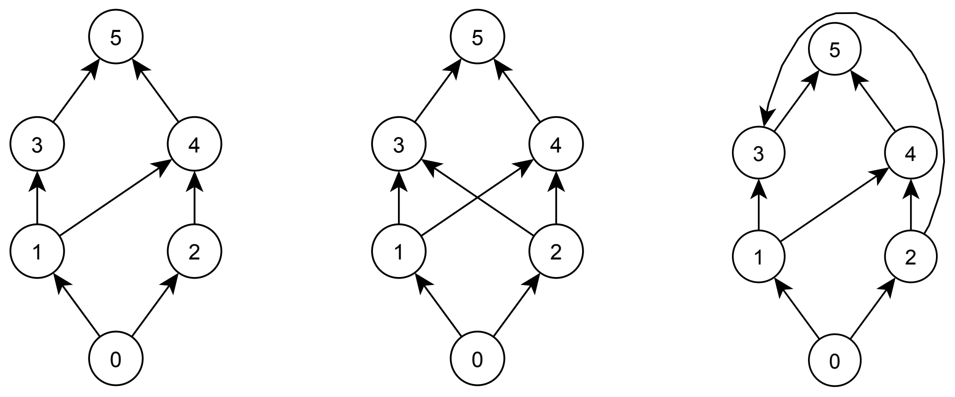
Definition 2.12 (Face)**.** A planar drawing partitions the plane into regions called *faces*. The unbounded face is called *external face*.

Definition 2.13 (Dual Graph)**.** The *dual graph* $G^\star$ of an embedding of a planar graph G has a vertex for each face of G and an edge (f, g) connecting every two faces f and g that are adjacent by an edge in G.

Figure 2.4c shows the planar drawing of Figure 2.4b and its dual graph. The five faces of the original plane are the vertices of $G^\star$ that are connected with dashed lines. A graph can be divided into different levels of connectivity.

Definition 2.14 (Connectivity)**.** A graph is *connected* if there is a path between every vertex in that graph. A *cutvertex* is a node that divides the graph into two components if it is removed. A *biconnected* graph contains no cutvertices.

[1]http://en.wikipedia.org/wiki/Planar_graph



a. Upward planar drawing. **b.** Upward non-planar drawing. **c.** Non-upward planar drawing.

Figure 2.5. Different drawings illustrating upward planarity.

Definition 2.15 (ST-Graph)**.** A *st-graph* is an acyclic directed graph with a single source s and a single sink t with following properties:

▻ Given a topological numbering of G, every directed path of G visits vertices with increasing numbers.

▻ For every vertex v of G, there is a simple directed path from s to t containing v.

Definition 2.16 (Planar ST-Graph)**.** A *planar st-graph* is a st-graph which is planar and embedded with vertices s and t on the boundary of the external face.

Definition 2.17 ((Mixed) Upward Drawing)**.** A drawing of a directed / mixed graph is called (mixed) upward drawing if each (directed) edge is represented by a curve monotonically increasing in the vertical direction.

Definition 2.18 ((Mixed) Upward Planarity)**.** A *(mixed) upward planar drawing* of a mixed / directed graph is a (mixed) upward drawing with no edge crossing. A graph G is called (mixed) upward planar if it has a planar (mixed) upward drawing.

Figure 2.5 shows different drawings to make the definition of the upward planarity clearer. Figure 2.5a illustrates a drawing that is upward planar, while Figure 2.5b is not planar since the edges $(1, 4)$ and $(2, 3)$ cross. This drawing can be adjusted to be planar as seen in Figure 2.5c, but then it looses the upwardness property.

2.1.1 Flow Networks

In the following, the flow network is introduced. It leads to the minimum cost flow problem that is needed in the context of the orthogonalization and the compaction. A complete overview about network flows and especially solving minimum cost flow problems is presented by Ahuja et al. [AMO93].

Definition 2.19 (Flow Network)**.** A *flow network* $\mathfrak{N} = (N, A)$ is a directed graph.

- ▻ Each *source node* has a production and respectively each *sink node* has a consumption.
- ▻ The sum of produced and consumed units of every node in the whole drawing is equal.
- ▻ An arc (v, w) contains the following properties:
  1. A *lower bound* $b_{vw} \in \mathbb{R}$.
  2. A *capacity* $u_{vw} \in \mathbb{R}$.
  3. A *cost* $c_{vw} \in \mathbb{R}$.

For a better distinction to the original graph G we call vertices of $\mathfrak{N}$ *nodes* and edges of $\mathfrak{N}$ *arcs*.

Definition 2.20 (Feasible Flow)**.** A *feasible flow* x is a function $x : A \to \mathbb{R}$ with following properties:

- ▻ Capacity: $b_{vw} \leqslant x_{vw} \leqslant u_{vw}\ \forall (v, w) \in A$
- ▻ Mass balance: $b(v) = \sum_{w:(v,w)\in A} x_{vw} - \sum_{w:(w,v)\in A} x_{wv}\ \forall v \in N$

Each node of $v \in N$ is associated with a number $b(v)$ denoted *supply value*, such that

- ▻ if $b(v) > 0 \to v$ is a supply node,
- ▻ if $b(v) = 0 \to v$ is a transport node, and
- ▻ if $b(v) < 0 \to v$ is a demand node.

The first property of the feasible flow definition states that the flow of an edge (v, w) cannot be less than the lower bound and cannot top the capacity border u_{vw}. The second property states that the subtraction of the total supply of a node minus the total demand of the node is $b(v)$ (supply / transport / demand value) of this node. For each non-source node and non-target node the amounts of the flow of the incoming arcs and of the outgoing arcs have to be equal (flow conservation).

Next, the minimum cost flow problem is introduced. Such a calculation is one of the fundamental problems in the network flow theory and has been studied extensively in the past.

Definition 2.21. The *cost* of flow x in $\mathfrak{N}$ is defined as $c(x) = \sum_{(v,w)\in A} c_{vw} \cdot x_{vw}$.

Definition 2.22 (Minimum Cost Flow Problem)**.** Minimize $c(x)$ for a given flow network $\mathfrak{N}$ with source $s \in N$ and target $t \in N$, with required flow d such that the following constraints are satisfied:

▻ *Capacity constraints:* $x_{vw} \leqslant u_{vw}$

▻ *Required flow:* $\sum_{(v,w)\in A} x_{vw} = d$

▻ *Skew symmetry:* $x_{vw} = -x_{wv}$

▻ *Flow conservation:* $\sum_{(v,w)\in A} x_{vw} = 0$, with $v \neq s$ and $w \neq t$

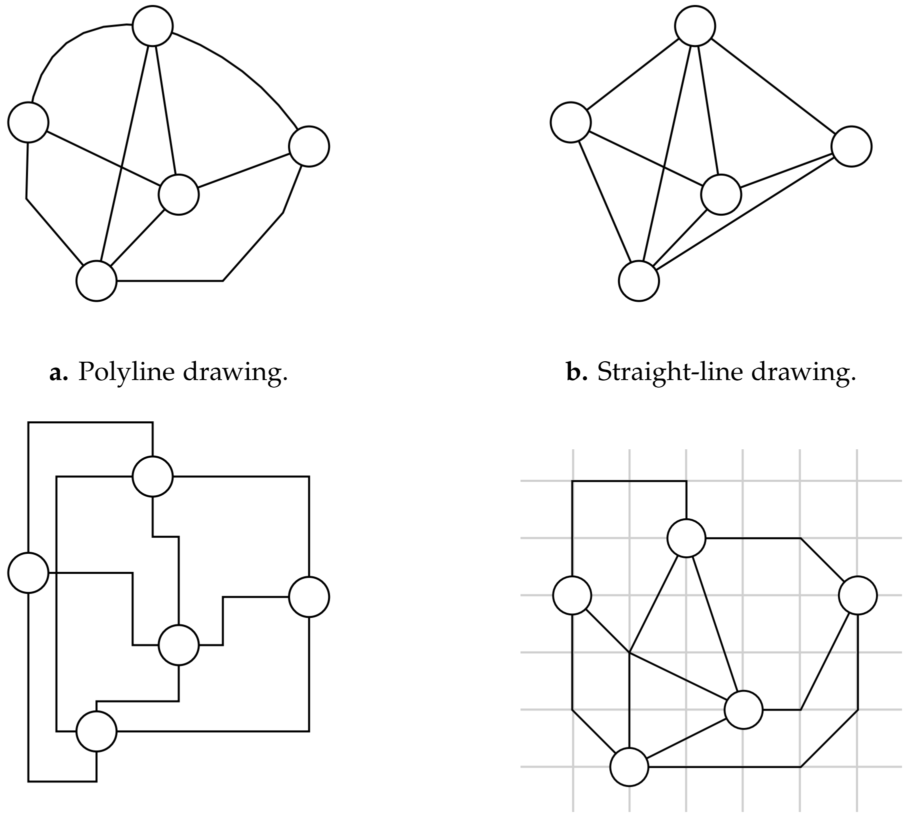
To work with minimum cost flow one has to assume that every node is reachable by another node and that there is at least one solution of the Minimum Cost Flow (MCF) problem.

2.1.2 Drawing Conventions

As seen above, a graph can be drawn in many different ways. Thus, the concept of drawing conventions is introduced which defines basic rules for drawings that have to be satisfied to be admissible [Di +99].

▻ *Polyline Drawing:* Each edge is drawn as a polygonal chain.

▻ *Straight-line Drawing:* Edges are drawn as straight line connections between two vertices.



a. Polyline drawing.

b. Straight-line drawing.

c. Orthogonal drawing.

d. Grid drawing.

Figure 2.6. Different drawings showing drawing conventions.

▻ *Orthogonal Drawing:* Edges are polygonal chains that only consists of horizontal and vertical segments. A segment is bounded by a vertex or a bend of that edge.

▻ *Grid Drawing:* Each vertex, bend, and crossing of the graph get an integer coordinate on a grid.

2.1.3 Aesthetics Criteria

A good layout can be a picture worth a thousand words; a poor layout can confuse or mislead. Graphs are used to represent information and structure in various areas of the software engineering. To achieve the readability of the information presented, properties of a drawing are specified, the so called *aesthetics criteria* [Di +99].

▻ *Minimization of edge crossings;* Ideally, there is a planar drawing so that there is no edge crossing, but not every graph admits one. If there is no planar embedding the goal is to find a drawing with a minimal total number of crossings between edges.

▻ *Minimization of the drawing area;* It is essential in practical visualization systems to save screen space. Furthermore, it is relevant if one cannot arbitrarily scale the graph down.

▻ *Minimization of the edge length;* This criterion is divided into three similar minimization concepts:

1. *Total edge length:* Minimize the sum of the edge lengths.
2. *Maximum edge length:* Minimize the maximum edge lengths.
3. *Uniform edge length:* Minimize the variance of the edge lengths.

▻ *Minimization of the bend number;* This criterion contains three concepts likewise the criterion for the edge length:

1. *Total bend number:* Minimize the total number of bends along the edges.
2. *Maximum bend number:* Minimize the maximum number of bends on an edge.
3. *Uniform bend number:* Minimize the variance of the number of bends on an edge.

▻ *Minimization of the aspect ratio;* Aspect ratio is defined as the ratio of the length of the longest side to the length of the shortest side of the smallest rectangle with horizontal and vertical side covering the drawing. Drawings with high aspect ratio may not be conveniently placed on a screen, even if it has small area.

In the following, the TSM approach after Tamassia is described. This algorithm is suited for orthogonal grid drawings and hence useful in industrial plans like data flow modelings or in UML class diagrams.

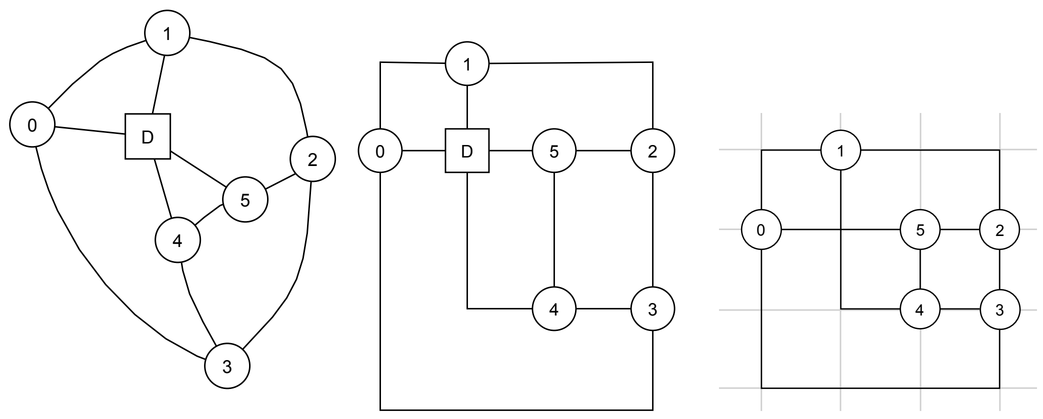
2.2 Topology-Shape-Metrics Approach

Roberto Tamassia et al. introduced the TSM algorithm first in 1987 and in 1988, respectively [Tam87; TDBB88]. The idea behind that approach is that an orthogonal drawing can be described by three properties, defined in the terms topology, shape and metrics that are used as follows [Di +99]:

- *Topology:* Two orthogonal drawings are topologically equal if one can be obtained from the other by continuous deformation that does not alter the sequence of edges contouring the faces of the drawing.
- *Shape:* Two orthogonal drawings have the same shape if, firstly, they have the same topology and secondly, one can be obtained from the other by modifying only the length of edges without changing their angles.
- *Metrics:* Two orthogonal drawings have the same metrics if they are congruent, up to a translation and/or rotation.

According to these three properties the TSM algorithm is divided into the following three steps:

1. *Planarization:* This step determines the topology of the drawing which is described by a planar embedding. This step reduces the number of edge crossings as much as possible. One way is to find an embedding that is planar, if there is one. Otherwise, a maximal planar subgraph is built and all violating edges are removed, such that the result is planar. In a second step these violating edges are added to the graph again by inserting one dummy vertex for each edge crossing.
2. *Orthogonalization:* This step determines the shape of the plane by calculating the angles and the bends in the drawing. The goal of this step is to minimize the number of bends without changing the topology. This is done by creating a network flow model that has to be minimized to get the angles of edges around a node and the correct placement of bend-points.
3. *Compaction:* Final coordinates of the vertices and edge bends are determined, and additionally, the previously added dummies are removed. In this step the main goal is to minimize the drawing area, by minimizing the length of the edges.



a. Result of planarization. **b.** Result of orthogonalization. **c.** Result of compaction.

Figure 2.7. Steps of the TSM approach.

The result of processing these three steps on a graph is an orthogonal shape with few edge crossings and with small edge length respectively small area space. Figure 2.7 shows the results of every step of the algorithm with an example. The considered example is the $K_{3,3}$ graph of Figure 2.3b that can not be drawn planar, and its concrete structure is $(\{0,1,2,3,4,5\},\{(0,1),(0,3),(0,5),(1,2),(1,4),(2,3),$ $(2,5),(3,4),(4,5)\})$. In the first phase a maximal planar subgraph is calculated, and the remaining violating edges are removed, here edge $(1,4)$. In a second phase, still part of the planarization, a dummy node is inserted at the place the edges would cross, and the removed violating edges are inserted again, such that the crossing edge $(0,5)$ and the violating edge $(1,4)$ are connected to the new dummy node. The result is a planar embedding of the original graph which contains dummy nodes that avoid edge crossings, see Figure 2.7a.

Such a plane is needed to process the orthogonalization phase. The result of this phase is presented in Figure 2.7b, that is a representation with angle-data around a node and minimum bend-data for each edge. Orthogonality is ensured because only multiples of 90° are assigned as angles and each edge consists of segments that are only horizontal or vertical. The resulting representation contains no real coordinates, it defines only the shape.

In order to set such coordinates to the graph elements the final compaction phase is used. This phase makes the drawing as compact as possible by minimizing the edge length. Additionally, the planar dummy nodes are removed and the edges are connected with their original source and original target nodes. The result, shown in Figure 2.7c, is an orthogonal and compact drawing of the given graph with few edge crossings.

The order of the steps taken represent an order of importance of aesthetics criteria as well. The planarization is the first step, such that the edge crossing minimization is the most important criterion. Then minimizing the number of bends with the orthogonalization is the second most important criterion, and least important is the minimization of the drawing area that is determined by the compaction. The result is that each phase affects the aesthetics criteria of its successor phase.

In addition, a drawing could consists of less bend-points if the orthogonalization would be processed first, and respectively, a drawing could be drawn with less space if the compaction would be processed before. Such a phase swapping is not possible, since each successor step requires the changed graph structure as input, as done by the preceding step. Without the preceding steps each phase would form a NP-hard problem.

An implementation of Tamassia's approach is part of the layout library of the following project.

2.3 Kiel Integrated Environment for Layout Eclipse Rich Client

KIELER[2] is an academical research project that aims to enhance the graphical model-based design of complex systems [Fuh11]. The main idea is to consistently employ automatic layout in all graphical components of the diagrams within the modeling environment. This leads to new possibilities for diagram editing, browsing, and dynamic visualizations. This project continues the Kiel Integrated Environment for Layout (KIEL) project. While KIEL concentrates on a single modeling language, namely *Statecharts*, KIELER aims to integrate different modeling languages into the rich client platform Eclipse[3]. KIELER is licensed under the Eclipse Public License [4],

[2]http://www.informatik.uni-kiel.de/rtsys/kieler/
[3]http://www.eclipse.org
[4]http://www.eclipse.org/legal/epl-v10.html

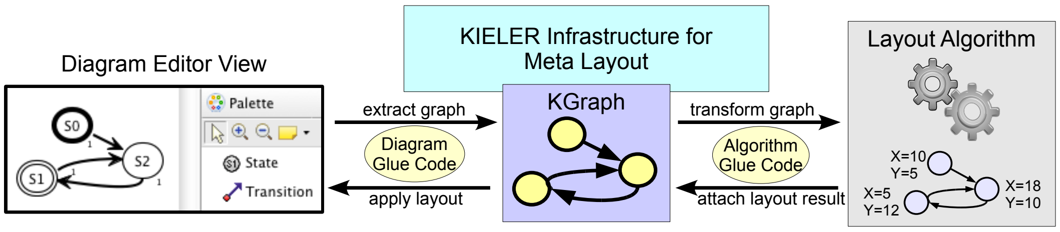


Figure 2.8. Overview of KIML structure [Fuh11].

an open source software license.

One of the main concepts of KIELER is to offer automatically generated layouts, to support the creation and maintenance of diagrams as well as modern dynamic visualization techniques. Thus, the user of KIELER is free from so-called *enabling steps*, which are things that have to be done to prepare the model for changes the user actually wants to make. Normally, these are things like making space for nodes by moving nodes of the diagram around, or taking care of the edge routing and so on. The KLay project provides Java-based implementations of layout algorithms for the automatic layout feature that can be used with any graphical editor. The core component of bridging the layout algorithms of KLay and graph editors is the KIELER Infrastructure for Meta Layout (KIML). KIML provides a bridge that puts the content of the editor in a data structure that can be understood by a layout algorithm. A brief overview of KIML is presented in the following subsection.

2.3.1 KIELER Infrastructure for Meta Layout

Apart from layout algorithms themselves, the topic of automatic layout consists of the problem of getting the diagram in a format understandable by the layout algorithm and applying the resulting layout back to the diagram. KIML solves such problems as generically as possible, such that many algorithms can be used for many diagram editors. Furthermore, KIML allows layout algorithms to offer layout options for the user that affect the resulting layout, for instance, setting a minimum distance between the nodes. Figure 2.8 illustrates the structure of KIML.

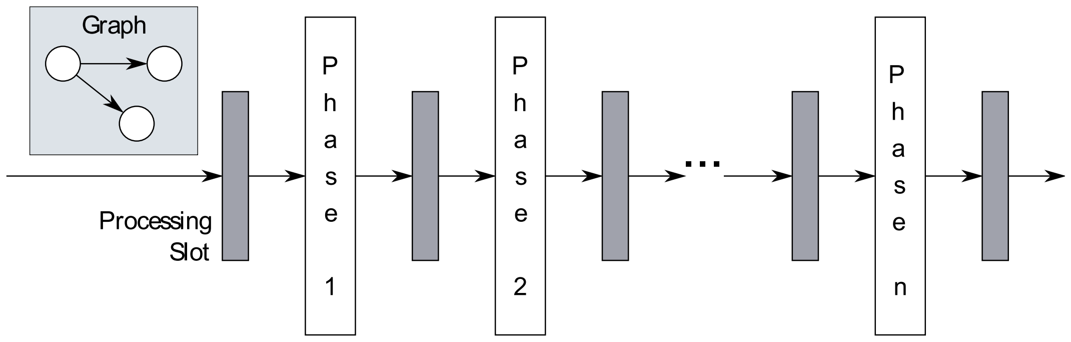


Figure 2.9. KLay architecture overview [Sch11].

A lot of existing layout algorithms are provided, libraries like Graphviz [Ell+02] and OGDF [Chi+07]. On the other hand the library of KLay includes implementations of own layout algorithms that are

▻ a force-based algorithm (*KLay Force*),

▻ a layerd-based algorithm (*KLay Layered*) [Spö09; Sch11; Car12], and

▻ the TSM algorithm (*KLay Planar*) [Kut10; Cla10]

which is considered in this thesis. The implementations of KLay base on a special architecture that is presented in the following section.

2.4 KLay Architecture

This architecture was presented first by Christoph Daniel Schulze [Sch11] in the KLay Layered implementation (see Figure 2.9). It is divided into main phases and several small intermediate processors.

A phase is basically an own part of the algorithm like the orthogonalization in the KLay Planar algorithm. There are empty slots before and after a phase that can be filled with intermediate processors, which solve tasks for each phase like bringing the data structure in readable form or cleaning up the data structure. This concept provides a high degree of exchangeability and extensibility, since each phase can be exchanged with another variant or can be extended with new features by changing the phase itself or introducing a new intermediate processor.

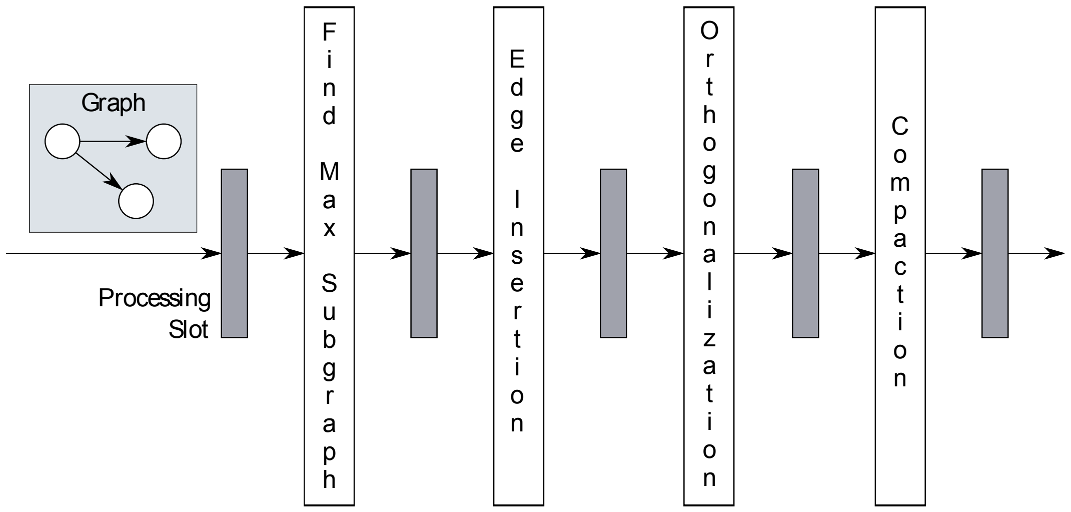


Figure 2.10. KLay Planar architecture.

In addition, the code is better readable since the complex layout algorithm is divided into smaller parts (phases), which taken by itself are easier to understand than the whole algorithm. Hence, the whole algorithm can be maintained easier.

2.4.1 KLay Planar Architecture

The phases and intermediate processors concept is suitable for the implementation of the TSM algorithm because the TSM algorithm is also divided into different parts. The planarization step consists of two parts with own context and own functionality, building the maximal planar subgraph and the additional edge reinsertion. Both pieces are part of the planarization algorithm itself and hence, they are handled as phases. Additionally, the orthogonalization and the compaction form own sections of the algorithm, thus they are both handled as single phases in the implementation.

The graph structure has to be adjusted between these phases, e. g., the faces of a planar embedding are computed before processing the orthogonalization, or a grid drawing can be calculated for the result of the compaction, for a more comfortable and efficient processing on the drawing. For such tasks intermediate processors are created and are added to the corresponding slots between the phases.

In the following chapters different phases of the TSM algorithm and intermediate processors as well as their realizations are discussed.

Chapter 3

Planarization

The task of planarization is to find a planar embedding of a given graph g. If there is such a plane graph we are finished. Otherwise a planar subgraph of g is taken. In a second step the missing, violating edges are added again to the graph with a dummy node for every crossing such that the resulting graph is planar, but contains some dummy nodes.

The dummy nodes do not distort the subsequent phases of the TSM algorithm and can be kept until the compaction phase is finished. After this last phase of the algorithm the planarization dummies are removed. Additionally, the edges are connected back to its original vertices.

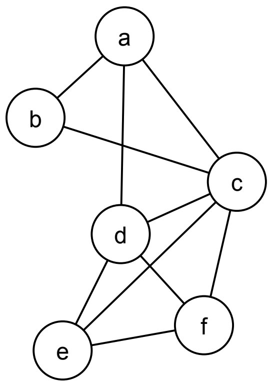
This chapter starts off with an introduction of an algorithm for planarity testing. in addition, the strategy for inserting dummy nodes is described.

3.1 Planarity Testing

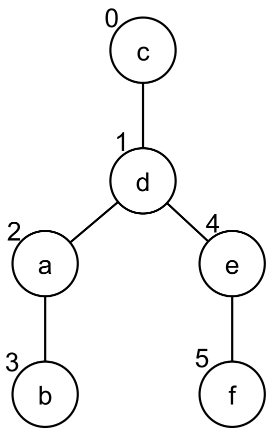
The presented technique bases on the work of Ole Claußen, who implemented the algorithm of Boyer and Myrvold [Cla10]. This algorithm checks a graph for planarity, and if the graph is not planar, it is able to find a planar subgraph of the given graph in linear time to the number of vertices [BM04]. The calculated planar representation contains a fixed order of edges on each vertex in counter-clockwise direction and therefore defines a topology for the given graph.

The general computation of planar subgraph with minimum edge crossings is known to be NP-hard [Yan78]. Among others, the reason is that a given graph can have exponentially many planar embeddings. Thus, testing algorithms compute a *maximal* planar subgraph, such that adding any missing edge of the subgraph would break the planarity property. Hopcroft and Tarjan presented the first linear time planarity testing algorithm, and several extensions and improvements have emerged until today [HT74; BM04; MM96; JLM98].

Boyer and Myrvold's algorithm first calculates a Depth First Search (DFS) on the given graph, which results in a spanning tree, generally known as the DFS-tree.



a. Non-planar drawing.



b. DFS tree of the drawing of Figure 3.1a.

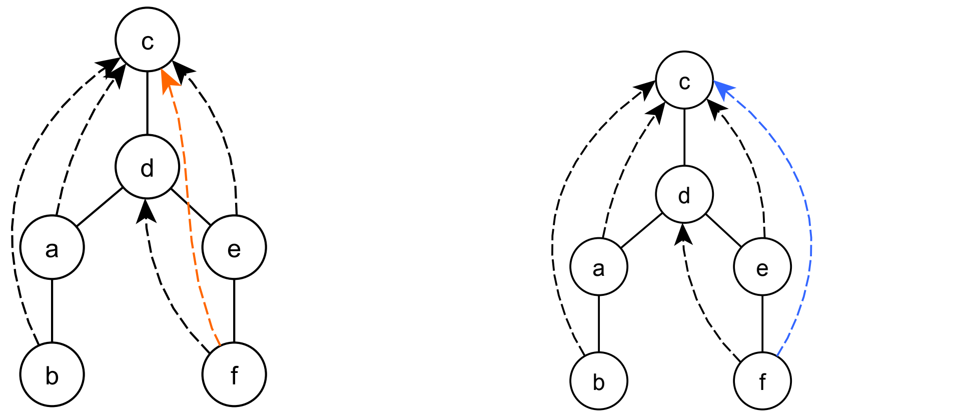
Figure 3.1. Creation of a DFS-tree.

Definition 3.1 (DFS-Tree)**.** A DFS-tree of a graph is a spanning tree emerged by a DFS. Nodes that have been visited before are not visited again. Each node is ordered by an index, called DFS Index (DFI).

Since visited nodes are not visited again not all edges of a given graph might be part of the tree. It is immediate that every DFS-tree is planar. These edges are called *back-edges*. In the following, the algorithm of Boyer and Myrvold is described, which is implemented in KLay Planar. Listing 3.1 illustrates the steps of that algorithm.

Listing 3.1. Algorithm of Boyer and Myrvold.

```
Procedure($g$ : Graph){
  calculate a DFS-tree;
  order its nodes according DFI ($v_0, v_1, ..., v_n$)
  for ($v_i : v_n, ..., v_1, v_0$){
    embed $v_i$ planar, together with all back-edges $(v_j, v_i), j > j$;
    if(embedding of a back-edge is not possible){
      return $g$ is not planar;
    }
  }
  return $g$ is planar;
}
```



a. Back-edge insertion.

b. Swapping of the wrong back-edge of Figure 3.2a.

Figure 3.2. Back-edge insertion.

First, an arbitrary DFS-tree is calculated with an arbitrary start node, such that each node can be identified by a DFI. Then, the nodes of the tree are processed in the opposite order in which they have been added. Thus, the graph rebuilding is started with the leaves of the DFS-tree, and the graph is processed upward to the root. In that process all back-edges are planar embedded. With every cycle of the algorithm, the invariant ensures that nodes that are involved in later embeddings are part of the external face. To ensure this invariant there is a possibility that the biconnected components need to be turned around prior to their embedding. If an embedding fails, meaning there are crossings in the tree, the given graph g is not planar. If all edge embeddings are complete, it is a planar embedding of g.

An example for this algorithm can be seen in Figure 3.1. The graph of the drawing of Figure 3.1a is checked to be planar. The first step is to create the DFS-tree with arbitrary root, here c (see Figure 3.1b). Its node list is (c, d, a, b, e, f). The DFI of each node is labeled with a small number next to the node.

Afterwards, the back-edges are reinserted to the graph (see Figure 3.2). A problem exists at the embedding of (c, f). Hence, the embeddings of f are swapped, such that the counter-clockwise order of the edges is changed and the problem is solved. This is illustrated with the blue non-crossing edge in Figure 3.2b.

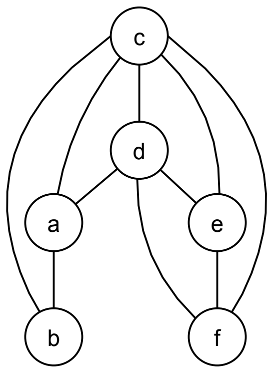


Figure 3.3. Planar embedding of Figure 3.1a.

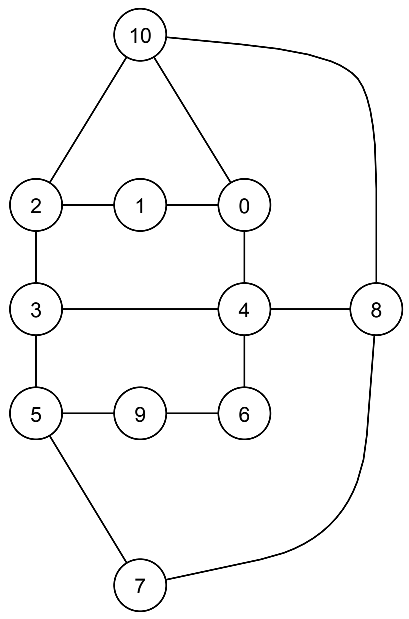
Since all back-edges are embedded correctly, the graph is planar, hence the embedding can be used to do the next phases of the TSM algorithm.

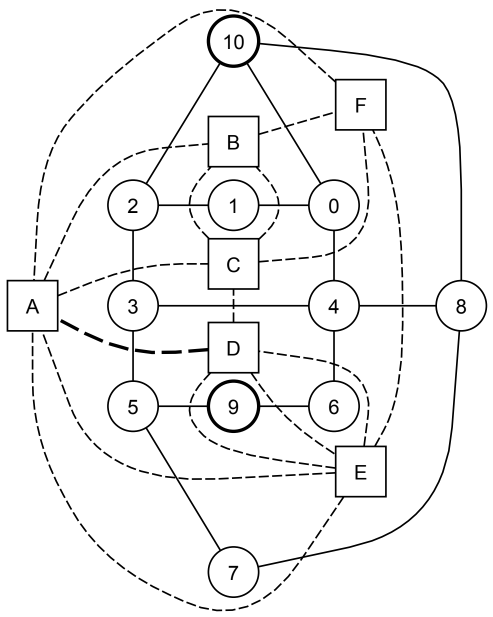
A more detailed description of that algorithm is presented in the thesis of Ole Claussen [Cla10]. The example graph of Figure 3.3 is planar. If a given graph is not planar, the algorithm calculates an embedding of a planar subgraph and a set of violating edges. As described before, each violating edge is inserted again by adding a dummy node to the position where the edges would cross. This process is described in the following section.

3.2 Edge Insertion

The task of this algorithm is to embed violating edges back into the planar graph, while creating a minimum number of dummy vertices. This is done by calculating a path for the new edge that crosses a minimum number of faces. The shortest edge path in the dual graph corresponds to this path.

The algorithm to embed the remaining edges (v, w) is as follows. First, the dual graph $G^\star$ of the input graph G is processed. Then, Dijkstra's algorithm is used to calculate the shortest path in $G^\star$ for every face adjacent to v and every face adjacent to w. Afterwards, the edge (v, w) is added to the graph along this path. In that process for every edge in $G^\star$ a dummy vertex is inserted in G. Then, this dummy vertex is connected with edges, crossing the involved faces to avoid edge crossings between these edges and the inserted edge (v, w).





a. Example of a planar embedding. **b.** Dual graph of Figure 3.4a.

Figure 3.4. Edge insertion example.

An example for the described algorithm is presented in Figure 3.4. It is assumed that an edge $(9, 10)$ is desired to be inserted into the embedding of Figure 3.4a. The dual graph of the embedding is calculated, which is illustrated in Figure 3.4b. Then, the source of the path would be $s \in \{D, E\}$ and the target would be $t \in \{A, B, F\}$. A shortest path, calculated by Dijkstra's algorithm, would be (D, A), such that the edge $(9, 10)$ would be placed along that path. The edge $(3, 5)$ is adjacent to the faces A and D. To avoid the crossing of edge $(3, 5)$ and $(9, 10)$, a dummy node is inserted to subdivide the two edges, as in Figure 3.5.

The determination of the faces of an embedding is assumed in the previously described algorithm. This process is explained in the following section.

3.2.1 Face Calculation

Christian Kutschmar implemented a method to determine the faces of a planar embedding [Kut10]. The aim of that processor is to walk along the edges. In that

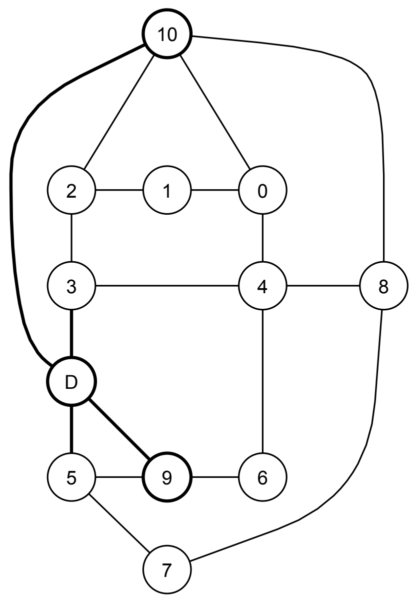


Figure 3.5. Result of the edge insertion of Figure 3.4.

process the adjacent faces of each edge in the planar embedding are calculated. The correct behavior is ensured since the edges of a planar embedding around a vertex are ordered counter-clockwise (see Section 4.2). Hence, each successor edge, whether clockwise (last counter-clockwise) or counter-clockwise, of an edge at a vertex can be calculated uniquely.

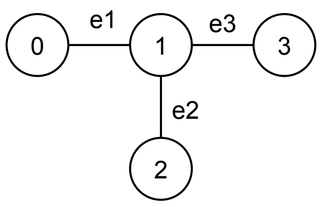
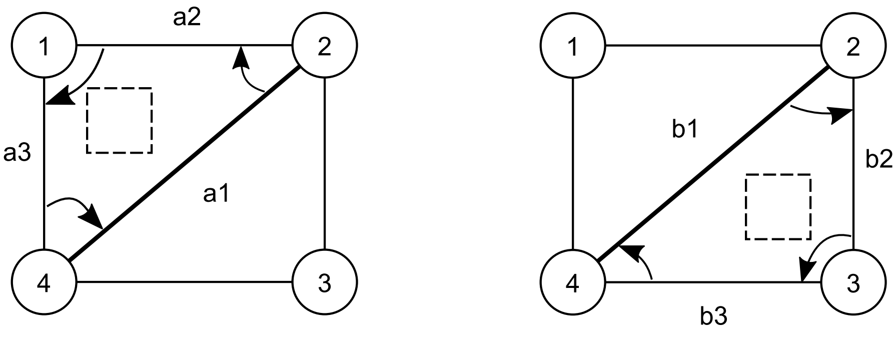


Figure 3.6. Simple connected graph example.

The algorithm starts by checking all edges whether the left or right face of that edge is unknown. If the left face of the edge is unknown a circular walk-through is



a. Finding the left face of edge a_1. **b.** Finding the right face of edge b_1.

Figure 3.7. Calculation of the left and right face.

triggered, which can be seen in Figure 3.7a. Each found edge is added to the face until the start node is reached. The resulting face of the example consists of the edges a_1, a_2, and a_3. Respectively, for an unknown face on the right hand side of an edge a walk-through in counter-clockwise order is started which is presented in Figure 3.7. The resulting face in the example contains the edges b_1, b_2, and b_3.

Listing 3.2. A snippet of the face calculation.

```
Procedure($g$ : Graph) {
        for ($e \in E$) {
            if left face of $e$ is unknown then
                create new face $f$
                set left face of $e = f$
                set $n_0 =$ start node of $e$
                set $n_1 = n_0$
                set $e_0 = e$
                set $e_1 = e$
                ...
                do {
                    add $f$ to $e_1$ and vice versa
                    set $n_1 =$ other adjacent node of $e_1$
                    set $e_1 =$ next clockwise edge of $n_1$
                } while $n_1 \neq n_0$ or $e_1 \neq e_0$
                ...
        }
}
```

Kutschmar's code works only for biconnected graphs, but not for all simple connected graphs. His code goes clockwise around the adjacent faces until the start node is reached again. However, a tree as in Figure 3.6 requires more time passing a node, such that the break condition of the do-while loop does not work for this case and has to be extended, which is a contribution of this thesis.

Even if the edges are passed until the current edge is equal to the starting edge, it would lead to the same result. For instance, let e_1 be the start edge in the tree example and the algorithm passes the other edges in clockwise direction, then the next edge again would be e_1 and the loop would break.

To solve this problem the break condition of the do-while loop is adjusted in a way that the start node has to be equal to the current node and the start edge has to be equal to the current edge to break the loop (see Listing 3.2). The resulting walk-through around a face passes the edges until the start edge and start node are reached. Now trees are processed correctly.

Chapter 4

Orthogonalization

As mentioned before, the orthogonalization works on a planar embedding produced by the planarization step and changes the shape of the graph. The result of that step is an orthogonal representation including angle- and bend-data. This chapter bases on the approach of Tamassia [Di +99].

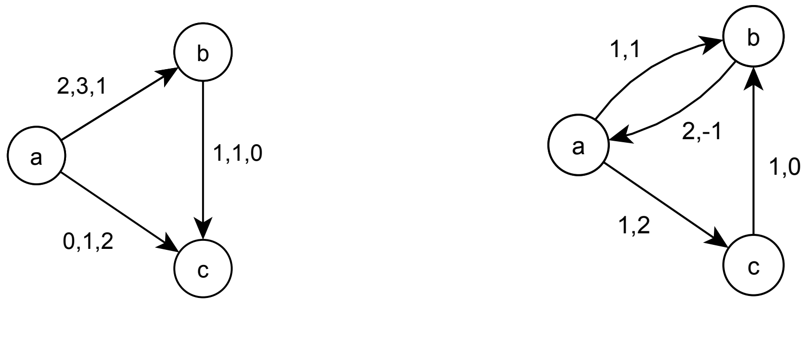
For a start, some preliminaries are presented to extended the definitions of the flow network context. Furthermore, the orthogonalization phase, bases on Tamassia's consideration, is discussed. In that context the technique for minimizing the number of bends in the flow model is considered, and the final mapping of that result on the original graphs elements is briefly described. Additionally, the intermediate processors `FaceDetermination` and the `ExternalFaceProcessor` are discussed.

Table 4.1. The architecture of the orthogonalization.

Typ	Name	Description
Preprocessing	`FaceDetermination`	Determining the faces of a planar embedding.
	`ExternalFaceProcessor`	Choice of an external face.
Phase	`TamassiaOrthogonalization`	Orthogonalization of TSM algorithm.
Postprocessing	-	

4.1 Preliminaries

In Section 2.1.1 the flow network was introduced. In that context the MCF problem is defined, which has to be solved to set angles to vertices and bends to edges. The established algorithms to solve the MCF problem take the so called *residual network* into account.



a. Example of an flow network with arc labeling x_{vw}, $cap(v,w)$, $cost(v,w)$.

b. Residual network of Figure 4.1a with arc labeling r_{vw} and $cost(w,v)$.

Figure 4.1. Flow network example with its corresponding residual network.

Definition 4.1 (Residual Network)**.** In the *residual network* $G(x)$ with flow x each arc $(v,w) \in A$ is replaced by two arcs (v,w) and (w,v) with the following properties:

▻ Arc (v,w) has costs c_{vw} and a residual capacity $r_{vw} = c_{vw} - x_{vw}$.

▻ Arc (w,v) has costs $c_{wv} = -c_{vw}$ and a residual capacity $r_{wv} = x_{vw}$.

Furthermore, arcs with a residual capacity of zero are removed so that the residual network consists only of arcs with positive residual capacity.

Figure 4.1b shows the residual network of the flow network of Figure 4.1a. Arc (a,b) is split into two edges (a,b) and (b,a). Two of three possible units of flow run from node a to node b. Thus, the residual capacity is 1 and the edge costs of arc (a,b) remain consistent. Arc (b,a) of the residual network has a residual capacity of the original flow naming 2 and negative costs. Both edges are accepted in the residual network. In addition, arc (a,c) of the flow network has only the edge (a,c) in the residual network since the flow is zero. Furthermore, arc (b,c) of the flow network has only a back-edge in the residual network because the residual capacity is zero.

Definition 4.2 (Reduced Cost)**.** For each node $i \in N$ the number π_i is the *potential* of node i. The *reduced cost* c^{π}_{ij} of an edge $(i,j) \in E$ is defined as $c^{\pi}_{ij} = c_{ij} + \pi_i - \pi_j$.

Theorem 4.3 (Reduced Cost Optimality)**.** *A feasible flow x is optimal if and only if there is a potential π such that all edges $(i,j) \in G(x)$ yield: $c^{\pi}_{ij} \geqslant 0$.*

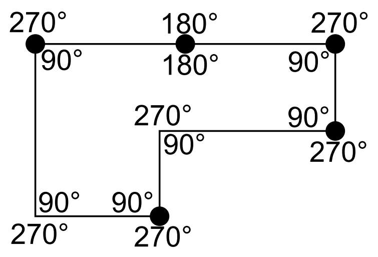


Figure 4.2. Vertex- and bend-angles of a planar orthogonal drawing.

4.2 Tamassia's Approach

Roberto Tamassia et al. [Di +99] considered a technique where the construction of a flow network is used to minimize the number of bends and to determine the relative position of the incident edges around each vertex by angles. This approach allows a maximal node degree of four, since it assumes that edges adjacent to a node can only lie on the right, top, left, or bottom of a node. In the following, some essentials are introduced to understand this approach.

4.2.1 Angles and the Orthogonal Representation

Definition 4.4 (Angle)**.** A *vertex-angle* is defined by the counter-clockwise angle between two consecutive edges adjacent to a vertex. The angle which is formed by a bend is called *bend-angle*.

Theorem 4.5. *The sum of the measures of the vertex-angles around a vertex in an orthogonal plane is equal to* $360°$*.*

Theorem 4.6. *The sum of the measures of the vertex- and bend-angles inside an internal face f of a planar orthogonal drawing is* $(2n-4)90°$ *and* $(2n+4)90°$ *for the external face respectively, where n is the number of edges.*

The theorem can be understood easily with Figure 4.2. Each arbitrary vertex has an angle measure of exactly $4 \cdot 90°$ around it. An orthogonal representation H for a graph G is an extension of the planar representation and describes, in addition to the topology, the shape of a drawing for G by specifying the angles inside a face and the bends, including bend-angles of the edges. Each edge (v, w) gets a list of angles that form the bends in the final drawing. In addition, each vertex v is equipped with a list of adjacent edge and angle pairs. Furthermore, an orthogonal

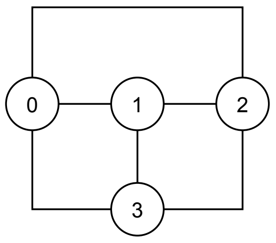


Figure 4.3. Orthogonal drawing with four vertices and four bends.

representation can be regarded as an equivalence class for drawings with similar shapes.

For a better orientation the counter-clockwise angle directions are mapped on identifiers:

▷ A 90° angle is called *left*.

▷ A 180° angle is called *straight*.

▷ A 270° angle is called *right*.

▷ A 360° angle is called *full*.

A drawing of the orthogonal representation is presented in Figure 4.3, and the angle-data of that representation is stated as follows:

Table 4.2. Angle-data of Figure 4.3.

Node	Angles (edge; angle direction)
0	((0,2); straight), ((0,3); left), ((0,1); left)
1	((0,1); left), ((1,3); left), ((1,2); straight)
2	((1,2); left), ((2,3); straight), ((0,2); left)
3	((0,3); straight), ((2,3); left), ((1,3); left)

For example, let us consider node 0. The edge $(0,2)$ has as counter-clockwise successor edge $(0,3)$. The angle between these edges is straight (180°). Respectively, the successor of edge $(0,3)$ is edge $(0,1)$ with an angle of (90°) positioned at its left hand side. Meaning if one looks along edge $(0,3)$ from node 0 to node 3, the edge $(0,1)$ is the counter-clockwise next angle and lies on the left hand. The same yields for the last entry $((0,1)$; left) of the orthogonal representation.

In the following, the transportation problem is described as an illustration in the context of network flow minimization.

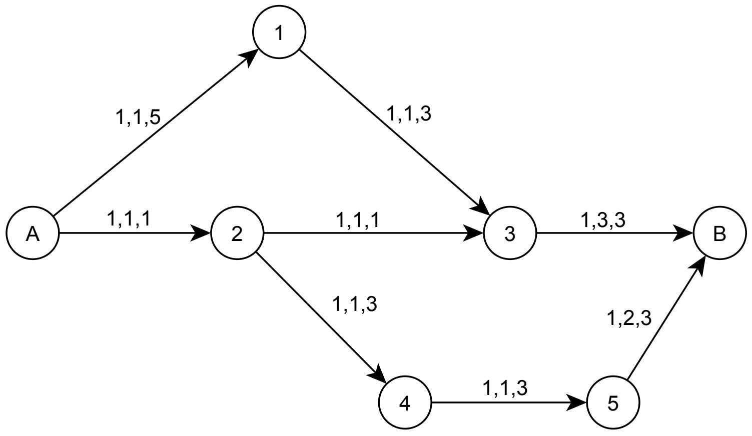


Figure 4.4. Drawing of a directed graph with source A and target B illustrating the road plan of a transportation problem.

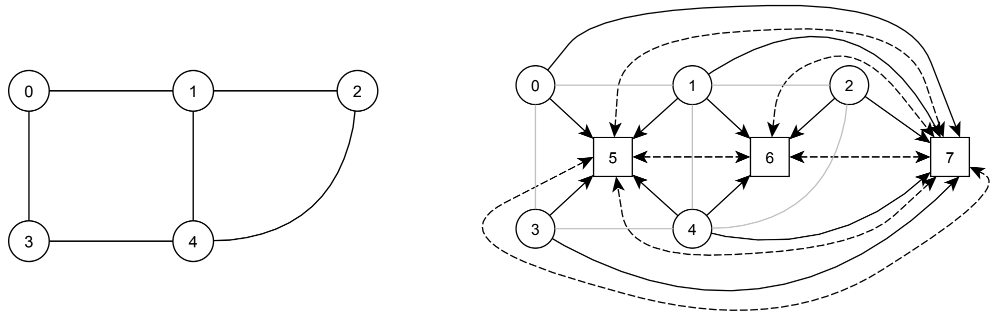
Transportation Problem

The transportation problem considers the problem of finding an optimal distribution plan for a single commodity. Assuming there is a company which produces commodity in a factory at location A that needs to be transported to a warehouse at location B. An one way road plan is given in Figure 4.4, and a maximal workload of the different roads is the goal. A single road consists of the tuple $(x_{vw}, c(x), u_{vw})$ and has a lower bound of zero.

Instead of determining the maximal workload, the goal of this part of Tamassia's algorithm is to calculate the minimum workload of the different roads under the condition that each road has to be passed at least one time. This leads to the next subsection.

4.2.2 Orthogonalization Network Flow

In the context of the orthogonalization, arc properties are only integer variables, and the flow function x produces only non-negative values. A *network flow model* for orthogonalization is a directed graph such that the nodes of that model are the vertices and faces of the original graph. Angles can be seen as commodity of the transportation problem that are supplied by the vertices and transported between the faces. The roads of the transportation problem illustrate the arcs of the network. Listing 4.1 shows a brief overview of Tamassia's orthogonalization algorithm which begins with the construction of the flow network. Afterwards, a minimum cost



a. Planar embedding. **b.** Flow network of 4.5a.

Figure 4.5. Transformation of a planar embedding to a flow network.

flow is computed for the constructed network. The used minimum cost flow solver in our implementation is the Successive Shortest Path (SSP) algorithm [AMO93]. Finally, the calculated minimum cost flow is used to determine the angle- and bend-data to generate an orthogonal representation.

Listing 4.1. Steps of Tamassia's approach.

```
Construct the flow network $N$;
Compute flow $x$ of minimum cost for $\mathfrak{N}$;
Compute orthogonal representation with respect to $x$;
```

Constructing the Flow Network

The result of the edge insertion phase 3.2 is a plane graph. To convert a graph into a flow network the following steps have to be done.

- ▻ Creating a node v in $\mathfrak{N}$ for every vertex and every face of the original graph G.
- ▻ Connecting each vertex node v of N with the face node w of N, if the face is adjacent with the vertex in the original graph G.
- ▻ Connecting each face node v of N with another face node w of N for every edge in G that is adjacent to both faces.

The example of Figure 4.5b shows the creation of the network flow model of a planar embedding. The network flow model consists of 8 nodes in which the round ones are the vertex nodes and the rectangular ones are face nodes. Black solid lines describe the node adjacent face arcs and the black dashed lines are the arcs between adjacent faces. The grey connections correspond to the edges of the original planar embedding 4.5a and illustrate the crossings of the face arcs.

The idea behind the flow network model is as follows. The flow in an arc (v, f) represents the measure of an angle formed at vertex v inside face f bounded by the lower bound $b_{vf} = \pi/2$ and the capacity $u_{vf} = 2\pi$. The cost is zero since such an angle is at a vertex and not at a bend. The flow in arc (f, g) represents the number of bends with $\pi/2$ angle in face f along an arc between faces f and g. Each unit of cost along such an arc is equal to a bend-point.

Minimum Cost Flow

The next step is to compute the flow x of minimum cost for $\mathfrak{N}$ (Listing 4.1). In the following, the SSP algorithm is discussed. It is a popular minimum cost flow solver that utilizes the residual network to find the shortest path.

This algorithm is based on the repeated usage of a shortest path algorithm. Starting with an optimal pseudo flow that ensures the condition of capacity and non-negativity but violates the condition of mass balancing, the algorithm tries to reach the mass balancing without breaking the conditions of optimality and capacity. This SSP algorithm can be viewed as a generalization of the Ford–Fulkerson algorithm [EK72].

Listing 4.2. Processing of the SSP minimizer.

```
Procedure($\mathfrak{n}$ : Network){
        Transform $\mathfrak{n}$ by adding source and sink
        Set initial flow $x = 0$
        Establish potentials $\pi$ with Bellman & Ford's algorithm.
        Reduce cost of $\pi$
        while($G(x)$ contains a path from s to t) {
            Find shortest path P from s to t with Dijkstra's algorithm.
            Reduce cost of $\pi$
            Get minimal capacity along P
            Increase current flow x along P with minimal capacity
            Update the residual network G(x)
        }
}
```

The functionality of the algorithm is described by Listing 4.2. In the first step source and target nodes are added to the network, such that each start node of the network is now successor of the source node and each final node of the

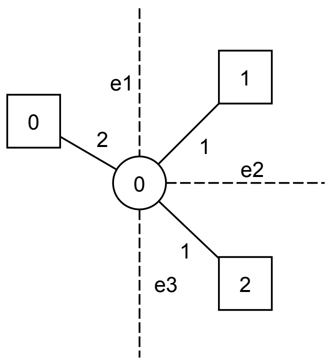


Figure 4.6. Mapping of the angles from network flow to original graph.

network is now ancestor of the target node. On the one hand, Bellman and Ford's shortest path finder calculates the node potentials, which defines the distance from a node to the source on the shortest path. On the other hand, Bellman and Ford's algorithm is able to detect negative cycles in the network and make all edge costs non-negative. Now, Dijkstra's algorithm[1] with better performance can be used to find shortest paths. In order to keep the edge costs non-negative on each iteration the node potentials are updated and the edge costs are reduced like in Definition 4.2. Additionally, the residual network is updated, and if that network does not contain a path from source to target the algorithm is finished.

Bellman and Ford's algorithm needs $O(nm)$ time, where n is the number of arcs and m is the number of nodes. The number of iterations is at most nU, where U is the largest supply, and Dijkstra's shortest path finder needs $O(n^2)$. Summing up $O(n^3U)$ complexity is received for the SSP algorithm.

4.2.3 Compute Orthogonal Representation

The minimum cost flow can be computed by the methods introduced in the preceding subsection. Now the question is how to get an orthogonal representation from that minimum cost flow solution. The angles and the bends are derived

[1]http://en.wikipedia.org/wiki/Dijkstras_algorithm

from the minimum flow, separately. To add edge angles to the vertices one has to consider the node arcs of the flow network.

Figure 4.6 illustrates the mapping of the flow of a node arc to its corresponding angle. The round node is the original vertex that has node arcs to the angular adjacent face nodes. The MCF solver calculates a flow value of 2 for arc $(0,0)$, 1 for arc $(0,1)$, and 1 for arc $(0,2)$. The flow is directly mapped to the angle between the adjacent face and a flow value, where

- ▻ 0 forms a left angle,
- ▻ 1 forms a straight angle,
- ▻ 2 forms a right angle, and
- ▻ 3 forms a full angle.

In the presented example the angle between $e1$ and $e3$ is a straight one, the one between $e3$ and $e2$ is a left angle, and the one between $e2$ and $e1$ is left as well.

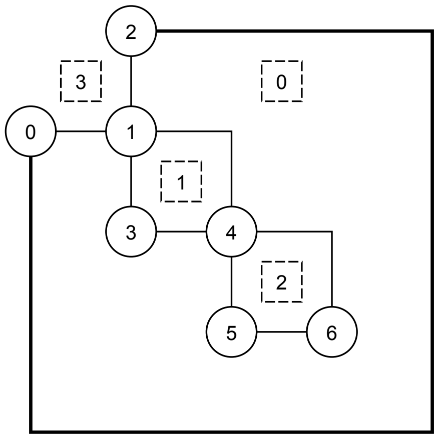
Bends are the units of the flow of face arcs. Hence, to add the bends to the original graph edges means to iterate over all face arcs of the network and furthermore adding the bends to the corresponding original edges in the original graph. Depending on the left and right face of an edge the bends are added with left or right angle.

4.3 Preprocessing

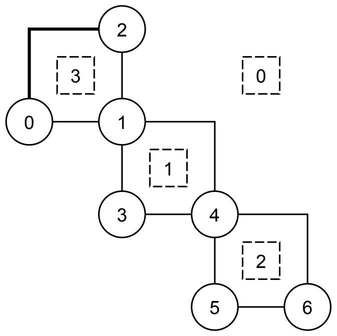
The orthogonalization phase uses the faces of the given planar embedding to create the flow network. After the planarization step, there is a possibility that these faces are unknown, especially other preprocessors are able to insert dummies that lead to another faces environment. Hence, an own intermediate processor is implemented for the determination of the faces, denoted as `FaceProcessor`. That algorithm is the same as the one in Section 3.2.1.

4.3.1 The External Face

An extension of the face calculation is the choice of a maximal external face. A face is maximal if its number of adjacent edges is higher or equal to the number of adjacent edges of any other face. Every face of the graph embedding can be treated as external, by simply turning inside out. Additionally, this ensures the embedding



a. Embedding with an external face with few adjacent edges.



b. Embedding with an external face with most adjacent edges.

Figure 4.7. Area discrepancy of the same embedding with different external faces.

since the counter-clockwise order of edges around every node of Figure 4.7a is equal to the order of edges around every node of Figure 4.7b.

Even if the embedding of the presented example on the left side is equal to the one on the right side, the external faces are different. Using a non-maximal external face (Figure 4.7a) results in a representation with the undesired edge $(0, 2)$ marked bold since the algorithm has to lay this edge around the remaining elements of the graph drawing. Figure 4.7b shows a drawing of the same embedding with a maximal external face. The length of the edge $(0, 2)$ is less than the length of the same edge in Figure 4.7a whereas all other edges have same edge lengths. Reflecting the aesthetics criteria of Section 2.1.3 the minimal total edge length of the graph drawing should be ensured to let the used area be minimal. Hence, it is meaningful to use a maximal face as external.

This feature is treated as an intermediate processor because there are extended features of the TSM approach that do not need to calculate a new external face, like Interactive Planarization which needs to keep the original drawing, and thus the original embedding and original faces.

Chapter 5

Compaction

This is the last part of the TSM algorithm and considers the problem of compacting the representation calculated by the orthogonalization. The task here is to assign minimum lengths to the segments of the edges of the orthogonal representation with the condition that there are no edge crossings and no node overlaps. Tamassia investigated a technique that uses a flow network to minimize the edge lengths [Di +99].

Table 5.1. The architecture of the compaction.

Type	Name	Description
Preprocessing	`BendPointProcessor`	Creating dummy nodes for bend-points.
	`RectShapeProcessor`	Bringing faces in rectangular shape.
	`FaceSidesProcessor`	Associating edges with face sides.
Phase	`TidyRectangleCompaction`	Doing the compaction of TSM algorithm.
Postprocessing	`GridDrawingProcessor`	Mapping the graph elements on a grid.
	`RectShapeRemover`	Removal of dummy nodes and dummy edges that ensure rectangular faces.
	`BendDummyRemover`	Removal of bend-point dummy nodes.
	`PlanarDummyRemover`	Removal of planar dummy nodes.

Table 5.1 shows the different steps of the compaction. This chapter is structured as follows. First, the algorithm for compacting orthogonal representations and its embedding in the architecture of KLay Planar are considered. In that context, the construction of flow networks and the used algorithm to minimize them are discussed. Furthermore, the preprocessing intermediate processors for handling more general graphs and its implementation in KLay are considered. Finally, the postprocessing steps are discussed, in which the intermediate processors for grid drawing and dummy removal are presented.

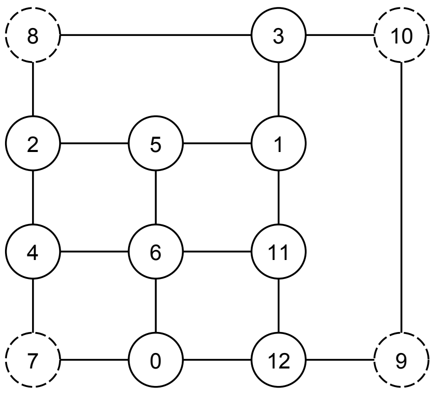


Figure 5.1. An example of an orthogonal representation with rectangular faces.

5.1 Tidy Rectangular Compaction

The problem of compacting general orthogonal representations with respect to minimum edge-length is NP-hard. Thus, Tamassia defined some restrictions to make such a computation possible. The compaction by Tamassia requires orthogonal representations that contain no bends as input. Furthermore, each face of the input has to have a rectangular shape.

Figure 5.1 shows an example of such an orthogonal representation. The dashed nodes represent bend-points and the solid nodes are previously added nodes of the graph structure. The orthogonal representation consists of at most four bend-points that are on the external face, since each internal face is in rectangular shape. An internal face can only be rectangular if it consists of four end nodes, where either all of them have left angles or all have right angles. Additionally, between these bend nodes, nodes with straight angles are allowed.

Listing 5.1 illustrates the process of the algorithm. The input of that algorithm is an orthogonal representation with n vertices with a maximum degree of four, with no bends and faces in rectangular shape. The output is a planar orthogonal grid drawing with minimum height, width, area, and total edge length with regard to the derived rectangular form. The edge lengths are calculated by solving a flow network, once for the horizontal segments and once for the vertical segments.

Listing 5.1. Tidy Rectangular Compaction.

```
Procedure($g$ : Graph){
    set solver = new simple flow solver;

    $N_{hor}$ = create horizontal segments flow network;
    calc flow of $N_{hor}$ with simple flow solver;
    map flow of $N_{hor}$ to lengths of edges of $g$;

    $N_{ver}$ = create vertical segments flow network;
    calc flow of $N_{ver}$ with simple flow solver;
    map flow of $N_{ver}$ to edge lengths of edges of $g$;
}
```

Then, the total flow of the networks is minimized by a simple flow solver. Afterwards, the flows of the networks are mapped to the length of the edges of the original graph. Finally, every edge segment of the graph has a relative edge length and can be drawn on a grid which is done in a postprocessing intermediate processor.

In the following, the described steps are discussed more detailed.

5.1.1 Again the Flow Network

A flow network is used to calculate minimal edge lengths just as it is done with the calculation of bend-points in the orthogonalization. In the following, the flow network for the horizontal segments is denoted N_{hor}, and the network for the vertical segments is denoted N_{ver}, respectively.

Network N_{hor} consists of nodes which represent the internal faces of the original graph. Additionally, N_{hor} has two nodes on the external face denoted s and t, representing the lower and upper region of the external face. N_{hor} has an arc for every adjacent pair of faces f and g, meaning they share a horizontal segment e.

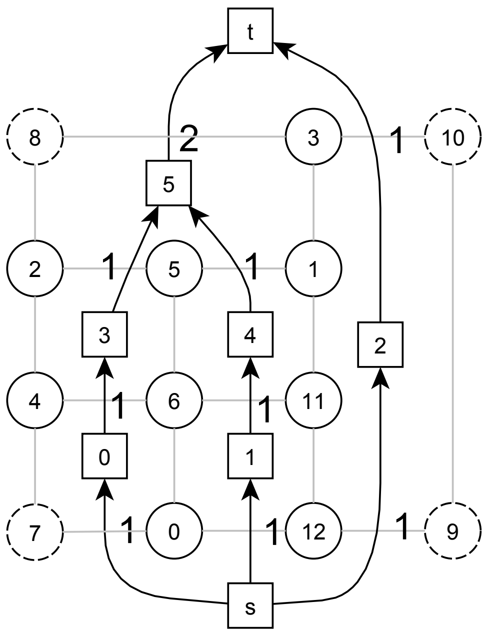
The arcs of the flow network in the context of the compaction have the following properties:

1. A *lower bound* $b_{vw} = 1$.

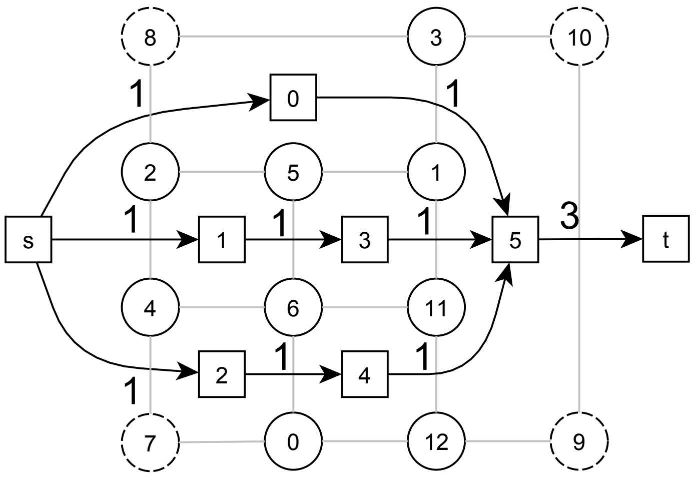
2. A *capacity* $u_{vw} = \infty$.

3. A *cost* $c_{vw} = 1$.

The flow in arc (f, g) represents the length of segment e.



a. Illustration of a minimum cost flow for the flow network N_{hor}.



b. Illustration of a minimum cost flow for the flow network N_{ver}.

Figure 5.2. Example for minimum cost flow in the context of the compaction.

Figure 5.2a and Figure 5.2b shows N_{hor} and N_{ver} with a flow of minimum cost for the example of Figure 5.1. The round vertices with grey edges represent the orthogonal representation and the rectangular nodes with source s and sink t are associated with the nodes of the flow network model. Its nodes are connected by directed edges, drawn black. Since for every arc the lower bound is $b_{vw} = 1$, every edge of the original network has at least a relative length of 1. As mentioned in Section 2.1.1, the incoming flow of a node has to be equal to the outgoing flow in general flow networks. In order to ensure this condition the flows of the arcs are increased iteratively.

Results as in Figure 5.2a and Figure 5.2b are computed. For example, the sum of the incoming flow of arc $(5, t)$ of Figure 5.2a is 2, so the sum of its outgoing flow has to be 2, too. And indeed, in the original graph the required length of the edge $(8, 3)$ is 2.

The following properties of the networks N_{hor} and N_{ver} are immediate. Both networks are planar and acyclic, with unique source nodes and unique sink nodes on their external faces. Thus, N_{hor} and N_{ver} are planar st-graphs.

The technique for solving such minimum cost flow problems is considered more detailed in the following. This method was developed and implemented in this thesis.

5.1.2 Simple Flow Solving

The SSP algorithm of Section 4.2.2 which solves the minimum cost flow problem during orthogonalization, is not useful in this context. The minimum cost flow problem here has constant arc costs of 1 and arc lower bounds equal to 1. Additionally, an initial flow of 1 is assumed. These constraints make the problem much easier. Hence, a flow solver is presented that solves the problem more efficiently.

Each non-source and non-sink node is a transport node. With the definition of feasible flow, one knows that the sum of the incoming flow and the sum of the outgoing flow have to be equal in order for the total flow to be feasible (see mass balancing Section 4.2.2).

The basic parts of the algorithm are presented in Listing 5.2. First, each arc gets a flow of 1 corresponding to the smallest relative edge length. Afterwards, a Breadth First Search (BFS)[1] from source to sink node is processed and each found node is added to a list of nodes, such that the list is ordered by the BFS. Then for each non-source and non-sink node of the list it is checked whether the sum of

[1]http://en.wikipedia.org/wiki/Breadth-first_search

incoming flows is equal to the sum of outgoing flows. If this is the case there is nothing to do since the flow of that arc is feasible. If the sum of outgoing flows is greater than the sum of incoming flows (gap is higher than 0), the sum of the incoming flow has to be increased to reach the mass balancing.

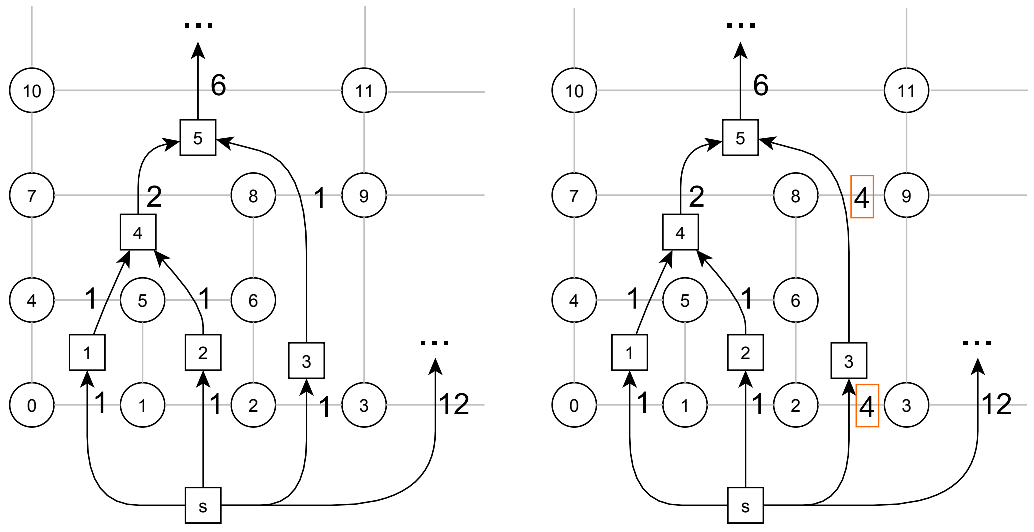
Increasing the flow of an arc (v, w) means that on the one hand the incoming flow of node w is increased but for the other incident node v the outgoing flow is increased. The same procedure is done for an incoming arc of node v, in order to ensure the mass balancing of v. This process is repeated until the source node is reached.

Listing 5.2. Simple Flow Solving Algorithm.

```
Procedure(n : Flow Network){
  set flow of arcs of n to 1
  bfs_struct = perform breadth first search on n
  for(node : bfs structure){
        if(node == source or node == sink){
          continue
        }
    gap = outgoing node flows − incoming node flows
    if (gap > 0){
      path = calc shortest path to source
      for(arc : path){
        add gap to arc flow.
      }
    } else if (gap < 0){
    path = calc shortest path to sink
    for(arc : path){
      add gap to arc flow.
    }
  }
}
```

A node can be incident to various incoming arcs and outgoing arcs, and the question is which arc flow should be increased to keep the flow minimal.

An example for that question is illustrated in Figure 5.3 for the horizontal edge lengths. The sum of the incoming flow of node 5 is equal to 3, and the sum of its outgoing flow is equal to 6 (see Figure 5.3a). The gap of 3 has to be added to one of the incoming arcs in order to let the node 5 satisfy the balancing condition. The solution is to take the incoming arc with the shortest path to the source. Then, the flow of the arcs with the lowest flows are increased, and the total edge length of the given orthogonal representation stays minimal. In the presented example the



a. Non-feasible solution.

b. Feasible solution by increasing the flows along the shortest path.

Figure 5.3. Flow increasing example.

path $(5, 3, s)$ is taken and each flow of arcs between that nodes are increased by 3. The result is shown in Figure 5.3b.

The same has to be done for the other direction if the sum of outgoing flows of a node is smaller than the sum of incoming flows. Then the shortest path to the sink node is calculated and the flow of each arc is adjusted.

The complexity of that simple flow solver is stated as follows. Adding initial flow to arcs and performing the BFS is only done once and so negligible. The used implementation of Dijkstra's algorithm needs $O(n^2)$ in rare cases. This is done for every node with gap not equal to 0. Assuming the rare case that all non-source and non-sink nodes have such a gap, Dijkstra's algorithm is processed for $n-2$ nodes. Increasing the nodes arcs is only linear and is negligible, too. Summing up a complexity of $O((n-2)\dot{(}n^2))$ is received for this algorithm.

In practical applications the algorithm is much faster, since the gap of incoming and outgoing flows during the processing is often 0. Furthermore, Dijkstra's algorithm is only performed for a subgraph $\mathfrak{N}'$ of $\mathfrak{N}$ such that the number of nodes of $\mathfrak{N}'$ is n' which is much smaller than n. For example, consider the minimum cost flow of the network in Figure 5.2a. It only needs a flow arc increase at node 5, and the path processed by Dijkstra's algorithm to the sink node is also short such that the algorithm works in linear time here. All in all, the complexity varies depending on the given orthogonal representation.

5.2 Preprocessing

The compaction phase needs a representation with edges without bend-points, because every edge segment is assigned its own edge length. Furthermore, faces in rectangular shape are required. Thus, the preprocessors `BendDummyProcessor` and `RectShapeProcessor` are introduced. Finally, a third processor is presented which calculates the sides of each face, which makes life easier in many processes.

5.2.1 Insertion of Bend-Point Dummies

The task of the `BendDummyProcessor` is to exchange the bend-points of the edges with dummy vertices.

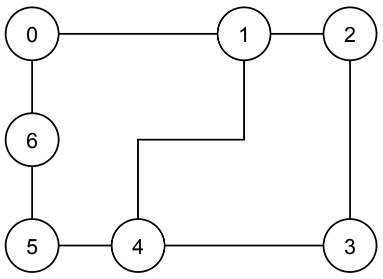
The idea behind that processor is presented in the following (see Listing 5.3). The process iterates over all bend-points of the edges of g, and each edge $e = (v, w)$ is subdivided with a new dummy node d for its bend-point. This includes the creation of d, connecting d with the source of e by updating e to (v, d), and the creation of a new dummy edge $e' = (d, w)$ that connects the d with the target of the original edge e.

Listing 5.3. Snippet of the `BendDummyProcessor`

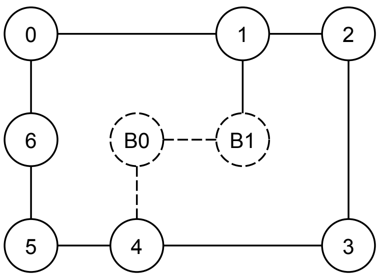
```
process(g : Graph){
   for(e : edges of the graph){
      for(bp : bends-points of e){
         subdivide e with new dummy node d;
         set angle-data of d;
      }
   }
}
```

Furthermore, angle-data of each new dummy node has to be set to ensure a correct orthogonal representation.

The following observation is immediate. If a bend-point forms a right turn, the edge combination of e and e' forms counter-clockwise (left) angle of 90°. The reverse combination e' and e forms (right) angle of 270°. Respectively, for a bend-point that form a left turn, it is processed vice versa.



a. Orthogonal representation example.



b. Example of Figure 5.4a with bend-point dummy nodes.

Figure 5.4. Example for bend-point dummy node insertion.

An example for the described algorithm is illustrated in Figure 5.4. The dummy nodes and dummy edges are marked with dashed lines. Edge $(1, 4)$ is subdivided. First, B_0 is inserted as well as the new dummy edge $(B_0, 4)$, and e is updated to $(1, B_0)$. Afterwards, B_1 is inserted. In that process, a new dummy edge (B_1, B_0) is added, and e is updated again to $(1, B_1)$. The resulting representation, seen in Figure 5.4b, contains no more edge bends.

5.2.2 Making Faces Rectangular

In general, the faces of a orthogonal representations may not be in rectangular shape. Hence, an intermediate processor is added to the processing slot before the compaction phase, namely the `RectShapeProcessor`, which transforms the shape of the given faces into rectangular shapes. This is done by splitting non-rectangular faces by adding dummy vertices and dummy edges. The presented technique bases on the book of Di Battista et al. [Di +99].

A rectangular refinement of an orthogonal representation H is denoted H'. The algorithm for the construction of H' works as follows. Firstly, for each face f of H it is checked whether its shape is rectangular. If f is in rectangular shape there is nothing to do. Otherwise f has to be transformed and the following properties for every adjacent edge e have to be calculated:

▻ *next(e)*; Representing the following edge of e when traversing the boundary of f counter-clockwise.

▻ *corner(e)*; Representing the shared vertex of the edge e and next(e).

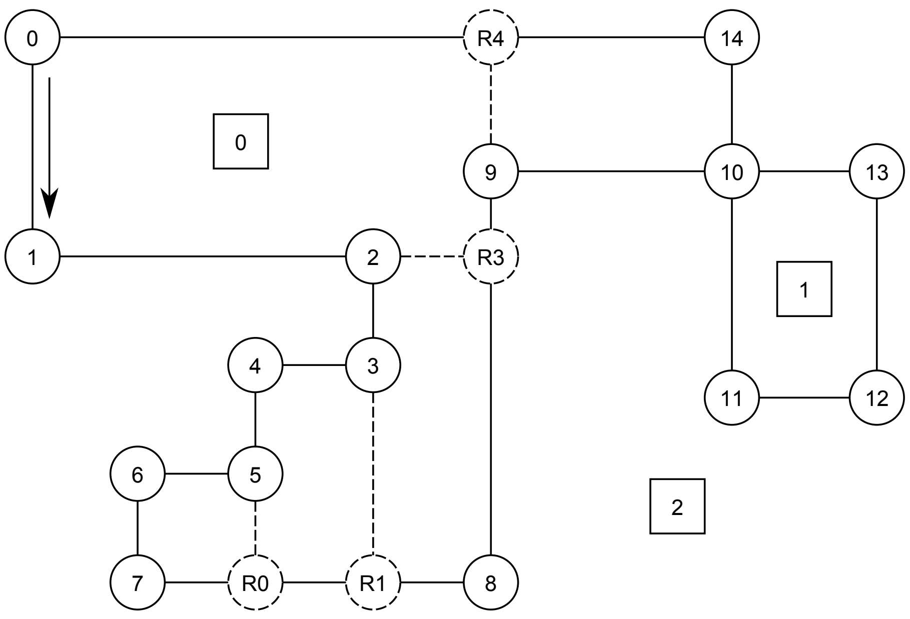


Figure 5.5. A more complex example of an orthogonal representation containing faces with non-rectangular shape.

⊳ *turn(e)*; A property that describes a direction change between e and $next(e)$ with the following conditions:

1. $turn(e) = +1$, if e and next(e) form a left turn.
2. $turn(e) = 0$, if e and next(e) form a straight line.
3. $turn(e) = -1$, if e and next(e) form a right turn.

⊳ *front(e)*; Representing an edge that is calculated by traversing f counter-clockwise such that the sum of the turn values for all edges between e and $front(e)$ is equal to 1. This includes the turn of e but excludes the turn of $front(e)$.

In the following, an example is presented in which the edge properties are computed. The the internal face 0 of Figure 5.5 is not in rectangular shape. Its edges are traversed in counter-clockwise direction, shown by the arrow on the left. Table 5.2 illustrates all adjacent edges of face 0 and their properties. For example, let us consider edge $(1,2)$. The next edge in counter-clockwise direction is edge $(2,3)$ with the corner 2. The edges $(1,2)$ and $(2,3)$ form a right turn, so its turn

Table 5.2. Edge properties of face 0 of the orthogonal representation shown in Figure 5.5.

Edge	Next	Corner	Turn	Front
(0,1)	(1,2)	1	+1	(1,2)
(1,2)	(2,3)	2	−1	(8,9)
(2,3)	(3,4)	3	−1	(7,8)
(3,4)	(4,5)	4	+1	(4,5)
(4,5)	(5,6)	5	−1	(7,8)
(5,6)	(6,7)	6	+1	(6,7)
(6,7)	(7,8)	7	+1	(7,8)
(7,8)	(8,9)	8	+1	(8,9)
(8,9)	(9,10)	9	−1	(0,14)
(9,10)	(10,14)	10	+1	(10,14)
(10,14)	(14,0)	14	+1	(14,0)
(14,0)	(0,1)	0	+1	(0,1)

is -1. The front of the edge $(1,2)$ is the edge $(8,9)$ since the sum of turns of the edges along the path $((1,2),(2,3),(3,4),(4,5),(5,6),(6,7),(7,8))$ is equal to 1.

For every edge of an internal face f the front is defined, since by Theorem 4.6 $\sum_{e \in E} turn(e) = 4$. Hence, traversing f with a start edge e with a turn of -1, 0 or 1 always leads to a sum of turns equal to 1, and thus $front(e)$ always exists.

Furthermore, only edges with a turn of -1 destroy the rectangular shape of an internal face. The faces are traversed counter-clockwise such that all left turns and straight passing edges do not destroy the rectangular face property. Thus, only edges with $turn(e) = -1$ are extended with a dummy edge.

After the described properties are calculated for every edge e of f, $front(e)$ for each edge with $turn(e) = -1$ is subdivided with a new dummy vertex r, and a dummy edge $e' = (corner(e), r)$ is inserted. It is important to ensure the correct angle-data in H'. Hence, H' is updated such that e and e' form a straight line. All involved elements are equipped with a property called `RectShapeDummy`. Elements with such a property are identified by the rectangular shape dummy removal processor, which removes them. The algorithm can only be used for internal faces, since adjacent edges of the external face might contain undefined fronts.

Face 0 of the example of Figure 5.5 is in rectangular shape after that algorithm is processed. In the following, the algorithm for the external face is presented.

The External Face

The refinement of the external face f_{ext} can be done with a variation of the above algorithm. The adjacent edges of f_{ext} are traversed in clockwise direction, and $\sum_{e \in E} turn(e) = -4$. Thus, traversing the adjacent edges of the external face until the start edge e is reached sometimes leads to $turn(e) = -4$. In this case the front of e is undefined.

The idea behind that algorithm is to add a new rectangle of dummy nodes and dummy edges around f_{ext}, which represents the new external face f'_{ext} that is in rectangular shape. Afterwards, f_{ext} is traversed in clockwise direction. Starting at an arbitrary edge, in Figure 5.6 the edge $(0, R4)$, every passed edge with a right turn is extended onto its front, just as in the algorithm for internal faces. Secondly, every edge with an undefined front is extended onto a side of the rectangle, with the result that f_{ext} is split into different faces that are in rectangular shape.

In the following, the implementations of the described algorithms are presented.

Implementation

The main parts of the `RectShapeProcessor` are illustrated in Listing 5.4. During the processing of the parts the faces are frequently traversed counter-clockwise or clockwise. For that purpose a face can be equipped with a property $(e, corner(e))$ that stores a start edge and the corner in counter-clockwise or clockwise direction.

Listing 5.4. Refinement of the faces to have a rectangular shape.

```
process(g : Graph){
   set ext_face = external face of g,
   determine clockwise direction of ext_face,
   if(ext_face is in not rectangular shape) {
      refine ext_face,
   }
   refine the internal faces of g,
}
```

The determination of the counter-clockwise direction can be easily calculated by traversing the adjacent edges of a face with an arbitrary start edge $e = (s, t)$. Then, the source s of e is taken as corner and all edges are traversed. If at the end of this process $\sum_{e \in E} turn(e) = 4$, the direction is counter-clockwise and the property is filled with the property (e, s). Otherwise it holds $\sum_{e \in E} turn(e) = -4$ and the property is filled with (e, t) to ensure the counter-clockwise direction. For

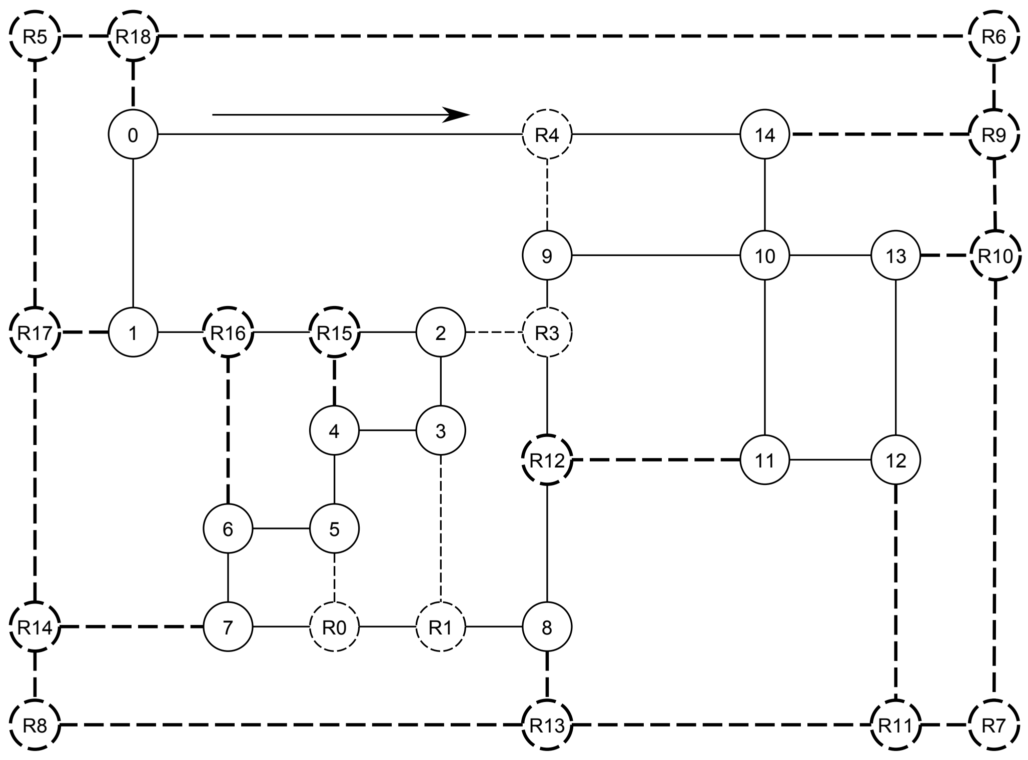


Figure 5.6. Refining the external face of the orthogonal representation of Figure 5.5.

the external face a clockwise traversing is desired, hence the direction property with corner *s* is adjusted such that *s* is exchanged with *t*, and vice versa.

In the next step of the implementation the external face is checked to be in rectangular shape. The technique for checking faces for rectangular shape can be used for external faces as well as for internal faces. The idea is to start at the previously calculated property edge and traverse the adjacent edges of a face. If a direction change happens, store the turn and go ahead. If another direction change is reached compare this one with the stored one. If they are not equal, the face is not in rectangular shape. But if they are, the remaining edges are passed an checked. If the start edge is reached, it is concluded that the face is in rectangular shape.

Refinement of the External Face If the external face is not in rectangular shape, its refinement bases on the algorithm mentioned before. Listing 5.5 gives an overview of the implementation of that refinement.

First, the edge's properties are determined. In this step, the adjacent edges of the external face are traversed clockwise and every edge *e* is equipped with the properties $next(e)$, $corner(e)$, and $turn(e)$. In the following steps, the stored properties $next(e)$ and $corner(e)$ are always used to get the next edge in clockwise direction by traversing the edges, such that e. g., the fronts can be determined easily with the technique described before.

Listing 5.5. Snippet of the refinement of the external face.

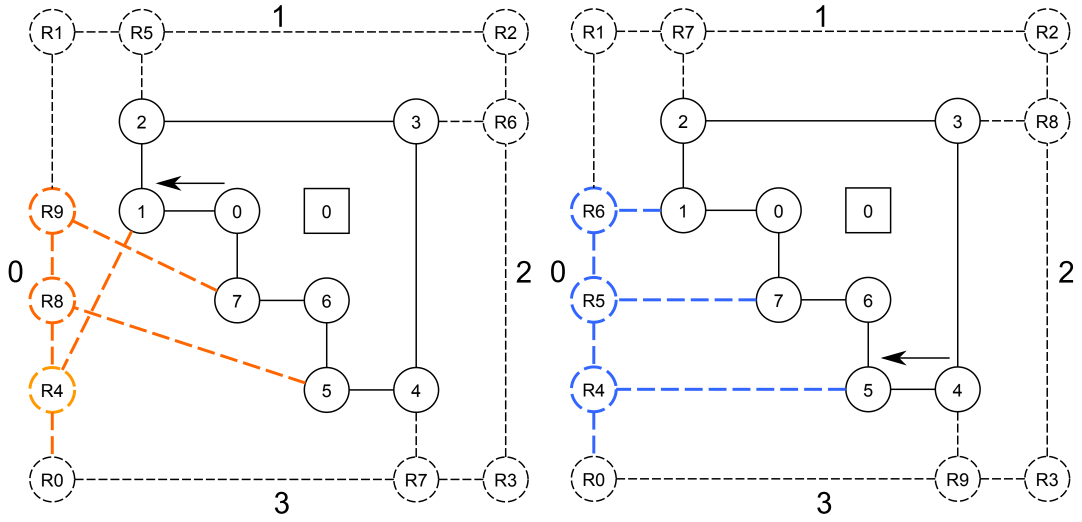
```
process(g : Graph){
   set ext_face = external face of g;
   determine edge's properties of ext_face;
   calculate edge's fronts of ext_face;

   add new rectangle to g;
   assign each edge with undefined front to a side;

   determine start edge of side left;
   connect original external face with rectangle;
   set rectangle as new external face;
}
```

Then, the rectangle skeleton is added to *g*, which consists of a dummy node for each bend and four edges representing their connections (see Figure 5.7a). Secondly the embedding of *g* is extended as well as the angle-data of the orthogonal representation. This is important for the following phases and intermediate processors which assume a correct orthogonal representation of *g*. After the rectangle is added each edge with undefined front is assigned to a side of the rectangle that describes which side is the front of the edge.

Connecting the edges of the original external face to the sides of the new external face consists of the following steps:

▷ A dummy vertex *r* is inserted on the front edge (v, w).

▷ For that process, the front is split into two edges connected with the dummy vertices (v, r) and(r, w). This is done by adjusting the front to (r, w) and inserting a new dummy edge (v, r).



a. Wrong edge extensions. **b.** Correct edge extensions.

Figure 5.7. Problems with the edge extensions.

▻ Then, the corner of the extending edge is connected to the dummy vertex r.

▻ All embeddings and angle-data of the involved nodes and edges are updated.

The edges of the old external face are passed clockwise, which ensures on the one hand the correct bridging of the faces, but on the other hand this does not solve the problem illustrated in Figure 5.7.

Note that the front is calculated once before bridging the faces. Figure 5.7a illustrates the latter. The sides of the rectangle are described by the number 0 for left, 1 for top, 2 for right, and 3 for bottom. The arrow shows the start position of the bridging. First, edge $(0, 1)$ is connected to the left side of the rectangle. Dummy node R_4 is added and the dummy edge (R_0, R_4) is inserted to split the front which is after the processing of the subdivision (R_4, R_1). R_4 is connected to the corner of $(0, 1)$ denoted 1 and the involved node, dummy embeddings and angle-data are updated, as described before. Then, face 0 is traversed clockwise and each edge with a front that equals one of the rectangle edges is extended. Everything works fine except for the extended dummy edges $(5, R_8)$ and edge $(7, R_9)$. They are connected to the wrong edges of the rectangle such that there are undesired edge crossings.

This results from the originally calculated fronts of these edges that are updated by the extension of edge $(0,1)$, and the later extension of edge $(4,5)$, such that $front(4,5) = (R_4, R_1)$ and $front(6,7) = (R_8, R_1)$. This can be solved with two variants.

1. Only 4 edges are attached to the rectangle with respect to the condition that only one edge is connected to one side. Then, there are no such problems. The resulting faces can be handled as internal faces and can be made rectangular later at the refinement of the internal faces.
2. A start side is determined, and the first edge in clockwise direction with a front that is equal to the start side is calculated. Afterwards, the face bridging is started with the calculated edge and the problem is solved.

Since all edge properties for the external face are calculated anyway, it would be wasteful to take the first option. Hence, the second one is taken and one has to think about its realization.

The left side of the rectangle is taken more or less arbitrarily as the desired start side. The idea behind that is to search for the first edge e_l with a front that is equal to the left side, by finding the last edge e_b with a front equals the bottom side.

The path, resulting by traversing the face from e_b to e_l in clockwise direction, may not include an edge with a front that is equal to the left or bottom side, in this case e_l is the sought-after start edge. In the example of Figure 5.7b the start edge $(4,5)$ is illustrated by the arrow. This is the first edge in clockwise direction with a front that is equal to the left side. Now the bridging can be done without any conflicts.

The described start edge is calculated and the final connection of the adjacent old external face edges to the rectangle can be done. The rectangle is set to be the new external face, and the procedure is finished.

Refinements of the Internal Faces After the shape of the external face is processed, all internal faces are handled. Each non-rectangular internal face is refined by the presented algorithm of Section 5.2.2. Listing 5.6 shows an overview of the implementation. Firstly, all non-rectangular faces are stored. In this process, the presented technique for checking whether a shape of a face is in rectangular shape is used. Edge properties are calculated like they were in the refinement of the external face, but here each face is traversed in counter-clockwise direction. This is done for an easier handling of cutvertices which are described in the following subsection.

Listing 5.6. Refinements of the internal faces.

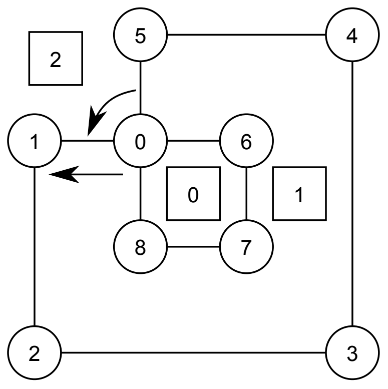
```
process(g : Graph){
   nr_faces = filter non-rectangular faces of g;
   for(f : nr_faces){
      determine edge's properties of f;
      calculate edge's fronts of f;
      determine start edge of a side of f;
      split f;
   }
}
```

As shown above all fronts of edges exist, so they can be calculated. Each face can be divided into face sides, just like the sides of the rectangle. The problem with the wrong order by connecting the edges with a side of a face exists at the refinement of an internal face as well. The same procedure as for the external face is done for getting a start edge. Then each face is split until there are no edges with a right turn. Each emerged face, by splitting the non-rectangular face, is rectangular, again, as can be seen in the example of Figure 5.5. Since each edge with a right turn is extended, all resulting faces have to be in rectangular shape.

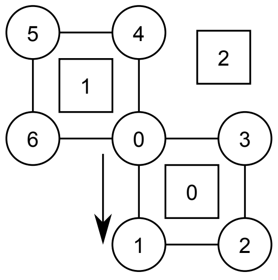
Handling Cutvertices

As mentioned in Section 2.1, a cutvertex is a node that divides a graph into different components if it is removed. The described methods for making the faces rectangular may yield wrong results for orthogonal representations with cutvertices, see Figure 5.8. When traversing a face, e. g., setting the edge properties in the algorithms above, the question is how to handle a cutvertex.

Let us consider the counter-clockwise traversed internal face 1 of Figure 5.8a. With the presented techniques the traversal would start with edge $(0,1)$, passing edges $(1,2)$, $(2,3)$,$(3,4)$, $(4,5)$ and would reach edge $(5,0)$. Here, the problem is how to decide which edge should be taken next, since the traverse is started counter-clockwise; logically speaking, the next edge in the walk-through would be edge $(0,1)$. This would cause an exclusion of the edges $(0,6)$, $(6,7)$, and $(8,0)$ from the walk-through. This would lead to an error.



a. Orthogonal representation with an internal face containing a cutvertex.



b. Orthogonal representation with an external face containing a cutvertex.

Figure 5.8. Refinement of the faces of an orthogonal representation with a cutvertex.

Hence, the next edge choice of a vertex, especially for a cutvertex, has to be handled differently from the global traversal direction. The solution to this is to take always the next edge in clockwise direction. This always leads to the longest, complete path. Since the global traversal starts with an edge in counter-clockwise direction, it is guaranteed that even after taking the next clockwise edge of a node the global traversal still keeps its counter-clockwise direction.

If the external face would do its traversal counter-clockwise a cutvertex would have to pass in counter-clockwise direction, otherwise not all edges are passed, see Figure 5.8b that has same problem as mentioned above. This would cause a lot of overhead, since anywhere a cutvertex is handled a distinction between external- and internal face would be needed. Thus, the external face is traversed clockwise and each cutvertex has to be passed in clockwise direction, too. Hence, each cutvertex can be treated equally everywhere in the implementation.

Handling Cutedges

Another problem is caused by cutedges. These are edges which would divide a graph in two components if they are removed. Figure 5.9 shows two typical examples for such edges. The algorithm above traverses the external face of an orthogonal representation in clockwise direction to determine the edge properties. In that process, a cutedge is passed twice in both direction. The properties of a twice passed edge are overridden such that only the last added properties are stored. The next steps of the algorithm would traverse with wrong edge paths. For example, Figure 5.9a has the traverse path $(0,1),(1,2),(2,3),(3,0),(0,4),(4,5),(5,6),(6,7),(7,4),(0,4)$. Table 5.3 illustrates the edge properties of the example of Figure 5.9a. The edge $(0,4)$ is passed twice such that the final properties of that edge would be the one of line 10.

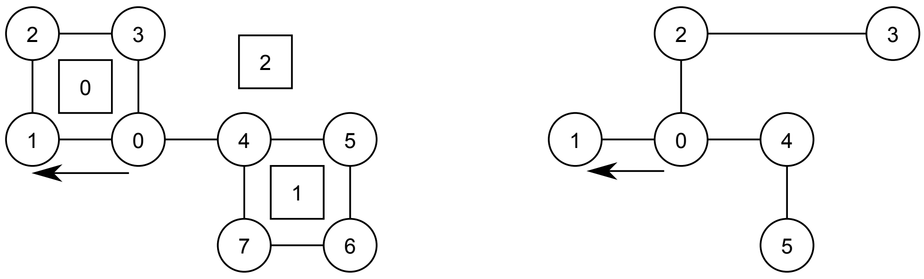
Table 5.3. Order of setting edge properties of the external face of Figure 5.9a.

Time counter	Edge	Next	Corner	Turn
1	(0,1)	(1,2)	1	−1
2	(1,2)	(2,3)	2	−1
3	(2,3)	(3,0)	3	−1
4	(3,0)	(0,4)	0	+1
5	(0,4)	(4,5)	4	0
6	(4,5)	(5,6)	5	−1
7	(5,6)	(6,7)	6	−1
8	(6,7)	(7,4)	7	−1
9	(7,4)	(0,4)	4	+1
10	(0,4)	(0,1)	0	0

Thus, every cutedge is equipped with two edge properties for every direction. An additional property $previous(e)$ is added to the edge properties in every direction. When traversing the adjacent edges of a face, it is checked whether the next edge e' of an edge e has the property $previous(e') = e$. If so, the direction is correct and the edge properties of this direction can be used. Otherwise the edge properties of the other direction of e' are taken.

Since the direction of an edge is important, the break condition of the edge traversed is adjusted. As mentioned in Section 3.2.1, the condition is extended to check whether the start node equals the current node and the start edge equals the current edge. Thus, the traversal ends at the start edge with the correct direction.

Trees A particular characteristic of trees are that all edges are cutedges, see Figure 5.9b. A second characteristic is that their leaves form full-angles. A full-angle represents an edge turn of -2 for the external face and an edge turn of 2 for internal faces. Thus, for the external face it is ensured that $\sum_{e \in E} turn(e) = -4$, and for internal faces it is ensured that $\sum_{e \in E} turn(e) = 4$, respectively.



a. Two graph components which are connected by a cutedge.

b. Tree representation.

Figure 5.9. Two examples for orthogonal representations with cutedges.

Table 5.4 shows a snippet of the edge properties of the external face of Figure 5.9b. Each edge (s, t) in the table exists with swapped source and target (t, s) to illustrate the different directions of the edge properties. Each of the edges $(0, 1), (2, 3), (4, 5)$ forms a full-angle and has a turn of -2, and summing up all edge turns result in -4.

Table 5.4. Result of setting edge properties of the external face of Figure 5.9b.

Time counter	Edge	Next	Corner	Turn
1	(0,1)	(1,0)	1	−2
2	(1,0)	(0,2)	0	+1
3	(0,2)	(2,3)	2	−1
4	(2,3)	(3,2)	3	−2
5	(3,2)	(2,0)	2	+1
6	(2,0)	(0,4)	0	+1
7	(0,4)	(4,5)	4	−1
8	(4,5)	(5,4)	5	−2
9	(5,4)	(4,0)	4	+1
10	(4,0)	(0,1)	0	0

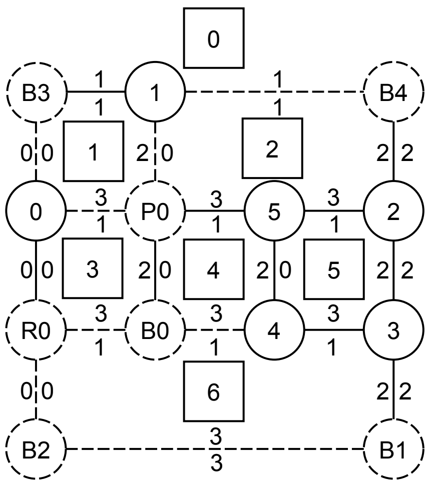


Figure 5.10. Representation with marked face sides.

5.2.3 Face Sides

The `FaceSideProcessor` is an intermediate processor that makes some processes easier on the implementation side. For example, during the compaction phase only the left and right face sides are interesting for the construction of the network N_{hor}, and the top and bottom sides for the construction of the network N_{ver}, respectively. These sides can be easily determined with the calculated faces sides.

Moreover, the structure of the graph does not change during the compaction, such that the calculated face sides can also be used in the grid drawing step. Firstly, the sides are used there to iterate over the edges and set their real coordinates, and secondly they are used to set the grid dimensions.

Figure 5.10 presents an example with marked face sides. Each side is described by a number, where

- ▻ 0 represents the left side,
- ▻ 1 represents the top side,
- ▻ 2 represents the right side, and
- ▻ 3 represents the bottom side.

Every edge e is adjacent to two faces f and g. A face f can contain e on the same side as g or on the opposite side, depending on the face. All internal faces that are adjacent by an edge to the external face, have the edge on the same side of the external face. All internal faces connected with another internal face have the edge on the opposite side of the other internal face.

Each face of the example is in rectangular shape. This is not a necessary condition to use that processor, but at this point of the TSM algorithm orthogonal representations in that shape are considered. Only the bend-points of each edge have to be exchanged by dummies for this algorithm, since each edge segment itself is mapped on a face side.

Listing 5.7. Face sides processor.

```
procedure($g$ : Graph){
    initialize $visited\_faces$ : Set of VisitEntry
    initialize $completed\_faces$ : Set of faces
    while($current\_face \neq$ null){
        initialize $face\_sides$ : Array of edge lists;
        add face edges to corresponding side;
        choose next $current\_face$;
    }
}
```

The algorithm to fill the face sides with edges is realized as follows. Two sets are created. $visited\_faces$ stores a $VisitEntry$ for each new found face, during the walk-around of the adjacent edges of a face f.

A $VisitEntry$ consists of the following attributes:

▷ The found new *face*.

▷ A *start edge* e_{se} which represents the current edge of the traversal.

▷ The incident *node* of e_{se} in clockwise direction of the found face. If f is an external face the direction stays the same, and if f is an internal face the node is the opposite node of e_{se}, to ensure the node in clockwise direction of the new face.

▷ An integer *side index* that stores the current face side of e_{se}. If f is an external face the side stays the same as above, and if f is an internal face, it yields $sideindex = (sideindex + 2)/4$, the opposite side.

The set $completed\_faces$ stores all finished faces.

Afterwards, the procedure iterates over all faces, and each of the four face sides is used as a list of edges. Starting on the left side (side index = 0), and the first edge in clockwise direction of the side is used as start edge to ensure the correct order in every side list.

Then, a traversal over all adjacent face edges is started. Each face that is found is added with the described properties to $visited\_faces$. A side is filled with edges until an edge and its next edge form a direction change. If the turn is a right turn the side index is increased, and if the turn is a left turn the side index is decreased. The example of Figure 5.10 contains only faces in rectangular shape. Thus, in that example there are only right turns with the result that the side index can only be increased. The traversal of adjacent face edges ends if the start edge and the start node are reached.

Finally, a non-completed face is chosen from the set of visited faces to be processed as next. The start edge and the start node of the stored properties are used to ensure the correct direction of the new face as well as the side index to ensure the correct side.

If $visited\_faces$ consists of only already completed faces the method ends and all faces are equipped with its face sides.

5.3 Postprocessing

After the preprocessing is done the orthogonal representation contains no bend-points, and secondly, all faces are in rectangular shape. The compaction phase sets the relative edge lengths. All that is left for the postprocessing to do is the mapping of all graph elements on a grid as well as the removal of the various inserted dummies. These issues are handled by intermediate processors, which are considered in the following.

5.3.1 Grid Drawing

A grid is introduced to give the elements of the orthogonal representation real coordinates. Nodes, bend-points and edge crossings get a unique position in the grid. Since edge crossings and bend-points are handled as nodes they are easy to integrate into the grid. The basic idea behind that approach is to go along the sides of each face and set each node that is not already set onto a position of the grid.

Then, the grid size is calculated easily, by taking the left and top sides of the external face. Summing up the relative lengths of their edges determines the height and width of the grid. For example, Figure 5.11 has a height of $1 + 1 + 1 = 3$ and a width of $1 + 2 = 3$.

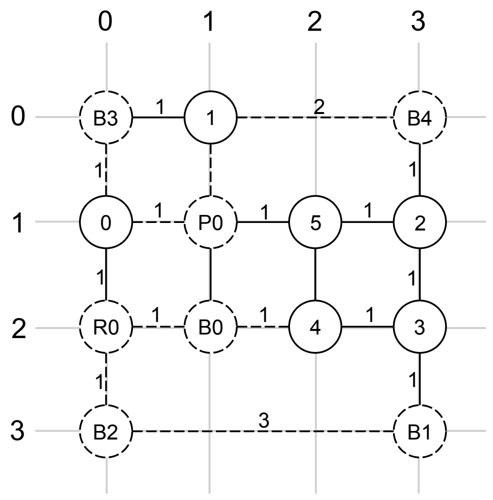


Figure 5.11. Grid drawing of $K_{3,3}$ with marked edge lengths.

Afterwards, the process of filling the grid is done. Listing 5.8 illustrates that method. The external face is taken as start face. To find a correct start position into the grid, the leftmost, lowermost node is used. That is the only node incident to an edge on the left side ($side[0]$) and incident to an edge on the bottom side ($side[3]$). The set $visited\_faces$ stores every new found face of the graph during an adjacent face edges traversal. If a traversal is finished, the processed face is added to $completed\_faces$.

Listing 5.8. Grid drawing processor.

```
fillGrid($g$ : Graph){
  $current\_face$ = external face of $g$;
  determine leftmost, lowermost start node;

  initialize $visited\_faces$ : Set of faces
  initialize $completed\_faces$ : Set of faces
  **while**($current\_face \neq$ null){
    set edge positions into the grid;
    choose next $current\_face$;
  }
}
```

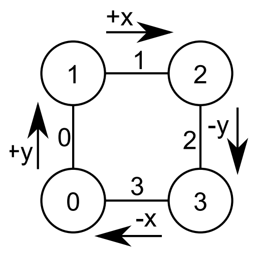


Figure 5.12. Increasing the grid positions.

After these sets are initialized, each face is traversed to set each adjacent, not already set node to a position into the grid. The face is traversed in clockwise direction with a start edge incident to the start node calculated before.

The relative length of the edge (v, w) determines the distance between the nodes v and w. Depending on the current face side the position is changed in horizontal or vertical direction. Figure 5.12 shows the increase of the grid position (x, y) depending on the current face side index si, labeled with $0, 1, 2, 3$.

▷ If $si = 0$ the y coordinate, representing the current height, is increased.

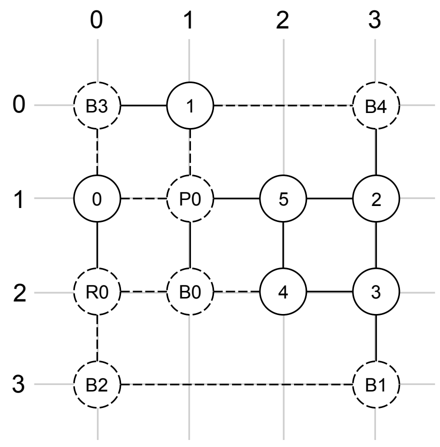
▷ If $si = 1$ the x coordinate, representing the current width, is increased.

▷ If $si = 2$ the y coordinate is decreased.

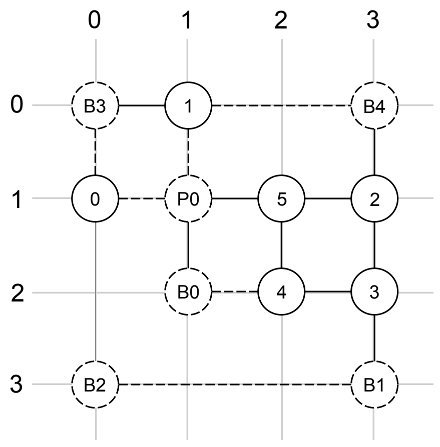
▷ If $si = 3$ the x coordinate is decreased.

Each found face during the traversal of a face is added to $visited\_faces$, and each finished face is added to $completed\_faces$ set, as mentioned in Section 5.2.3. Then a visited, not completed face is taken to do the described procedure again. After all faces are considered, each node of the graph has a unique position in the grid. Now, node sizes and real diagram coordinates can be set to each node in the grid.

The last step of the algorithm is to remove the before added dummies, which is described in the next subsection.



a. Grid representation of the $K_{3,3}$ with all dummies.



b. $K_{3,3}$ without rectangular shape dummies.

Figure 5.13. Removal of rectangular shape dummies.

5.3.2 Dummy Removal

The dummies are removed in three steps described by the following intermediate processors:

▷ `RectShapeRemover`

▷ `BendDummyRemover`

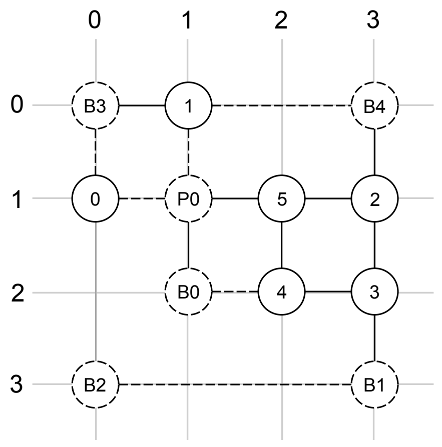
▷ `PlanarDummyRemover`

To illustrate the dummy removal steps the repeatedly considered $K_{3,3}$ graph is used (see Figure 5.13).

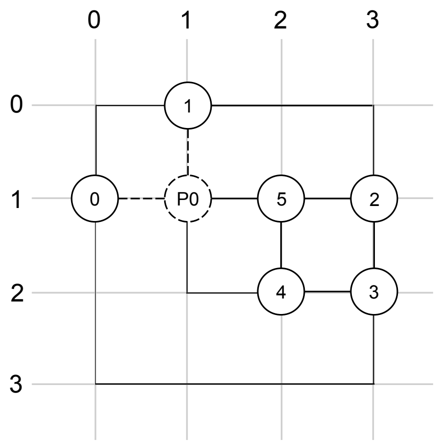
Rectangular Shape Dummies

The task of this intermediate processor is to remove all edges and nodes introduced by the `RectShapeProcessor`. In that process, each node n of the grid is checked for a `RectShapeDummy` property, which is added at the insertion. If such a node is found, it is processed as follows.

The edges (v, r) and (r, w) of the dummy node r with no property that identifies the edge $(r, corner(e))$, are merged back to (v, w). Additionally, r is removed from the graph and from the grid as well as the edge $(r, corner(e))$.



a. $K_{3,3}$ without rectangular shape dummies.



b. $K_{3,3}$ without bend dummies.

Figure 5.14. Removal of bend dummies.

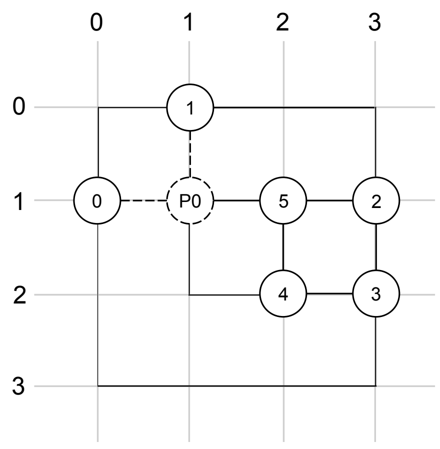
This is illustrated in Figure 5.13. Figure 5.13a has one rectangular shape dummy labeled with R_0. The edges $(0, R_0)$ and (R_0, B_1) are merged to $(0, B_1)$. Furthermore, node R_0 and the extending edge (R_0, B_4) are removed. The result, seen in Figure 5.13b, contains no rectangular shape dummies.

Bend Dummies

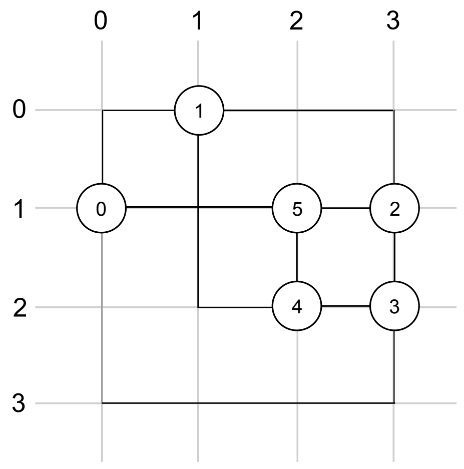
After the rectangular shape dummies are removed, the bend dummies are considered. The dummies are nodes for each bend-point itself and dummy edges, which are needed to subdivide the original edges during the `BendDummyProcessor`. Thus, it is easy to remove the dummies by simply iterating over all nodes of the grid, and if a node with property `BendDummy` is found, it is processed as follows.

The edges (v, b) and (b, w) incident to the bend dummy node b are merged back to one edge (v, w). In that process, the original edge is retained and the dummy edge is removed. If the dummy edge contains bend-points, they are added to the bend-point list of the original edge. If no incident edge of b is an original edge, an arbitrary edge is taken to be retained.

An example for this process is presented in Figure 5.14. For example, the edges $(0, B_2)$ and $(B_2, 1)$ can be easily merged back to the edge $(0, 1)$. B_2 and $(0, B_2)$ are removed, and the original edge $(B_2, 1)$ is updated to $(0, 1)$.



a. $K_{3,3}$ without bend dummies.



b. $K_{3,3}$ without planarization dummies.

Figure 5.15. Removal of planarization dummies.

Additionally, the edge $(0, 1)$ is equipped with a bend-point with the position of B_2. Since the edge $(0, B_2)$ contains no bend-points, the bend-point list of $(0, 1)$ needs no more updates.

For edges with more bend dummies like $(0, 3)$, there is a possibility that no original edge around a bend dummy. For example, if the grid iteration finds B_0 first and then B_1, then one of the incident edges is removed and the other is equipped with a bend-point at the position of B_0. If B_1 is removed, the originally merged edge $(0, 3)$ is equipped with a bend-point with the position of B_1. Secondly, the bend-points of the removed edge $(B_1, 3)$ are added to the bend-point list of $(0, 3)$. When the process is done for all bend dummies the result contains no bend-point dummies (see Figure 5.13b).

Planarization Dummies

The last remaining dummies are the planarization dummies. A planarization dummy is always incident with four edges, two inserted dummy edges and two previously set edges. The dummy edges can be identified with the `PlanarizationDummy` property. The opposite incident edge of a dummy edge is a previously set edge. This can be easily determined with the angle-data of the orthogonal representation. Hence, the opposite incident edges are merged back to the previously set edges and the planarization dummy is removed.

Figure 5.15 illustrates that method. The only planarization dummy node P_0 is removed and the opposite incident edges $(1, P_0), (P_0, 4)$ and $(0, P_0), (P_0, 5)$ are merged. The result is presented in Figure 5.15b which does not contain any planarization dummies.

The order of the dummy removal processors is important. They come in the opposite order in comparison to the dummy insertion processors. This results from the fact that each dummy insertion step assumes an input graph that includes dummies that were inserted in the preceding steps. For example, the `RectShapeProcessor` needs dummies for edge bends. If these bends would be removed before the rectangular shape dummies are removed, exceptions would be caused during the step of the `RectShapeRemover`. The implementation would need to handle these exceptions, such that the code would not be readable and would be more error prone. Furthermore, the intermediate processors should be as generic as possible in order to depend on other intermediate processors as less as possible. This is ensured by the described order.

Chapter 6

Extensions

The preceding chapters consider the different parts of the TSM algorithm. This chapter describes implemented features as well as ideas for not realized features.

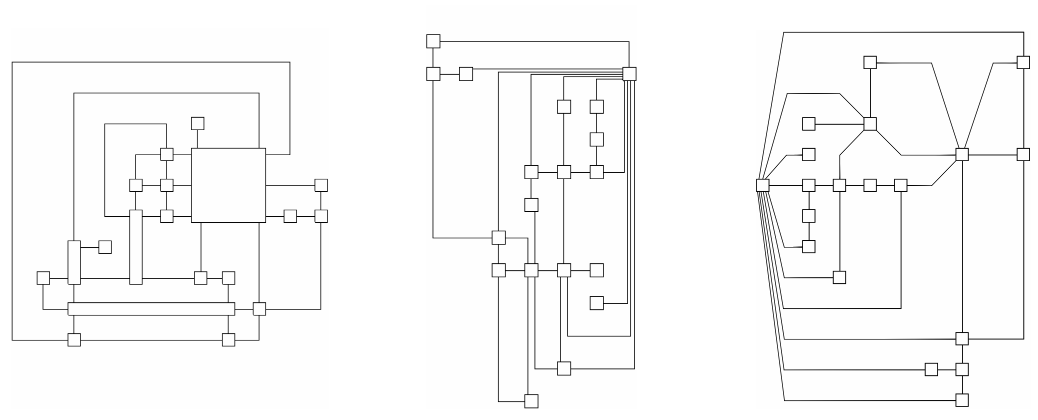
Firstly, different techniques for extending the TSM algorithm to provide graphs with a higher degree than four are considered. In that context the implementation of the Giotto algorithm as well as the implementation of the Quod algorithm are presented.

Furthermore, extensions such as interactive planarization and the finding of an optimal embedding are discussed. Secondly, their possible realizations in KLay Planar are described. The compaction works with input graphs that contains faces in rectangular shapes. This often leads to non-space-minimal results. Hence, a second compaction heuristic is mentioned. Furthermore, planar drawing alternatives are considered. Additionally, a method is presented to allow directed graphs. Finally, ideas to handle basic extensions, like handling of self-loops and multi-edges are presented.

6.1 Node Degree Higher Than Four

A restriction of the presented TSM approach is that nodes cannot be incident to more than four edges, since in the orthogonalization phase at most four edges can be incident to a node and in the compaction part vertices of a graph are placed on a two-dimensional grid. Nodes with arbitrary degree are required in many practical applications. Hence, this problem has been studied extensively in the past and several extensions were formulated. Three of them are considered which mainly bases on Klau and Mutzel [KM98], Kaufman and Wagner [KW01],and Eiglsperger, Kaufman and Siebenhaller [EKS03]. They are illustrated in Figure 6.1.

One approach is the so called *Giotto* model [TDBB88]. Vertices with a higher degree than four are drawn as boxes on the grid that consist of more than one grid position. This approach is simple to realize but requires more drawing space than



a. Giotto result. **b.** Kandinsky result. **c.** Quod result.

Figure 6.1. Planar graphs drawn with different high degree approaches [KM98].

other approaches. Furthermore, the size of high-degree nodes is increased, which makes this approach problematic for graphs containing size constraints on vertices.

The *Kandinsky* approach represents all graph vertices as squares of equal size, arranged on a coarse vertex grid [FK96]. Edges are allowed to run on a finer grid, which is defined by the maximal vertex degree. Thus, the edges can be put closer to each other.

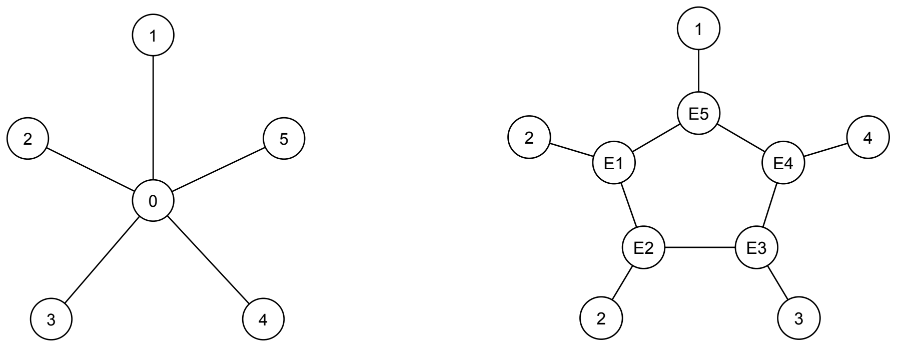
Another approach is called *Quod* (*quasi–orthogonal grid embedding*) algorithm that is an extension of the Giotto model, which allows the edges to leave the grid around high degree vertices and become diagonal [KM98].

Each of the presented techniques has its advantages and its disadvantages [KM98]. In short, they are illustrated in Table 6.1.

Table 6.1. Advantages and disadvantages of the different high degree approaches.

Property	Giotto	Kandinsky	Quod
Used Space	$--$	$+$	$--$
Number of Bends	$++$	$-$	$-$
Original Vertex Size	$-$	$+$	$+$
Complexity	$++$	$-$	$+$

As seen in Figure 6.1, Giotto and Quod lead to results with more drawing space than the Kandinsky algorithm, since in the Kandinsky approach lots of edges are drawn in the finer grid. Giotto produces results with few bends, because the size of



a. High-degree vertex.

b. Expansion cycle that avoids vertex degrees higher than four.

Figure 6.2. Transformation of a high-degree node into an expansion cycle.

each high degree node is increased on multiple grid positions. Thus, the incident edges can be connected with that node without additional bends. However, the original vertex size is increased, which can be problematic in graphs with fixed vertex size constraints such as UML class diagrams.

To avoid this, the Giotto approach can be extended by the Quod algorithm. Each grown high degree node can be resized to its original size. Then, the incident edges are equipped with bend-points at the connecting positions, such that a diagonal edge segment connects that bend-point with the resized node.

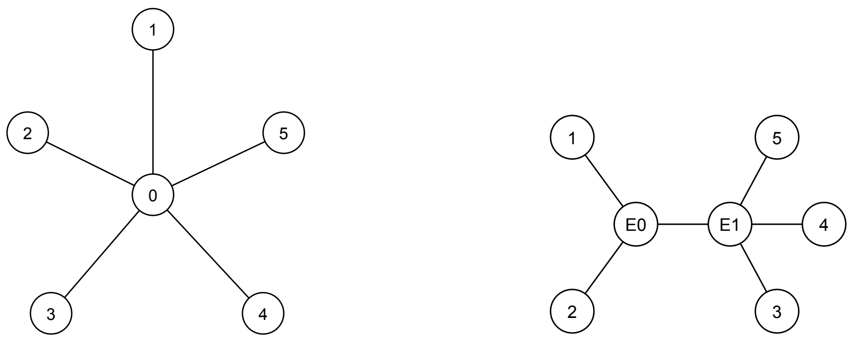
In order to allow high degree nodes in the context of KLay Planar, the Giotto algorithm is realized, because of its simple nature and the low bend size. Additionally, the implementation of the Giotto approach is extended with the Quod algorithm to support node size constraints. The user of KIELER is allowed to decide which high-degree node strategy should be taken.

The Giotto approach and the Quod approach belong to the type of *reduction* algorithms that can divide a node in two different ways:

▻ Expansion cycle of dummy nodes.

▻ Sequence of successive dummy nodes.

An *expansion cycle* replaces every high-degree vertex v_h of a graph. Each edge of v_h is assigned to a vertex of the cycle and all cycle nodes are connected to two other cycle nodes (see Figure 6.2). Thus, each vertex of that cycle is incident to two



a. High-degree vertex.

b. Sequence of dummy nodes that avoids vertex degrees higher than four.

Figure 6.3. Transformation of a high-degree node into an sequence of dummy nodes.

dummy edges and an original edge. This is done before the orthogonalization is processed such that the input embedding of the orthogonalization contains only nodes with a maximal degree of four. After the compaction phase is finished and the grid drawing processor is finished, the dummy nodes of the expansion cycle are set to a position of the grid. These positions are used to determine the size and position of v_h. Results like in Figure 6.1a are produced.

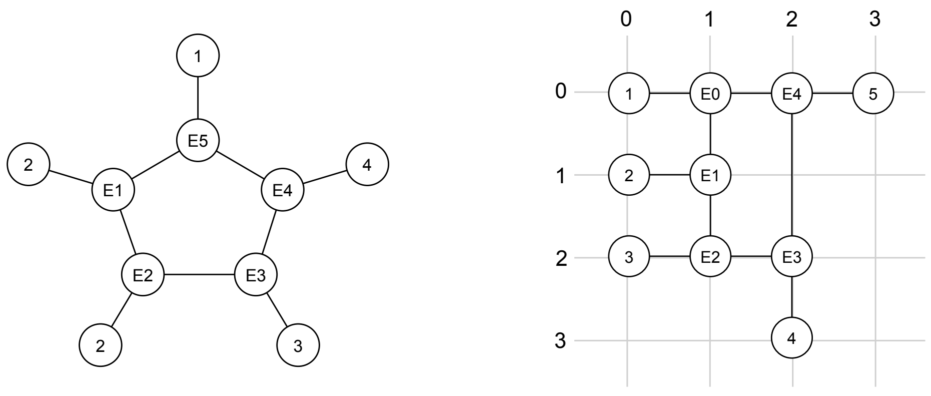
The second approach of the reduction is to replace v_h with a sequence of dummies as in Figure 6.3. This allows every dummy node to be incident to at least two original edges. In many cases this approach leads to more compact results.

6.1.1 Implementation

The implementation is divided into three parts. Firstly, the creation of expansion cycles to avoid high-degree nodes is considered. Furthermore, the realization of the transformations back to high-degree nodes with the Giotto approach and the Quod approach are discussed.

Reduction

In order to create expansion cycles for every high-degree node an intermediate processor, called `ExpansionCycleProcessor` is introduced. It is added to the first processing slot of the preprocessors of the orthogonalization. The possibly added



a. Embedding with an expansion cycle. **b.** Resulting diagram.

Figure 6.4. Performing orthogonalization and compaction on expansion cycle.

cycles form new faces that are calculated at the face determination processor. However, the external face processor chooses the face with maximal edge count, which could possibly be the expansion cycle. Thus, an exception has to be done where no expansion cycle face is taken as the external face. Dummy nodes of the expansion cycle faces are marked with special properties to identify them.

Table 6.2. The adjusted preprocessor list of the orthogonalization.

Typ	Name	Description
Preprocessing	`ExpansionCycleProcessor`	Extend an embedding with expansion cycles to avoid high-degree nodes.
	`FaceDetermination`	Determining the faces of a planar embedding.
	`ExternalFaceProcessor`	Choice of an external face.

Each high-degree vertex v_h of the embedding is processed as follows to create an expansion cycle. All incident edges of v_h are stored, and v_h is removed from the graph. Then, the expansion cycle is created by adding a dummy node for each of the stored edges. Each dummy node is connected with its two neighbor dummy nodes by dummy edges (see Figure 6.2b). The correct embedding of the cycle has to be ensured. In that process the dummy edges are embedded in a way so that they form a new face and each original edge is connected to a dummy node such that no edge crossing appears. Since the embedding of v_h is planar, the cycle is built by creating the dummy nodes with respect to the counter-clockwise order of incident edges of v_h.

An expansion cycle may not have a rectangular shape after the orthogonalization is processed. Since the coordinates and the position of each high-degree vertex of that cycles are computed in order to be placed in the grid correctly, it would not be a good idea to let the shapes of the expansion cycle get too complicated. Thus, the expansion cycle is forced to be in rectangular shape by modifying the network creation of the orthogonalization algorithm.

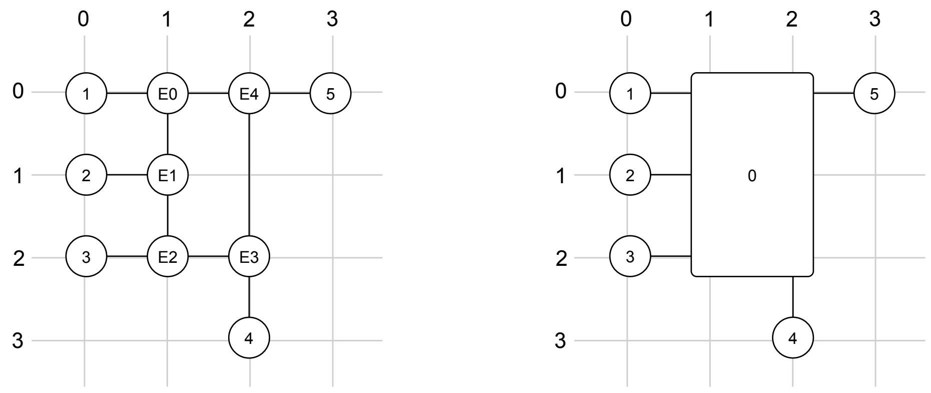
Mutzel et al. presented a technique for that extension in 1998 [Mut+98]. In that method it is distinguished between normal faces and expansion cycle faces. Each face has a rectangular shape if neither of the angles of the vertices in that cycle exceeds 180°. Since each vertex of the expansion cycle has a degree of 3 the cycle can only form angles of at most 180°. Thus, the $\leqslant 180°$ angle requirement is automatically satisfied. Secondly, it has to be avoided that bends are added to the bounding cycle edges. Bends are set by setting the flow in the network on arcs (g, f) where g and f are faces. Hence, deleting the arcs (g, f) where f is an expansion cycle face makes such flows impossible, and no bends are placed on the expansion cycle edges.

Table 6.3. The adjusted postprocessor list of the compaction.

Typ	Name	Description
Postprocessing	`GridDrawingProcessor`	Mapping the graph elements on a grid.
	`GiottoProcessor`	Transformation of expansion cycles back to high-degree nodes.
	`RectShapeRemover`	Removal of dummy nodes and dummy edges that ensure rectangular faces.
	`BendDummyRemover`	Removal of bend-point dummy nodes.
	`PlanarDummyRemover`	Removal of planar dummy nodes.

Giotto Implementation

The high-degree node enlarging is done by an intermediate processor called `GiottoProcessor`. This is added to a processing slot after the compaction phase and after the grid drawing processor, since a grid drawing is needed to calculate the sizes of high-degree nodes. The adjusted postprocessing order is presented



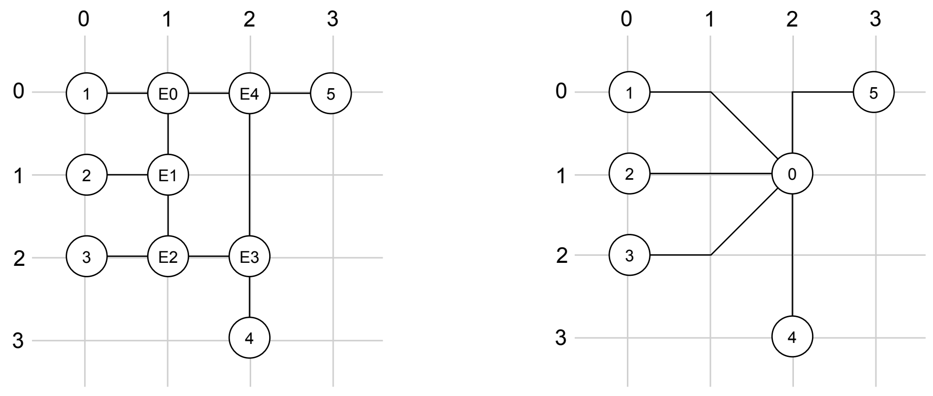
a. Drawing with expansion cycle. **b.** Giotto result.

Figure 6.5. Back transformation to the high-degree node with Giotto approach.

in Table 6.3. The processor is added before the other dummy removal processors. This processor decreases the number of nodes of the grid. Thus, it is added before the other removal processors, such that these do not need to iterate unnecessarily over the expansion cycle. This can be done, because the `GiottoProcessor` does not further influence the other removal processors.

Figure 6.5 illustrates the transformation of an expansion cycle back to a high-degree node. Each dummy node of the expansion cycle consists of a property identifying its original node v_h. Hence, each dummy can be assigned to a unique high-degree node. As mentioned before, the positions of the dummy nodes are used to determine the size of v_h as well as to determine the node positions. The position of the dummy node with the lowest x- and y-coordinate is taken. Secondly, the position of the node with highest x- and y-coordinate is taken. Afterwards, all dummy elements of the expansion cycle are removed from the grid. Then, v_h is spanned from the smallest x-coordinate to the highest, and from the smallest y-coordinate to the highest. Since each expansion cycle always forms a rectangle there are exactly one dummy node with smallest x- and y-coordinate as well as exactly one dummy node with highest x- and y-coordinate. The original edges are again connected to the vertex at the positions of the dummy nodes, and v_h is transformed back to a single node.

The example of the Figure 6.5 has an expansion cycle $(E_0, E_1, E_2, E_3, E_4)$, which is desired to be transformed into one high-degree node. The position of E_0 is $(1, 0)$ with smallest x- and y-coordinate. E_3 has a position of $(2, 2)$ that forms the highest



a. Drawing with expansion cycle. **b.** Quod result.

Figure 6.6. Back transformation to the high-degree node with Quod approach.

coordinates. The edges and nodes of the expansion cycle are removed. v_h, in the example vertex 0, is spanned from $(1, 0)$ to $(2, 2)$ and occupies six grid positions.

Quod Implementation

As mentioned before, the Quod approach does not change the size of the nodes, and it works on the expansion cycle of the reduction, like the Giotto algorithm. The intermediate processor `QuodProcessor` is introduced for this approach that can be exchanged with the Giotto approach by a layout option of KIML. In that context, the smallest x- and y-coordinates and the highest x- and y-coordinates are used to calculate the position of v_h, e. g., the position of v_h is the middle point of these coordinates. This point is the average value of the smallest and highest x-coordinates, and the average of the smallest and highest y-coordinates. If the result is a non-integer, the coordinates are rounded up.

The cycle is removed, and the original edges are connected with v_h, just like it is in the Giotto approach. Now the Giotto and Quod approach start to differ. The original edges are equipped with bend-points at the positions where they have been connected with the dummy nodes. Then, the resulting drawing contains diagonal segments from the dummy positions to v_h, and the drawing is quasi orthogonal.

An example of the Quod transformation is presented in Figure 6.6. The high-degree vertex 0 is set to the grid position $(2, 1)$, since the rounded up average of the positions $(2, 2)$ and $(0, 1)$ is $(2, 1)$. The expansion cycle is removed, and a bend-point is added to each original edge at the positions $(0, 1), (1, 1), (2, 1), (0, 2), (2, 2)$.

Future Work

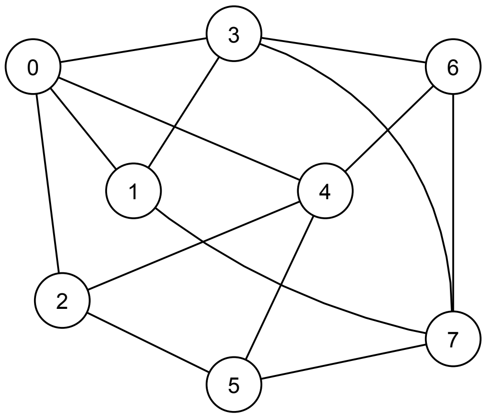
The Kandinsky approach could be implemented to get more compact results and intermediate processors can be used for the creation of finer grids. The implemented method to resolve high-degree nodes could be reused.

A second extension would be the implementation of the dummy nodes sequence to resolve high-degree nodes. Again, an intermediate processor that resolves high-degree nodes in a sequence instead of in an expansion cycle could be introduced. In some situation this leads to more compact drawings, as can be seen in Figure 6.2 and Figure 6.3. Here, only two dummy nodes are introduced by the sequence approach.

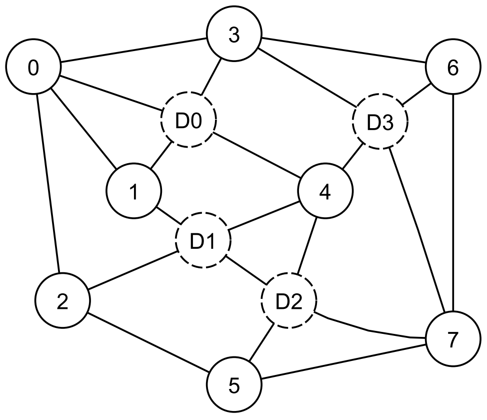
6.2 Interactive Planarization

Additionally to the aesthetics criteria of Section 2.1.3 there is another important criterion, the so called *user mental map*. In an interactive visualisation system, changes to a drawing are made constantly, sometimes by the user and sometimes by the application. These changes frequently spoil the layout, since node overlaps might happen. A layout algorithm that rearranges the layout preserves the user mental map criterion if it destroys the mental map of the user as less as possible, by minimizing changes to the layout [Ead+91]. Most of the existing layout algorithms are designed for layout creation, and so is the TSM approach. Such layout algorithms may completely rearrange the layout and thus destroy the mental map of the diagram.

The implemented TSM algorithm could lead to completely different drawings, since the planarization takes an arbitrary embedding of the given graph. In order to set the user mental map criterion on the top of the importance list of the TSM algorithm's criteria and preserving more the mental map, the planarization could be exchanged. In KIELER automatic layout is always done on a given graph drawing. The first step of the existing planarization algorithm calculates for a possibly non-planar graph G a more or less arbitrary planar subgraph G', without regarding the original embedding. Thus, the embedding of the graph G can be calculated from the original node positions of the given graph drawing. The given embedding is possibly may not be planar. All edge crossings needs to be identified. This could also simply be done by checking every two edges of the graph whether of their segments cross each other. For each crossing a dummy vertex could be introduced to remove the non-planarity property.



a. Embedding containing edge crossings.



b. Adjusted planar embedding of Figure 6.7a.

Figure 6.7. Planarization with preserving the given embedding.

The described planarization variant could be implemented within a phase of KLay Planar which could be optionally exchanged with the original one. The consecutive parts of the algorithm would not need to be adjusted, since the result of the exchanged planarization would be a planar graph. Finally, the planarization dummies would be removed in the `PlanarDummyRemover` intermediate processor. An example is presented in Figure 6.7. The embedding is the same as in the original graph but with the disadvantage that possibly more crossings dummy nodes are inserted. An embedding with less dummies would result if edge $e = (1, 7)$ would be placed around the graph such that e would cross $(0, 2)$.

In addition to the approach of Boyer and Myrvold, and the interactive planarization a combination of the two algorithms could be useful. That is, first to generate a layout with Boyer and Myrvold approach to get a drawing with few edge crossings. Then, all the other times a new layout is desired the interactive planarization approach could be used to ensure a better orientation of the user and still few edge crossings of the existing embedding. This is a more dynamic approach, which would need some research and possibly some changes to the KIML architecture.

6.3 Optimal Embedding

Thorsten Kerkhof considered the determination of an optimal embedding for a given planar embedding [Ker07]. There are different embedding properties that can be optimized at the calculation of a planar embedding. Two of them are:

- ▻ Minimization of the depth graph embedding [PT00].
- ▻ Maximization the edges of the external face.

The depth of a planar embedding of a graph is a measure of the topological nesting. Maurizio Pizzonia et al. figured out that minimum depth has advantages compared to high depth for the aesthetics criteria total bend number and sum of edge lengths [Piz06]. Both approaches can be combined since their ways of processing are similar and both approaches are orthogonal to each other [GM04]. Kerkhof implemented the different approaches and showed with experimental studies that the combination of both approaches leads to the best results.

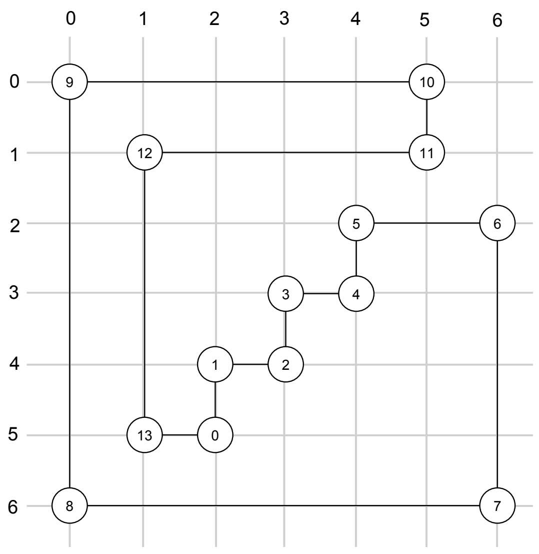
This extension could be implemented in the environment of KLay Planar, too. The existing `ExternalFaceProcessor` takes the face with maximal edge count as external face. This intermediate processor could be extended with the mentioned algorithms to get better results.

6.4 Compaction

The implemented compaction requires input graphs which are in rectangular shape. This restriction often leads to results that are not minimal. In order to illustrate the drawing gap resulting the example of Figure 6.8a is presented that shows a drawing of an orthogonal representation computed with the implemented algorithm of Tamassia. The problem is that the drawing takes more drawing area than necessary. The orthogonal representation with minimal drawing area is shown in Figure 6.8b.

The compaction of Tamassia computes often non-minimal results, however in polynomial time. Klau and Mutzel developed a more complicate approach and compared it with the heuristic of Tamassia [KM99b]. The comparison shows that their approach produces much more compact drawings than the one of Tamassia and secondly they showed that their algorithm works in polynomial time.

They provide a necessary and sufficient condition for all feasible solution which bases on existing paths in the so called *constraint graphs* in x- and y-direction. The


a. Drawing of Figure 6.8a with removed dummies.

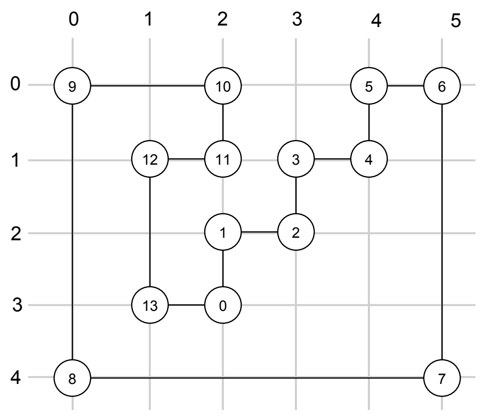

b. Drawing with minimum space.

Figure 6.8. Two drawings with different edge length, respectively different drawing area.

two constraints graphs in horizontal and vertical direction specify the shape of the given orthogonal representation. Their algorithm works with input graph drawings that contain no bends and no edge crossings. The task of that algorithm is to extend the constraint graphs to a complete pair of constraint graphs by adding new arcs for which the defined condition is satisfied and the total edge length of the drawing is minimal. A more detailed description is presented in "Optimal Compaction of Orthogonal Grid Drawings" [KM99b].

6.5 Planar Drawing Alternatives

Drawing planar graphs orthogonally is only one option. In the literature there are various approaches to draw them. Most of them base on so called triangulated graphs [FPP90; KB92; Kan92; Kan96; Bad+10]. Hence, this approach is presented here. As mentioned in Section 2.1, adding any edge to a maximal planar graph would lead to an edge cross, such that the graph would be no longer planar. Maximal planar graphs consist of faces bounded by three edges, thus they are often called *triangulated graphs*. De Faysseix, Pach and Pollack introduced an algorithm

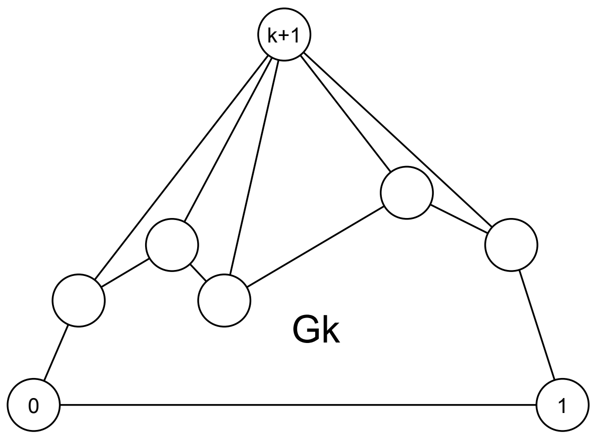


Figure 6.9. Triangulating a planar graph bases on the canonical ordering.

that draws planar graphs straight-line without crossings [FPP90]. The algorithm works in three steps:

1. First, the graph is triangulated.
2. Afterwards, a *canonical ordering* for triangulated plane graphs is computed.
3. Finally, the vertices of the graph are placed according to the canonical order.

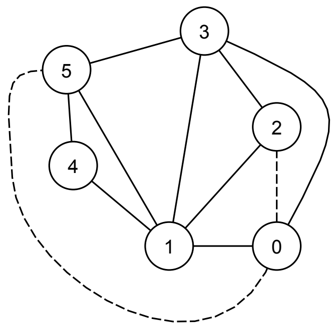
Kant introduced a technique to triangulate a planar graph in linear time [KB92]. The planar canonical ordering is an ordering of vertices $v_1, v_2, ...v_n$ of a maximal planar graph G with the characteristics that

▻ G is biconnected, and

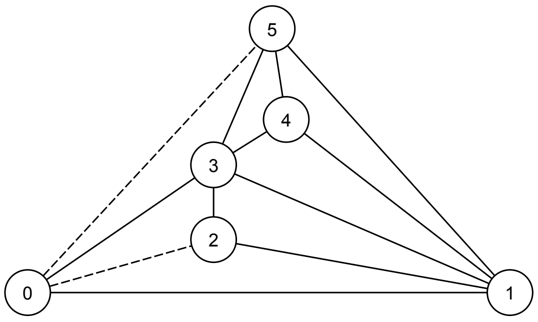
▻ v_1 and v_2 are on the external face.

Figure 6.9 illustrates the constructing of a maximal planar graph with the canonical order. After a subgraph G_k of G is drawn, v_{k+1} can be drawn without edge crossings, since it is adjacent only to a sequence of vertices on the outer face of G_k.

An example for the mentioned algorithm is presented in Figure 6.10. The dashed lines are dummy edges introduced by the augmentation to triangulate



a. Planar graph.



b. Triangular straight-line drawing.

Figure 6.10. An example of a planar straight-line drawing.

the graph. Computing the canonical order of the graph leads to a drawing of Figure 6.10b. The dummies are still part of the graph and have to be removed for the final drawing.

This approach can be implemented in KLay by using the planarization phase of KLay Planar and then introducing a new phase which does the steps described above.

6.6 Handling Directed Edges

There are a lot of applications that are modeled with mixed graphs, for example class diagrams containing class hierarchy (see Figure 6.11). The problem in the context of directed edges at the planarization is to find a planar upward embedding. As mentioned in Section 2.1, an upward drawing of a directed graph is a drawing where each edge is represented by a curve monotonically increasing in the vertical direction. Even if the testing of arbitrary mixed graph to be upward planar is NP-hard, there are heuristics that work efficient for graphs with single source node, the so called st-graphs [HL91].

Chimani, Gutwenger, Mutzel, and Wong introduced an layer free approach for upward edge crossing minimization [Chi+10]. Their algorithm can be applied to acyclic directed graphs with single source.

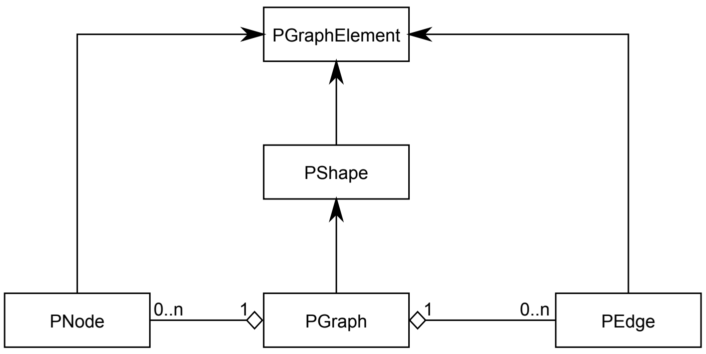


Figure 6.11. Snippet of a class diagram of KLay Planar.

Secondly, they compared their algorithm with the one of Sugiyama et al. who introduced a method to draw graphs upward in 1981 [STT81]. Sugiyama's approach consists of three phases:

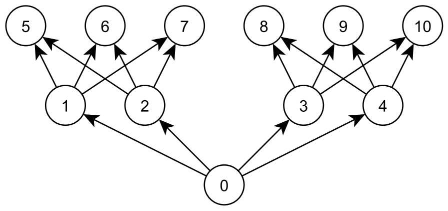
1. *Layer Assignment*; Assign the nodes to layers such that edges point from lower to higher layers. Furthermore, split edges spanning several layers such that edges connect only nodes on adjacent layers.

2. *Crossing Reduction*; Reorder the nodes on each layer with the objective to reduce the number of edge crossings.

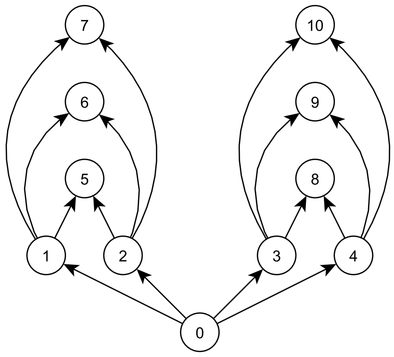
3. *Coordinate Assignment*; Assign final node and bend-point coordinates.

Chimani et al. tried to solve the problem that assigning nodes to fixed layers in the first step can massively influence the subsequent crossing minimization step, see Figure 6.12a. Requiring edge crossings would become unnecessary if another "better" layer assignment would be chosen, see Figure 6.12b. Hence, they merged the phases 1 and 2 of Sugiyama's algorithm to combine the layer assignment and crossing minimization in order to obtain drawings with less edge crossings.

The algorithm of Chimani et al. underlies the following idea. As input a st-graph G is assumed. If G contains more than one source node, an additional dummy node is introduced as super source node, which is connected to all original sources. At first a *feasible upward planar subgraph* is calculated, and afterwards the remaining edges are iteratively added to that subgraph such that few crossings arise. Then



a. Representation with unfortunate layering.



b. Upward planar representation of the same graph as of Figure 6.12a.

Figure 6.12. Two representations with different layering strategies.

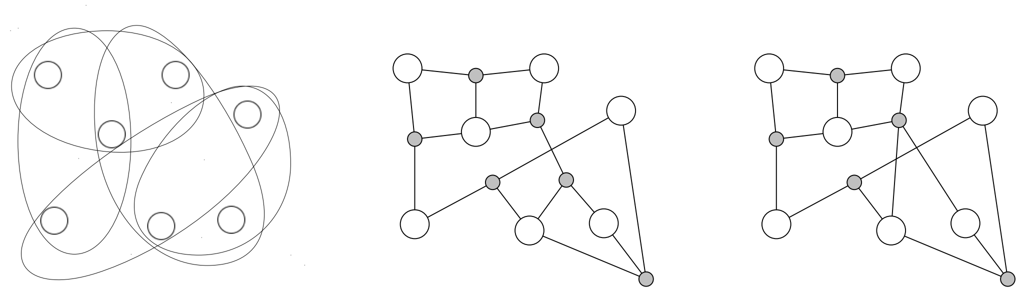
these crossings are replaced with dummy nodes such that the consecutive parts of Sugiyama's algorithm (phase 3) are able to process on a planar graph. Inserting an edge *e* into a planar graph thereby means that all arising crossings will lie on *e*; they do not introduce additional crossings purely on the planar graph itself. The main challenge with this approach is that, in contrast to the approach for undirected graphs, the edge insertion steps are not independent of each other.

A more detailed explanation and a pseudo code for handling that edges is presented in the paper *Layer-Free Upward Crossing Minimization* [Chi+10]. In order to implement this algorithm in the existing TSM implementation the planarization phases have to be exchanged with a variant of their own, such that upward planarity testing would be possible, and the edge insertion steps would need to be extended to allow dependent edge insertions.

6.7 Hypergraphs

Many applications such as electric schematics require hypergraphs, hence handling of such graphs would be a nice extension of the implemented TSM algorithm.

Definition 6.1 (Hypergraph)**.** A *hypergraph* is a generalization of a graph in which an edge can connect any number of vertices. Formally, a hypergraph is a pair $H = (X, E)$ where X is a set of vertices, and E is a set of non-empty subsets of X called *hyperedges*.



a. Drawing with *subset standard*. **b.** Drawing with *edge standard, tree-based*. **c.** Drawing with *edge standard, point-based*.

Figure 6.13. Different representations of a hypergraph [Kut10].

Mäkinen defined different variants to draw hypergraphs, namely the *subset standard* and the *edge standard* [Mäk90]. The second variant is divided into the *tree-based* drawing style and the *point-based* drawing style. Figure 6.13 illustrates the different drawing styles. The *subset standard* tends to get confusing quickly. Secondly, it is hard to determine clear edge crossings. Thus, most applications use the *edge standard* style [San04; EGB06]. In the tree-based drawing style each hyperedge $h \in E$ is drawn as a tree-like structure of lines, whose leaves are the incident nodes of h. If the tree-like structure of every hyperedge is restricted to be a star which consists only one hypernode per hyperedge, the point-based drawing style is obtained.

In 2010 Kutschmar presented a technique that extends the planarization of the implemented TSM algorithm, in which it is possible to remove edge crossings from hypergraphs [Kut10]. His implementation work on hypergraphs with *edge standard*, especially the point-based shape. Hypergraphs in tree-based shape are transformed to be in point-based shape. His method to handle hypergraphs efficiently includes the insertion of dummy nodes. A back transformation to the original graph could be done as the dummy removal processors of the compaction's processing.

6.8 Port-Constraints

The *ports* of an edge $e = (v, w)$ of a graph drawing are the points where e touches the nodes v and w. Ports are used at dataflow diagrams and at electric schematics and have a specific semantic interpretation. Thus, practical applications have additional constraints on the drawings for such ports, since the positions of the ports is not arbitrary. Using rectangular vertices, the two ports of an edge can lie at the top, right, bottom or left side of a vertex. If an edge e is restricted to be connected with its incident vertices at prescribed sides, then it is called a *side-constraint*. Besides to side-constraints, *port-constraints* define prescribed attachment points of edges at a vertex.

In order to provide port-constraints in the implemented TSM algorithm the technique by Gutwenger, Klein and Mutzel could be implemented [GKM07]. They investigated embedding constraints that restrict the acceptable order of incident edges around a vertex. Additionally, they introduced a set of hierarchical embedding constraints that include *grouping*, *mirror*, and *oriented* constraints. To be more precise, the embedding constraints of the edges around a vertex v can be modeled as a rooted, ordered tree whose leaves are the edges incident to v. The inner vertices of the trees represent the different embedding constraints as follows:

- ▷ Grouping constraint vertices; the order of the children of these vertices is arbitrary.
- ▷ Mirror constraint vertices; the order of the children of these vertices may be reversed.
- ▷ Oriented constraint vertices; the order of the children of these vertices is fixed.

A graph is called *ec-planar* if it admits a planar embedding satisfying the given embedding constraints. Gutwenger et al. present an $O(n)$ algorithm for computing the corresponding *ec-embedding* in the context of the planarization of the TSM algorithm. In that process, the mentioned tree structure is used by an extended planarization algorithm in order to obtain a planar embedding that respects the predefined constraints.

6.9 Edge Enhancement

The algorithm implementation of KLay Planar is not yet ready to handle labeled edges, multi-edges and self-loops. In the following, strategies are presented to handle such edges.

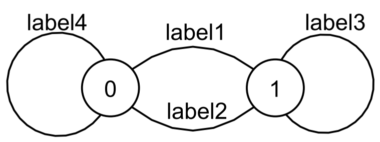


Figure 6.14. Example graph containing labeled multi-edges and labeled self-loops.

6.9.1 Edge Labels

A typical example of the usage of edge labels are the streets of a city map, which must be easy to identify by their names. Binucci et al. compared some edges labeling heuristics with each other [Bin+02]. In that context they presented three requirements of good labeling:

▻ Any label must be easy to identify to an edge to which it is assigned to.

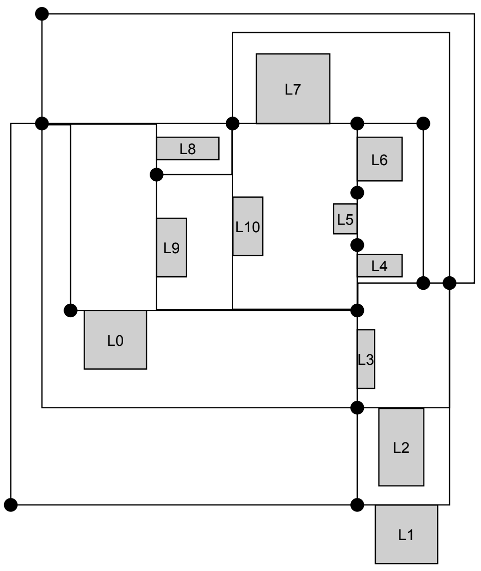
▻ A label assigned to an edge can not overlap with other drawing elements.

▻ A label must be placed in the best position among all acceptable positions.

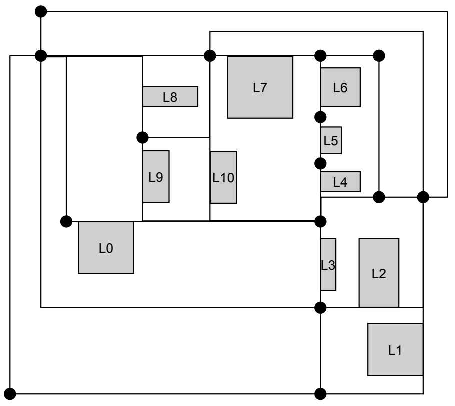
One optimization goal is to minimize the drawing area. Most of the labeling problems have been proved to be NP-hard. The mentioned edge label placement approaches treat drawings as sets of distinct straight-lines. Secondly, they do not allow to change the geometry of the given drawing.

A first approach to handle edge labels in the context of the TSM algorithm is to model the edge labels as dummy vertices. Then, these dummies are placed arbitrary at a segment of the labeled edge and in an arbitrary direction. However, it is not clear how to guarantee that the resulting labeled drawing is well readable, since this approach does not allow any control on the choice of the best position for a label along an edge. Figure 6.15a shows an orthogonal drawing with such a random label placement.

Klau and Mutzel et al. investigated a technique to put labels on vertices in a suitable manner, where they combine the compaction of the TSM algorithm with labeling techniques [KM99a]. Binucci et al. focus on the integration of a variant of Klau's technique for edge labeling.



a. Drawing with random label placing.



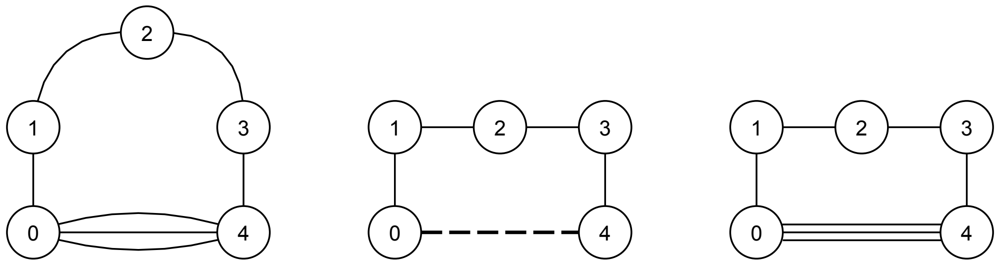
b. Drawing with the `Slow Labeler`.

Figure 6.15. Different labeled drawings of the same labeled orthogonal representation [Bin+02].

A description of their considered problem is as follows. Let G be a planar graph, let H be an orthogonal representation of G and let L be a set of labels for the edges of G where each edge is associated with at most one label. The algorithm computes an orthogonal grid drawing of G such that its edges are labeled and have the shape defined in H. Each edge label is modeled as rectangle with a given integer width and height. Their proposed heuristics bases on the *greedy strategy* that computes a drawing by inserting a label at a time. They designed different algorithms for the edge labeling problem and experimentally compared their performances. The best proven heuristic is defined as follows:

▷ *Max-ratio-delta-area*: The label with highest insertion priority is the one with maximum aspect ratio. If two labels have the same aspect ratio, the one that causes the minimum increase of the area is chosen.

They implemented two variants of that heuristic, namely `Fast Labeler` and `Slow Labeler`. The `Fast Labeler` needs much less processing time than the `Slow Labeler` and produces useful results. The `Slow Labeler` computes more compact drawings



a. Embedding with a multi-edge. **b.** Drawing with a dummy multi-edge. **c.** Drawing with an multi-edge.

Figure 6.16. Processing steps of handling multi-edges.

than `Fast Labeler`, however it needs more processing time. Figure 6.15b shows a drawing that would result if the `Slow Labeler` would be used to arrange the same graph as in Figure 6.15a. The detailed algorithms can be looked up in *Labeling Heuristics for Orthogonal Drawings* [Bin+02].

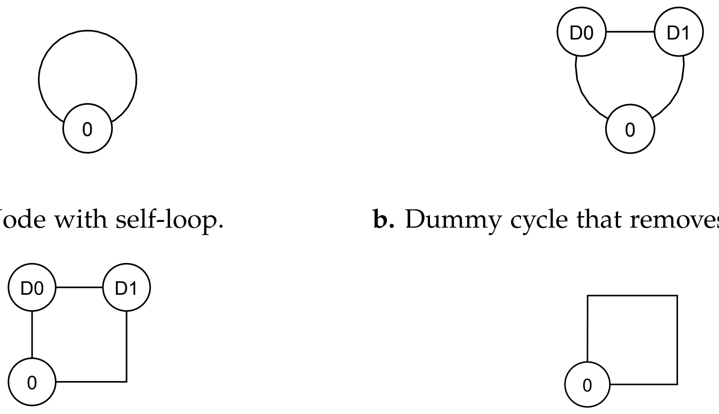
6.9.2 Multi-Edges

Multi-edges are problematic for many parts of the algorithm, since an edge is identified by its source and target nodes. Multi-edges can be connected several times to the same source and target.

Hence, multi-edges need special attention. An idea to handling such edges is described in the following. A multi-edge e_{mu} could be treated as a single edge e'_{mu} such that there are no problems while processing the algorithms phases. Afterwards, e'_{mu} could be extended back to e_{mu}. Instead of setting e'_{mu} on the grid, all the edges of e_{mu} could be placed parallel to each other along the endpoints and bends of e'_{mu}.

The implementation of merging a multi-edge to one edge could be done with an intermediate processor at the beginning of the implemented algorithm. All following algorithm parts would then work with a normal edge instead of the multi-edge. The back transformation could be enqueued in the dummy removal processors of the postprocessing of the compaction.

An example of this technique is presented in Figure 6.16. The edge $(0, 4)$ is a multi-edge containing three single edges. This edge would be replaced by the dashed edge in Figure 6.16b. Then, all phases could be processed correctly. Afterwards, the edge $(0, 4)$ would be divided into three parallel edges.



a. Node with self-loop. **b.** Dummy cycle that removes the self-loop.

c. Orthogonal drawing of the dummy cycle. **d.** Final drawing of a self-loop.

Figure 6.17. Handling of self-loops.

Edge Crossings Another problem with multi-edges are edge crossings. Such crossings with e_{mu} are expensive multiple times, since each containing edge of e_{mu} is crossed.

The position of each crossing is determined by the planarization and identified by a dummy node. During the mentioned planarization, Dijkstra's algorithm is used to determine the shortest path in the dual graph between the endpoints of the edges to be inserted. Thus, the edges cross the fewest other edges. In order to let multi-edges be crossed as less as possible all edges of the graph could be equipped with weights. All non-multi-edges could be weighted with 1 and all multi-edges could be weighted with the number of contained edges, respectively. Dijkstra's algorithm is known to be able to handle such weighted edges [Dij59]. Hence, using the edge weights would lead to less edge crossings of multi-edges.

6.9.3 Self-Loops

As mentioned before, a self-loop $e_{sf} = (v, v)$ is an edge with same start- and target node. Such edges are problematic for every phase of the algorithm, thus it would be meaningful to transform self-loops before the phases are processed in something the phases can handle.

An idea for solving this is to introduce a dummy cycle for each e_{sf}. The cycle would contain the node v and two additional dummy nodes d_0, d_1 with the edges

$(v, d_0), (d_0, d_1)$ and (d_1, v). The result of this process is illustrated in Figure 6.17.

Two dummy nodes are introduced instead of one, since the edges from v and back to v should not be treated as a multi-edge. Secondly, the cycle would get its own face during the algorithm such that each single dummy edge gets a relative length of 1. Drawings as in Figure 6.17c are the result. Only the cycle dummies have to be removed and the vertex v is connected with a self-loop containing only horizontal and vertical segments (see Figure 6.17d).

The dummy cycle creation could be implemented in an intermediate processor, which is added to the preprocesors of the planarization. Furthermore, the back transformation to the self-loop could also be done in an intermediate processor after the dummy removal processors during the postprocessing of the compaction.

Chapter 7

Conclusion

To complete this thesis an evaluation is given in order to compare the different KLay layout algorithms with each other. In that context a upward planarity and a TSM-based algorithm of OGDF are considered. Finally, a short summary is given.

7.1 Evaluation

In the following some drawings are presented that are computed with different layout algorithms. These drawings are exported from KIELER. KIELER contains a graph analysis tool. The following properties were measured with that tool:

- ▻ Number of edge crossings.
- ▻ Number of bend-points.
- ▻ Used drawing area which is represented by the geometric mean of the width and height in pixel.
- ▻ Aspect ratio.
- ▻ Edge lengths in pixel which is divided into the average edge length and the maximal edge length of the drawing.

Firstly, the algorithms KLay Force, KLay Layered, and KLay Planar are compared with each other. In that process a drawing D_1 containing a tree is arranged. A second diagram D_2 with few edges is arranged with the KLay algorithms and the orthogonal layout algorithm of OGDF. Finally, a third diagram D_3 containing a mixed graph is arranged with KLay Layered, KLay Planar, and the upward planarity algorithm of OGDF. If not further mentioned each layout algorithm is performed with minimum spacing between the nodes of 20 pixels.

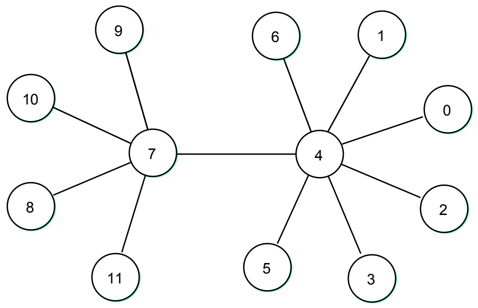
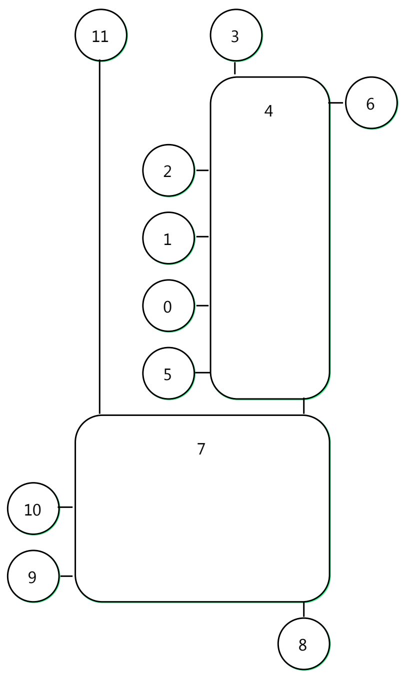
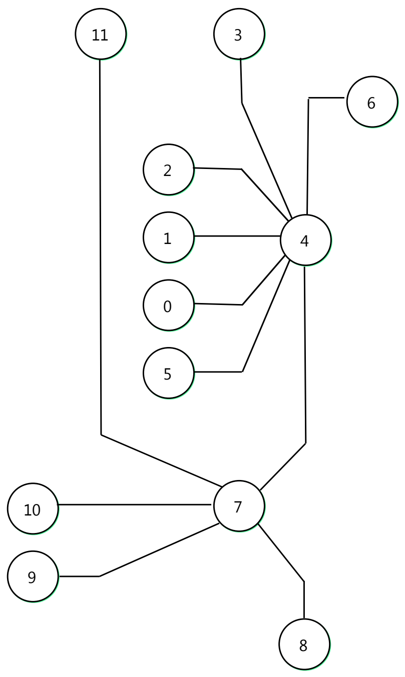


Figure 7.1. Drawing D_1 computed with KLay Force with Fruchterman and Reingold strategy, used 10000 iteration and a minimum space of 40.



a. Orthogonal drawing of D_1 with Giotto strategy.



b. Quasi orthogonal drawing of D_1 with Quod strategy.

Figure 7.2. KLay Planar with different high-degree strategies.

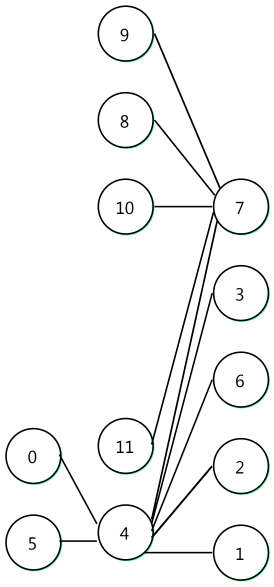


Figure 7.3. Drawing D_1 computed with KLay Layered in right direction.

Table 7.1. Analysis of the tree drawing D_1 computed with different KLay algorithms. It consists of 12 nodes, 11 edges, and an average node degree of 1.83.

Layout Algorithm	Crossings	Bends	Area	Aspect Ratio	Edge Lengths
KLay Force	0	0	300	1.56	$(68, 94)$
KLay Layered	0	0	271	0.45	$(142, 300)$
KLay Planar (Giotto)	0	0	384	0.59	$(32, 260)$
KLay Planar (Quod)	0	9	384	0.59	$(132, 366)$

The KLay Force layout algorithm turns out to perform the best results. The computed drawing contains no bend-points or edge crossings. Secondly, drawings are clearly structured (see Figure 7.1). KLay Layered divides D_1 into three layers and produces the most compact drawing with bad readability, however. The drawings computed with KLay Planar are well structured. The disadvantage at using the Giotto strategy are the adjusted node sizes, but there are no bends. The Quod approach purchases unchanged node sizes with bend-points and with increased edge lengths.

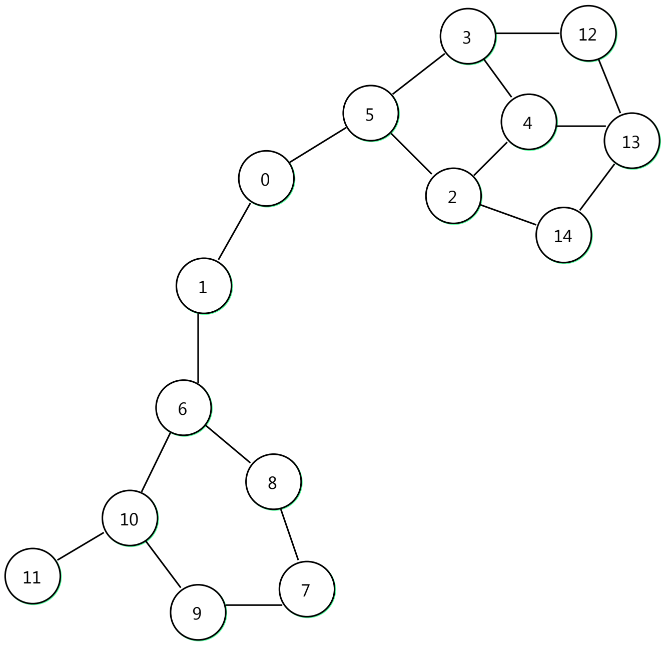


Figure 7.4. Drawing D_2 computed with KLay Force with Fruchterman and Reingold strategy.

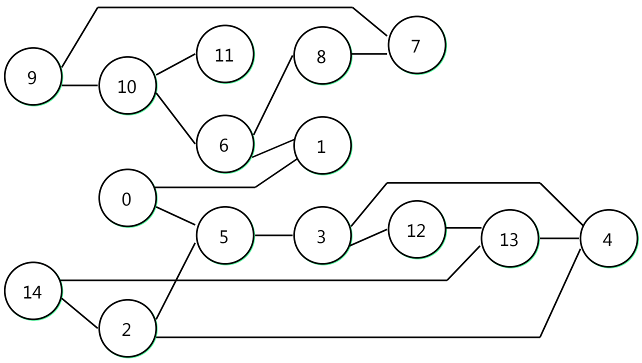


Figure 7.5. Drawing D_2 computed with KLay Layered.

KLay Force reaches again a clearly structured result after the arrangement of the diagram D_2. However, the drawing requires the most space. The KLay Layered algorithm tries to form a direction in the drawing and causes one edge crossing and 7 bends.

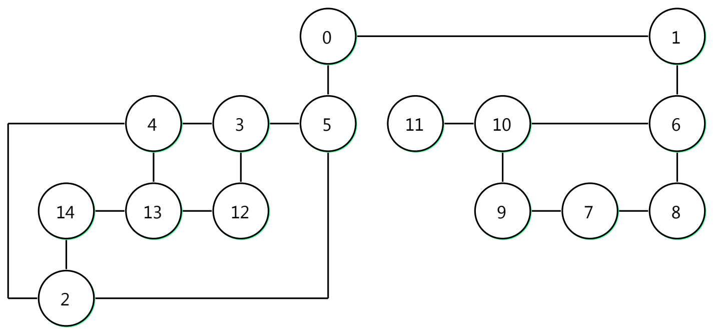


Figure 7.6. Drawing D_2 computed with OGDF orthogonal algorithm.

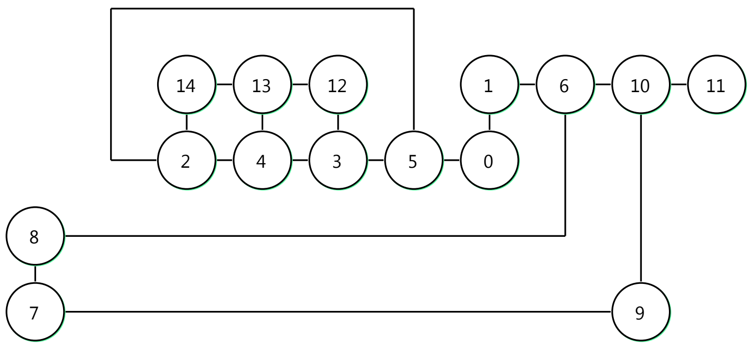


Figure 7.7. Drawing D_2 computed with KLay Planar.

Table 7.2. Analysis of different drawings D_2 with different layout algorithms. The graph consists of 15 nodes, 18 edges, and an average node degree of 2.4.

Layout Algorithm	Crossings	Bends	Area	Aspect Ratio	Edge Lengths
KLay Force	0	0	437	1.03	$(40, 48)$
KLay Layered	1	7	312	1.85	$(87, 320)$
OGDF Orthogonal	0	3	318	2.09	$(58, 260)$
KLay Planar	0	4	305	2.58	$(79, 410)$

The OGDF Orthogonal and the KLay Planar algorithms both produce compact drawings. They computed different embeddings without edge crossings and 0, respectively 1 bend-point. If the number of edges counts approximate twice the number of nodes, the nodes of the given graph can be put really compactly. The disadvantage of the rectangular shape problem of KLay Planar, mentioned in Section 6.4, can be well observed. The edges $(7,9)$ and $(6,8)$ are drawn really long. If such problem could be avoided, much more compact results would be computed.

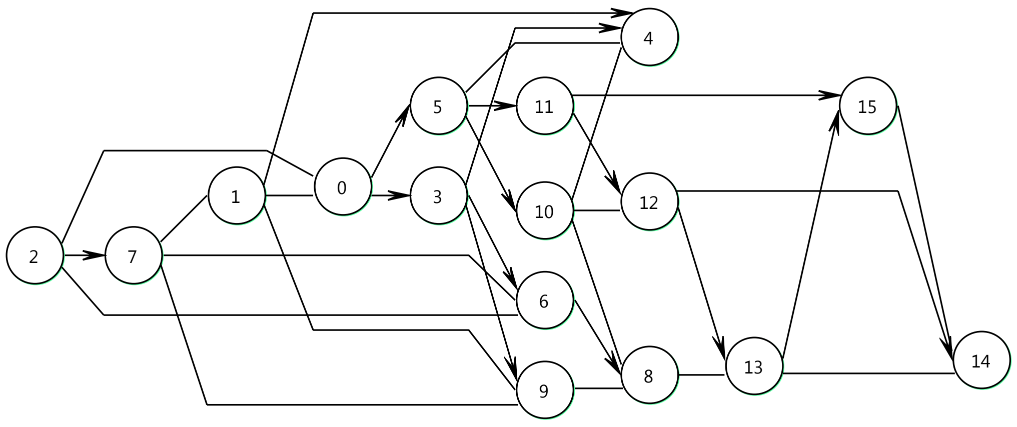


Figure 7.8. Drawing D_3 computed with KLay Layered .

Table 7.3. Analysis of different drawings of D_3 computed with different layout algorithms. The graph consists of 16 nodes, 30 edges, and an average node degree of 3.75.

Layout Algorithm	Crossings	Bends	Area	Aspect Ratio	Edge Lengths
KLay Layered	12	13	431	2.44	$(128, 333)$
OGDF Upward Planarity	12	26	715	0.41	$(257, 1256)$
KLay Planar	3	20	459	0.72	$(155, 1010)$

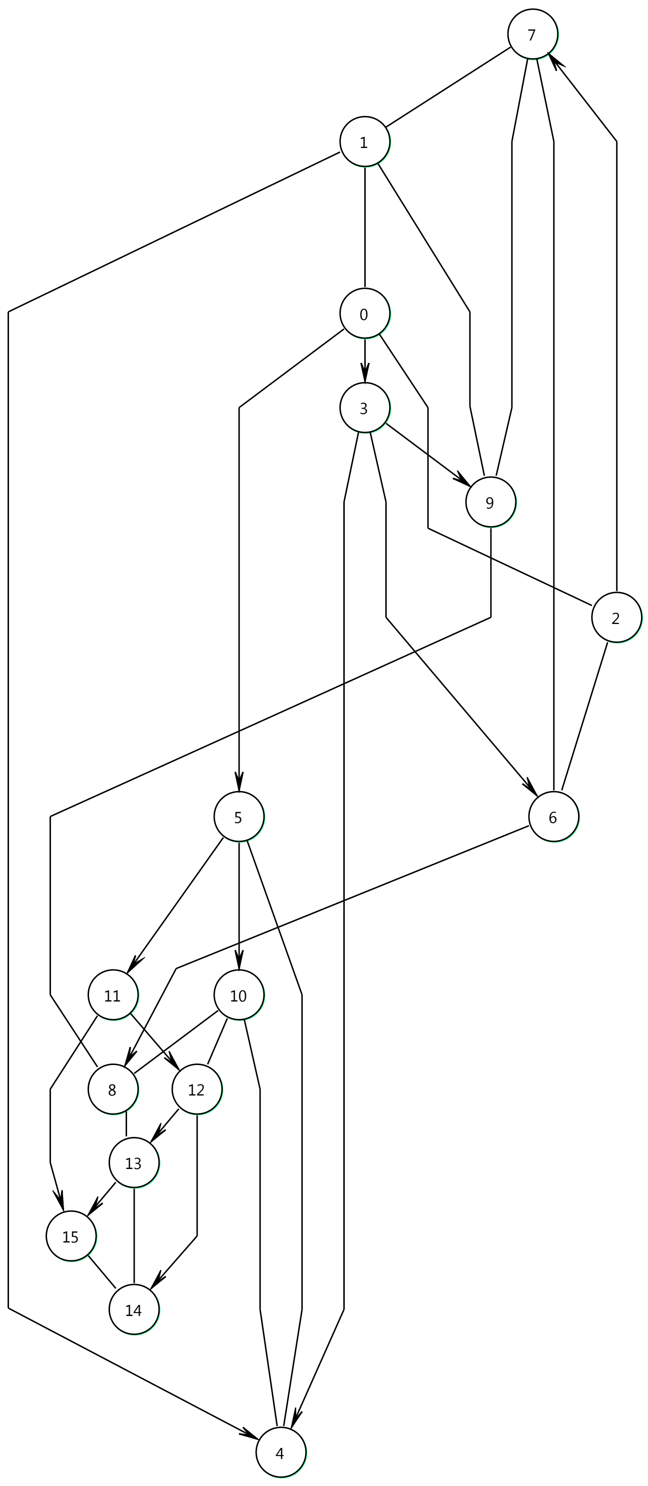


Figure 7.9. Drawing D_3 computed with OGDF upward planarity.

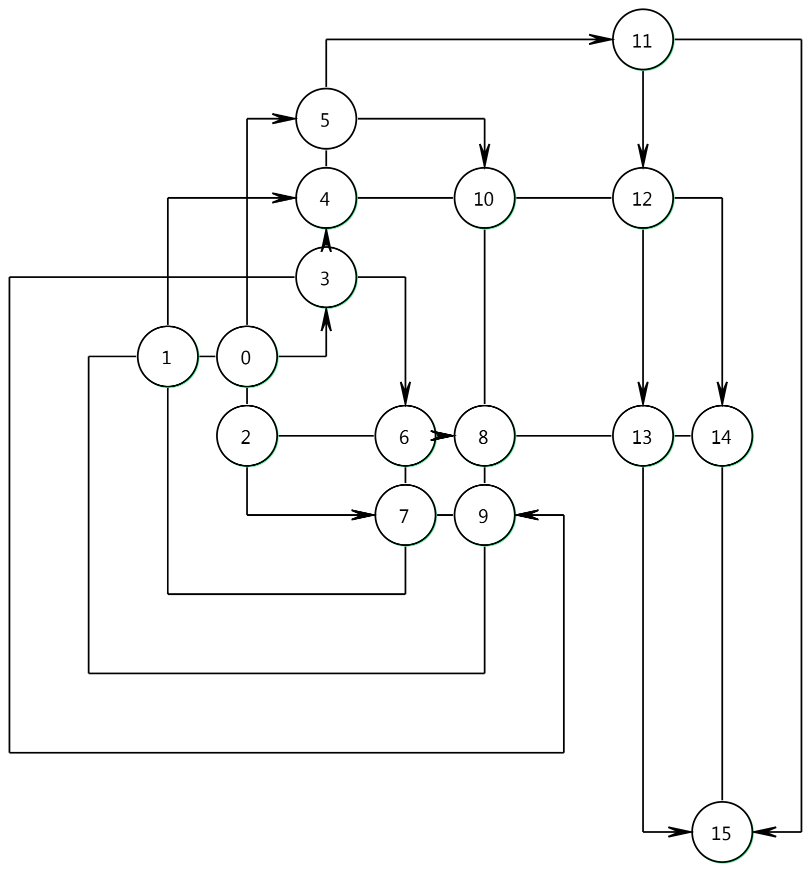


Figure 7.10. Drawing D_3 computed with KLay Planar.

D_3 is a mixed graph which can be drawn well by the layered algorithm. The flow from the single source node on the left to the single target nodes on the right is presented in a nice way. However, the resulting drawing contains 12 edge crossings which disturb the reading flow. The upward planarity drawing requires the most drawing area and also 12 edge crossings. Even if it tries to show a flow, it tends to fail. The reason for that are the undirected edges in the given graph.

The drawing computed with KLay Planar is clearly structured and requires only 3 edge crossings. However, the flow is not presented in a suitable way. The second disadvantage results from the bend-points that further blow up the drawing. Many bend-points arise when the graph contains many edges compared to the number of nodes, here a node degree of 3.75.

The algorithm of KLay Planar calculates drawings with arbitrary aspect ratio. However, in some applications this criterion is important. Thus, a layout option would be useful to determine a horizontal or a vertical orientation of the drawing. This extension could be done by simply rotating the grid, if it is in the wrong orientation.

Altogether, the implemented TSM algorithm computes results with really few edge crossings in comparison to other algorithms. In addition, the algorithms for the orthogonalization and for the compaction produce compact drawings with few bend-points with respect to the previously calculated embedding. Fulfilling such aesthetics criteria leads to clearly structured drawings. Hence, using this algorithm makes modeling more efficient.

The separation of the three TSM parts builds a base for the properties of expandability, exchangeability and maintainability. Besides that separation the generic architecture of KLay Planar enhances these properties as well. Intermediate processors and phases split the algorithm in smaller parts that are easier to handle than the algorithm in a whole.

Even if the algorithm is divided into separate phases each phase still consists of a complex structure. Thus, fundamental knowledge background is needed to maintain or expand them. Furthermore, in comparison to other drawings the orthogonal grid drawings require relatively much drawing space. This disadvantage can arise when the graph contains many edges relative to the number of nodes. This is caused by the orthogonal grid drawing and by the high importance of few edge crossings.

7.2 Summary

The considered TSM algorithm is suited for the computation of orthogonal grid drawings. The concepts of the planarization and orthogonalization were described and the technique of the implemented compaction were discussed in more detail. The standard TSM algorithm was extended to be able to handle high-degree nodes with two different strategies. For drawings in tree shape the KLay Force approach is the better choice and for many drawings containing mixed graphs directed approaches such as KLay Layered or OGDF upward planarity are more suitable, since they represent a flow between the graph nodes.

The aesthetics criteria of that algorithm can be further optimized by the implementation of some of the described extensions. For instance, to produce more compact drawings an optimal embedding could be determined. A different possibility could be to take other compaction algorithms that do not restrict the input graph with faces in rectangular shapes. Self-loops and multi-edges could be implemented easily with the presented methods. In order to handle directed graphs mixed upward planarization could be realized.

Additionally, directed edges, hypergraphs and port constraints are considered separately. Chimani, Gutwenger, Mutzel, Spönemann and Wong presented a paper in 2011 where these three elements are combined. They investigated directed hypergraphs with port constrains and showed how edge crossings could be minimal in this context [Chi+11].

Bibliography

[AMO93] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. Network Flows. Englewood Cliffs, NJ: Prentice Hall, 1993.

[Bad+10] Melanie Badent, Michael Baur, Ulrik Brandes, and Sabine Cornelsen. "'Leftist Canonical Ordering'". In: *Proceedings of the 17th international symposium on Graph Drawing*. Vol. 5849. LNCS. Springer Berlin / Heidelberg, 2010, pp. 159–170.

[Bin+02] Carla Binucci, Walter Didimo, Giuseppe Liotta, and Maddalena Nonato. Labeling Heuristics for Orthogonal Drawings. 2002.

[BM04] John Boyer and Wendy Myrvold. "On the cutting edge: simplified $o(n)$ planarity by edge addition". In: *Journal of Graph Algorithms and Applications* 8.3 (2004), pp. 241–273.

[Car12] John Julian Carstens. "'Node and Label Placement in a Layered Layout Algorithm'". Master Thesis. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Sept. 2012.

[Chi+07] Markus Chimani, Carsten Gutwenger, Michael Jünger, Karsten Klein, Petra Mutzel, and Michael Schulz. The Open Graph Drawing Framework. Poster at the 15th international iymposium on Graph Drawing. Sydney, 2007.

[Chi+10] Markus Chimani, Carsten Gutwenger, Petra Mutzel, and Hoi-Ming Wong. "Layer-free upward crossing minimization". In: *Journal of Experimental Algorithmics* (2010), 2.2:2.1–2.2:2.27.

[Chi+11] Markus Chimani, Carsten Gutwenger, Petra Mutzel, Miro Spönemann, and Hoi-Ming Wong. "'Crossing Minimization and Layouts of Directed Hypergraphs with Port Constraints'". In: *Proceedings of the 18th international symposium on Graph drawing*. Vol. 6502. LNCS. Springer Berlin / Heidelberg, 2011, pp. 141–152.

[Cla10] Ole Claußen. "'Implementing an Algorithm for Orthogonal Graph Layout'". Bachelor Thesis. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Sept. 2010.

[Dij59] Edsger W. Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische Mathematik* 1 (1959), pp. 269–271.

[Döh10] Philipp Döhring. "'Algorithmen zur Layerzuweisung'". Bachelor Thesis. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Sept. 2010.

[Ead+91] Peter Eades, Wei Lai, Kazuo Misue, and Kozo Sugiyama. "Preserving the mental map of a diagram". In: *Proceedings of the 1st international symposium on Computational Graphics and Visualization Techniques*. International Institute for Advanced Study of Social Information Science, Fujitsu Limited, 1991, pp. 34–43.

[EGB06] Thomas Eschbach, Wolfgang Guenther, and Bernd Becker. "Orthogonal hypergraph drawing for improved visibility". In: *Journal of Graph Algorithms and Applications* 10.2 (2006), pp. 141–157.

[EK72] Jack Edmonds and Richard M. Karp. "Theoretical improvements in algorithmic efficiency for network flow problems". In: *Journal of ACM* 19.2 (Apr. 1972), pp. 248–264.

[EKS03] Markus Eiglsperger, Michael Kaufmann, and Martin Siebenhaller. "A topology-shape-metrics approach for the automatic layout of UML class diagrams". In: *Proceedings of the ACM symposium on Software Visualization*. SoftVis '03. San Diego, California: ACM, 2003, 189–ff.

[Ell+02] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C. North, and Gordon Woodhull. "Graphviz - open source graph drawing tools". In: *LNCS* 2265 (2002), pp. 594–597.

[FK96] Ulrich Fößmeier and Michael Kaufmann. "Drawing High Degree Graphs with Low Bend Numbers". In: *Proceedings of the 4th international symposium on Graph Drawing*. Springer Berlin / Heidelberg, 1996, pp. 254–266.

[FPP90] H. De Fraysseix, J. Pach, and R. Pollack. "How to draw a planar graph on a grid". In: *Combinatorica* 10.1 (1990), pp. 41–51.

[Fuh+10] Hauke Fuhrmann, Miro Spönemann, Michael Matzen, and Reinhard von Hanxleden. "Automatic layout and structure-based editing of UML diagrams". In: *Proceedings of the 1st workshop on Model Based Engineering for Embedded Systems Design*. Springer Berlin / Heidelberg, 2010.

[Fuh11] Hauke Fuhrmann. "'On the Pragmatics of Graphical Modeling'". Dissertation. Kiel: Christian-Albrechts-Universität zu Kiel, Faculty of Engineering, 2011.

[GKM07] Carsten Gutwenger, Karsten Klein, and Petra Mutzel. "Planarity testing and optimal edge insertion with embedding constraints". In: *Proceedings of the 14th international symposium on Graph Drawing*. Vol. 4372. LNCS. Springer Berlin / Heidelberg, 2007, pp. 126–137.

[GM04] Carsten Gutwenger and Petra Mutzel. "'Graph Embedding with Minimum Depth and Maximum External Face'". In: *Proceedings of the 11th international symposium on Graph Drawing*. Vol. 2912. LNCS. Springer Berlin / Heidelberg, 2004, pp. 259–272.

[GT02] Ashim Garg and Roberto Tamassia. "On the computational complexity of upward and rectilinear planarity testing". In: *SIAM Journal of Computing* 31.2 (2002), pp. 601–625.

[HL91] Michael D. Hutton and Anna Lubiw. "Upward planar drawing of single source acyclic digraphs". In: *Proceedings of the 2nd ACM symposium on Discrete Algorithms*. 1991, pp. 203–211.

[HT74] John Hopcroft and Robert Tarjan. "Efficient planarity testing". In: *Journal of the ACM* 21.4 (1974), pp. 549–568.

[JLM98] Michael Jünger, Sebastian Leipert, and Petra Mutzel. "Level planarity testing in linear time". In: *Proceedings of the 6th international symposium on Graph Drawing*. Vol. 1547. LNCS. Springer Berlin / Heidelberg, 1998, pp. 224–237.

[Kan92] G. Kant. "Drawing planar graphs using the lmc-ordering". In: *Proceedings of the 33rd annual symposium on Foundations of Computer Science*. SFCS '92. Washington, DC, USA: IEEE Computer Society, 1992, pp. 101–110.

[Kan96] Goos Kant. "Drawing planar graphs using the canonical ordering". In: *Algorithmica* 16.1 (1996), pp. 4–32.

[KB92] Goos Kant and Hans Bodlaender. "'Triangulating planar graphs while minimizing the maximum degree'". In: *Algorithm Theory - SWAT 1992*. Vol. 621. LNCS. Springer Berlin / Heidelberg, 1992, pp. 258–271.

[Ker07] Thorsten Kerkhof. "'Algorithmen zur Bestimmung von guten Graph-Einbettungen für orthogonale Zeichnungen'". Diploma Thesis. 2007.

[KM98] Gunnar W. Klau and Petra Mutzel. Quasi-orthogonal drawing of planar graphs. Technical Report MPI-I-98-1-013. Saarbrücken: Max-Planck-Institut für Informatik, 1998.

[KM99a] Gunnar Klau and Petra Mutzel. "'Combining Graph Labeling and Compaction'". In: *Proceedings of the 7th international symposium on Graph Drawing*. Vol. 1731. LNCS. Springer Berlin / Heidelberg, 1999, pp. 27–37.

[KM99b] Gunnar Klau and Petra Mutzel. "'Optimal Compaction of Orthogonal Grid Drawings (Extended Abstract)'". In: *Integer Programming and Combinatorial Optimization*. Vol. 1610. LNCS. Springer Berlin / Heidelberg, 1999, pp. 304–319.

[Kut10] Christian Kutschmar. "'Planarisierung von Hypergraphen'". Bachelor Thesis. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Sept. 2010.

[KW01] Micheal Kaufman and Dorothea Wagner. Drawing Graphs: Methods and Models. LNCS. Springer Berlin / Heidelberg, 2001.

[MM96] Kurt Mehlhorn and Petra Mutzel. "On the embedding phase of Hopcroft and Tarjan planarity testing algorithm". In: *Algorithmica* 16 (1996), pp. 233–242.

[Mut+98] Petra Mutzel et al. "'A Library of Algorithms for Graph Drawing'". In: *Proceedings of the 6th international symposium on Graph Drawing*. Vol. 1547. LNCS. Springer Berlin / Heidelberg, 1998, pp. 456–457.

[Mut05] Petra Mutzel. Zeichnen von Diagrammen - Theorie und Paxis. Max-Planck-Institut für Informatik Saarbrücken, 2005.

[Mäk90] Erkki Mäkinen. "How to draw a hypergraph". In: *International Journal of Computer Mathematics* 34 (1990), pp. 177–185.

[Piz06] Maurizio Pizzonia. "Minimum depth graph embeddings and quality of the drawings: an experimental analysis". In: *Proceedings of the 13th international symposium on Graph Drawing*. GD'05. Limerick, Ireland: Springer Berlin / Heidelberg, 2006, pp. 397–408.

[PT00] Maurizio Pizzonia and Roberto Tamassia. "'Minimum Depth Graph Embedding'". In: *Algorithms - ESA 2000*. Vol. 1879. LNCS. Springer Berlin / Heidelberg, 2000, pp. 356–367.

[San04] Georg Sander. "Layout of directed hypergraphs with orthogonal hyperedges". In: *Proceedings of the 11th international symposium on Graph Drawing*. Vol. 2912. LNCS. Springer Berlin / Heidelberg, 2004, pp. 381–386.

[Sch11] Christoph Daniel Schulze. "'Optimizing Automatic Layout for Data Flow Diagrams'". Diploma Thesis. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, July 2011.

[SFH09] Miro Spönemann, Hauke Fuhrmann, and Reinhard von Hanxleden. Automatic Layout of Data Flow Diagrams in KIELER and Ptolemy II. Technical Report 0914. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, 2009.

[Spö09] Miro Spönemann. "'On the Automatic Layout of Data Flow Diagrams'". Diploma Thesis. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Mar. 2009.

[STT81] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. "Methods for visual understanding of hierarchical system structures". In: *IEEE Transactions on Systems, Man and Cybernetics* 11.2 (1981), pp. 109–125.

[Tam87] Roberto Tamassia. "On embedding a graph in the grid with the minimum number of bends". In: *SIAM Journal of Computing* 16.3 (June 1987), pp. 421–444.

[TDBB88] Roberto Tamassia, Giuseppe Di Battista, and Carlo Batini. "Automatic graph drawing and readability of diagrams". In: *IEEE Transactions on Systems, Man and Cybernetics* 18.1 (Jan. 1988), pp. 61–79.

[Yan78] Mihalis Yannakakis. "Node-and edge-deletion np-complete problems". In: *Proceedings of the 10th annual ACM symposium on Theory of computing*. San Diego, California, United States: ACM, 1978, pp. 253–264.

[Di +94] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. "Algorithms for drawing graphs: an annotated bibliography". In: *Computational Geometry: Theory and Applications* 4 (June 1994), pp. 235–282.

[Di +99] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. Graph drawing: algorithms for the visualization of graphs. An Alan R. Apt book. Prentice Hall, 1999.